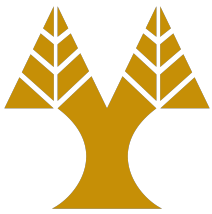
Thesis Dissertation

EXPLORING SWIFT AS A SERVERLESS LANGUAGE ON OPENWHISK

Andreas Loizides

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Exploring Swift as a Serverless Language on OpenWhisk

Andreas Loizides

Supervisor
Dr. Haris Volos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2023

Acknowledgements

I would like to express my profound gratitude to a number of people whose support and contributions were invaluable throughout the course of this thesis.

First and foremost, I would like to thank my supervisor and mentor, Dr. Haris Volos, for his unwavering guidance and continuous support. Whenever issues arose, Dr. Volos was always ready to thoroughly evaluate the problem, imparting not only his expert knowledge but also a sense of shared purpose and curiosity. His dedication to my research gave me the confidence and motivation to navigate the complex challenges that arose during this thesis. It truly felt like we shared the same research goals and the drive to satisfy curiosity which is crucial in a research context. His care and attention were instrumental in enabling me to reach my research goals.

I am deeply grateful to my university for granting me access to the CloudLab's infrastructure. This access was not just a privilege but a necessity. The comprehensive results and the eventual completion of this thesis would not have been possible without it. It provided a robust platform to explore Swift as a serverless language on OpenWhisk, contributing significantly to the outcomes of my research.

Lastly, but by no means least, I would like to extend my heartfelt thanks to my parents. Their unconditional faith in my abilities and their financial support provided a stable foundation for my academic journey. It allowed me to comfortably pursue my interests and devote my energy to my research. Their sacrifices and belief in my vision were a beacon of light that guided me through the most challenging phases of this thesis.

To all of you, thank you. This accomplishment would not have been possible without your collective support and belief.

Summary

This thesis delves into the potential of Swift in the rapidly growing field of serverless computing. Serverless computing has gained significant attention due to its scalability and cost-effectiveness, making the choice of programming language a crucial factor in this context.

Swift, a language renowned for its popularity on Apple platforms, is known for its speed, safety, and reliability. Despite its widespread use in the Apple ecosystem, Swift's potential as a serverless language remains largely unexplored. This thesis aims to bridge this gap, driven by Swift's promise of speed and safety.

The methodology for evaluating Swift's capabilities as a serverless language involves a two-pronged approach. Initially, a qualitative comparison is conducted with popular serverless languages to investigate potential performance benefits Swift might provide. Subsequently, Swift's capabilities are evaluated through a case study, comparing a serverless to a monolithic implementation of a synchronization system.

The key findings from the research indicate that while Swift holds promise, its lack of community support and unstable Linux support with various functionalities missing, make it unsuitable for production use as a systems language, let alone serverless. Furthermore, the case study highlighted OpenWhisk's need to support intra-concurrency to fully utilize the hardware and achieve real concurrency in invoking actions.

In terms of benefits, Swift is enjoyable to write in and is a very expressive language. Most errors are caught at compile time, saving critical time and effort. Its seamless Copy-on-Write (CoW) support has the potential to greatly benefit memory and performance-critical environments, such as serverless. However, the limitations, particularly the lack of community support and ambiguity in many features in regards to their Linux support, pose significant challenges.

In conclusion, this thesis provides a comprehensive exploration of Swift as a serverless language, contributing to the ongoing discourse in this area and highlighting areas for further research.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The landscape of computing has witnessed a paradigm shift in recent years, with the emergence of serverless computing. Serverless computing enables developers to focus on writing their application's code without worrying about underlying infrastructure management, provisioning, and scaling. As a result, serverless computing has gained popularity as a cost-effective and scalable alternative to traditional web and application development practices. Some of the many advantages of serverless computing include auto-scaling, elimination of idle server costs, pay-per-use pricing, and seamless scaling to handle fluctuating workloads.

In the realm of serverless computing, one's choice of a programming language is crucial. It affects not only the performance but also the ease of development and maintenance of serverless applications. The focus of this thesis is the Swift programming language, which has demonstrated its potential for speed, safety, and reliability, particularly in Apple platforms.

1.2 Swift in Serverless Computing

Swift is a statically typed, general-purpose, multi-paradigm programming language developed by Apple Inc. for iOS, macOS, watchOS, and other platforms. Since its release in 2014, Swift has gained significant popularity and achieved acclaim for its efficiency, safety, and ease of use. The Swift programming language demonstrates potential as a viable serverless language due to its inherent speed, safety, and its compatibility with OpenWhisk, which is the selected environment for this study.

1.3 Choice of Serverless Environment: OpenWhisk

OpenWhisk [12] is an event-driven, open-source serverless computing platform that supports a wide range of programming languages, including Swift, Python, Java, and Node.js. The reasons for selecting OpenWhisk as the environment for this study include its widespread adoption in serverless computing contexts, its flexibility, and its support for Swift as a first-class language. For the purpose of the comparative analysis in this thesis, we will examine the advantages and disadvantages of using Swift against other popular serverless computing languages such as Python, Java, and Node.js, all of which have established themselves in this domain.

1.4 Case Study: Synchronization System in eCommerce

A case study detailing a monolithic and serverless implementation of a synchronization system in the eCommerce industry will be presented in this thesis. The synchronization system serves a critical function in ensuring consistency and accuracy across online stores, as it aids in maintaining inventory, order management, and customer accounts up-to-date. This case study will help demonstrate Swift's viability as a serverless language in a real-world scenario and contribute to the broader discussion on the appropriateness of Swift for serverless computing.

1.5 Objective

The overarching objective of this thesis is to provide a comprehensive analysis of Swift as a serverless language, examining its potential for performance and ease of development within the OpenWhisk environment. By conducting an in-depth comparison with other common serverless languages and illustrating Swift's applicability through a practical case study, this thesis aims to contribute valuable insights and perspectives, further advancing the understanding of Swift's role in serverless computing.

1.6 Research Questions

With the background and context established, this thesis will attempt to answer the following research questions:

1.6.1 Performance Comparison

How does the performance of Swift in serverless computing compare to other languages, such as Python, Java, and Node.js? This involves examining any potential benefits Swift could provide over other languages in a serverless context, as well as any possible performance limitations or challenges it may present.

1.6.2 Ease of Development

To what degree is Swift an accessible and efficient language for developers in a serverless environment? This question will be addressed through evaluating the development experience of Swift compared to other serverless languages, investigating aspects such as language features, syntax, tooling, and libraries that impact the ease of development of serverless functions in Swift.

Chapter 2

Updating the Swift runtime

2.1 Introduction

In this chapter, we delve into the process of updating the Swift Runtime for OpenWhisk, a critical component of our exploration of Swift as a serverless language. One of the key elements of this exploration is the transition from Swift 5.4 to Swift 5.8. This version update brings with it a wealth of new features that greatly enhance the capabilities of Swift as a serverless language, including concurrency support (via async/await constructs), structured concurrency, the actor model, and SwiftNIO 2.

These features provide significant benefits when it comes to the development and deployment of serverless applications. In particular, they have a profound impact on our synchronization system case study. For instance, the async/await constructs, part of the concurrency support, simplify the handling of asynchronous tasks, making the code easier to write and understand. This was particularly beneficial in the development of both a monolithic and a serverless implementation of the synchronization system, which heavily relied on these constructs.

Runtimes in OpenWhisk are the backbone that enables the execution of actions in the serverless environment. Key to this is understanding the Action Interface and the ActionLoop Proxy. The ActionLoop proxy simplifies the development of new runtimes by implementing most of the Action Interface specification, making it possible to create a compliant and efficient runtime with fewer resources.

Updating the Swift runtime in itself was not complicated, but understanding how it all works together was a challenge. In this chapter we dive into the details of how this is achieved, to provide a reference point for future researches in understanding how OpenWhisk runtimes work, and particularly the Swift runtime.

2.2 Overview of the officially supported Swift runtime in OpenWhisk

Apache OpenWhisk supports a variety of programming languages for writing actions, through the use of specific runtimes. As per the official documentation, the following runtimes are currently supported [12]:

- .Net: OpenWhisk runtime for .Net Core 2.2.
- Go: OpenWhisk runtime for Go.
- Java: OpenWhisk runtime for Java 8 (OpenJDK 8, JVM OpenJ9).
- JavaScript: OpenWhisk runtime for Node.js v10, v12, and v14.
- PHP: OpenWhisk runtime for PHP 8.0, 7.4, and 7.3.
- Python: OpenWhisk runtime for Python 2.7, 3, and a 3 runtime variant for AI/ML (including packages for Tensorflow and PyTorch).
- Ruby: OpenWhisk runtime for Ruby 2.5.
- Swift: OpenWhisk runtime for Swift 5.1, 5.3 and 5.4.

These runtimes are officially supperted by OpenWhisk, meaning one can deploy an action in those languages by specifying with it the -kind option. Any docker image can be used as a runtime to deploy actions, as long as it adheres to the Action Interface and is a Docker image publicly available on DockerHub. Our updated version of Swift (v5.8) is publicly available on dockerhub with the image tag andreas16700/swift58-1. Swift 5.4 lacks behind, notably Swift 5.5 which brings numerous crucial improvement which we'll explain below.

2.3 Swift 5.5 Enhancements and their Impact

We elucidate how these enhancements facilitated the development and benchmarking of a monolithic and a serverless implementation in the synchronization system case study, with a focus on the substantial use of the async/await constructs in the pre-existing monolithic implementation.

2.3.1 Concurrency Support and Async/Await

In Swift 5.5, the introduction of concurrency support, in particular, the async/await constructs, revolutionized the way asynchronous code is written and understood. The async/await model allows for the execution of asynchronous tasks in a manner that closely resembles synchronous code, eliminating the complexity of nested callbacks and error-prone manual threading.

In the context of our synchronization system case study, this feature had a profound impact. The pre-existing monolithic implementation was built with heavy use of asynchronous constructs. With the transition to Swift 5.5, we could leverage the async/await model to handle these tasks in a more readable and maintainable way, making the code easier to comprehend and modify.

2.3.2 Structured Concurrency

Structured concurrency, another significant feature introduced in Swift 5.5, provides a way to manage and control asynchronous tasks. It introduces new concepts like task groups and task cancellation, which can be used to group related tasks and cancel them if they are no longer needed.

For the synchronization system case study, structured concurrency helped ensure that asynchronous tasks were well-managed and tidied up after completion. This reduced the risk of memory leaks and made the system more efficient.

2.3.3 Actor Model

The actor model in Swift 5.5 provides a way to handle shared mutable state in concurrent settings. It isolates state to individual actors and ensures that only one piece of code can access that state at a time.

In the context of the synchronization system, the actor model was invaluable in managing shared resources, avoiding race conditions and thus increasing the robustness of the system.

2.3.4 SwiftNIO 2

SwiftNIO 2, a low-level tool for building high-performance networking applications, brought about enhancements that improved the performance and functionality of our serverless implementation. It enabled the efficient handling of networking tasks in the synchronization system, resulting in a performance boost and more robust networking capabilities.

2.3.5 Impact on the Synchronization System Case Study

Overall, the enhancements in Swift 5.5 significantly improved the development and benchmarking process for both the monolithic and serverless implementations of the synchronization system. The async/await constructs, structured concurrency, the actor model, and SwiftNIO 2 all contributed to a more efficient, maintainable, and robust system. They allowed us to write cleaner, more understandable code, manage asynchronous tasks more effectively, avoid common concurrency issues, and handle networking tasks more efficiently. This demonstrated the potential of Swift 5.5 as a powerful language for serverless computing.

2.4 Understanding Runtimes in OpenWhisk

We delve into the function and significance of runtimes in OpenWhisk, paying special attention to the role of the ActionLoop proxy and Action Interface. We clarify how the ActionLoop proxy aids in the creation of compliant runtimes by implementing most of the specification, enabling the development of an efficient runtime in a short span of time.

2.5 Actions in OpenWhisk

2.5.1 Action Interface in OpenWhisk

The core concept in the OpenWhisk environment is an Action. Actions are the smallest deployable units of code and primarily constitute two components: the user function and its corresponding proxy. The user function represents the code logic to be executed, and the proxy serves as an intermediary, implementing a canonical protocol to enable the user function to interact with the OpenWhisk platform [10]. The proxy is the runtime. Anything can be a runtime, as far as OpenWhisk is concerned, as long as it implements the Action Interface, which specifies a behavior that OpenWhisk expects. In other words, a runtime can be thought of a black box (docker container) that listens on two HTTP routes: /init and /run on port 8080, which are used for deploying and executing actions, respectively. OpenWhisk calls on the /init route with the action code to initialize an action. Whenever the action is invoked, OpenWhisk will send a POST request to the /run route with the action's parameters as payload. The runtime should return the result of the action. The behavior is shown abstractly in figure 2.1.

The /init endpoint is tasked with container initialization and accepts a POST request consisting of a JSON object that includes the action name, function to be executed, the source code, and several environment variables. This initialization happens exactly once

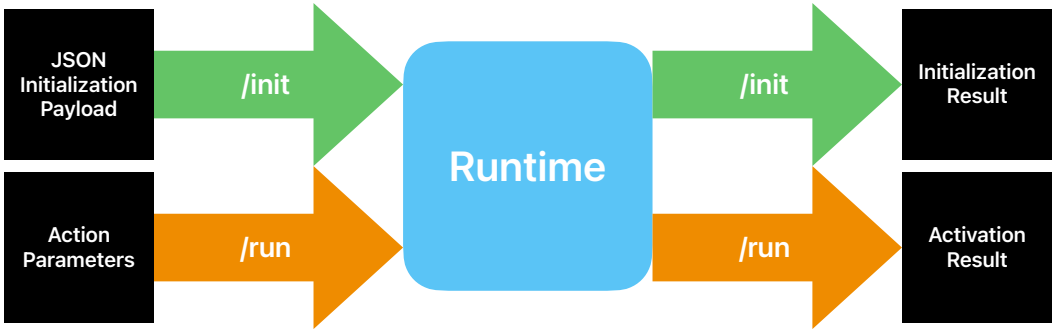


Figure 2.1: Abstract representation of the Action Interface in OpenWhisk.

and must be completed within a predefined time limit set by the platform [10].

Once initialization is completed successfully, the runtime transitions to a state where it can activate the function. The /run endpoint is responsible for this task. It receives an HTTP POST request with a new activation context and the function's input parameters. The route needs to accept a JSON object and respond with one, adhering to the OpenWhisk platform's prescribed schema. Every action in OpenWhisk has a set time limit, within which the activation must complete. If the function executes successfully, the route responds with 200 OK, and the response body is recorded as the result of the activation [12].

The runtime should flush all the logs generated during initialization and execution, adding a unique frame marker at the end of each activation log stream. This marker is necessary to avoid delayed or truncated activation logs [12].

2.5.2 ActionLoop Proxy in OpenWhisk

The ActionLoop proxy is a tool designed to simplify the process of creating new OpenWhisk runtimes. While one can develop a new runtime by following the Action Interface, the ActionLoop proxy offers a quicker and more efficient way to achieve this by implementing most of the specification out-of-the-box [11].

The ActionLoop proxy is a runtime engine, developed in Go, initially designed to support the OpenWhisk Go language runtime. However, its generic design allows it to be adapted for other language runtimes such as Swift, PHP, Python, Rust, Java, Ruby, and Crystal. It was engineered keeping both compiled and scripting languages in mind.

Using the ActionLoop proxy, one can develop a new runtime in a fraction of the time it would take to create one from scratch. This is due to the fact that the ActionLoop proxy requires the developer to write only a command line protocol instead of a full-fledged web

server, reducing complexity. Additionally, the ActionLoop proxy is known to significantly enhance the performance of existing runtimes, providing speed improvements ranging from 2x to 20x [11].

Since Swift, like all other language runtimes except Javascript, leverage it, understanding the ActioxnLoop proxy is crucial to be able to update the OpenWhisk runtime to support the latest version of Swift, enhancing the potential for Swift to be used as a Serverless language.

2.5.3 Compiling a Swift action with the ActionLoop Proxy

Runtimes are docker images. One of the features of the ActionLoop proxy is compiling an action into a binary file that can then be used by the runtime to run the action. This could be useful in debugging, as well as accelerating development. Having the ability to compile locally is crucial in the development process. Developers can immediately catch any compile-time errors, which in the context of Swift and most statically typed languages, is especially useful since many errors are caught there. This effect is more pronounced in the case of Swift, as we will see in later chapters where we explain some of Swift's safety features. Suppose the Swift runtime is built into the docker image tagged as swift-one. With the following command we can build the binary for the action that runs the fuction mainName:

```
docker run swift-one -compile mainName
```

The standard input of this command is the source file of our function. There are two cases on where that function could reside:

Single File the function is contained in a single Swift file

Package the function is part of a Swift package

In the case of the Swift package, it should be zipped and be given as standard input to the command. The Swift runtime expects to be able to unzip it, and in the resulting directory run `swift build -c release`. This is extracted into a src directory inside the docker container. In the parent folder there exists a sample swift package skeleton. In the case that the contents of the zip file are nested inside another folder, when the runtime tries to run the swift build command, unexpected behavior will occur. That is because the swift compiler when it can't find the Package.swift file in the current directory, it searches up the hierarchy to find it, resulting in it trying to build the blank swift package (which will most likely not contained the desired function). In our experience, this has caused countless hours of debugging and trying to figure out what the problem is, because of unhelpful feedback such as "function X not found". Fully understanding the inner workings of the

ActionLoop proxy and the Swift runtime was required. One could argue the simplicity the ActionLoop proxy aims to provide is lost in this. The following command produces the binary executable fun for the function fun() which is contained in the Swift package which is zipped inside myPackage.zip:

```
docker run swift-one -compile fun <myPackage.zip >fun.zip
```

fun.zip contains a folder which contains the actual binary.

The behavior of the binary is as follows: It listens on standard output for the JSON payload OpenWhisk will send it (which contains the parameters of the function). It writes to standard output the result of the function. In case that the output is a Codable type, it returns the JSON encoded text.

2.6 Process of Updating the Swift Runtime

The current Swift runtime repository [7] contains a folder called core which contains the files necessary to build the image for each swift version.

Figure 2.2: Contents of the core folder of the Swift runtime

Each folder contains the following files:

Whisk.swift A client for invoking actions, creating triggers, and most basic wsk functionalities in Swift. It is not directly needed in the runtime, but it is useful as a simple OpenWhisk SDK for swift.

build.gradle Contains some gradle options. Mainly the name of the docker image (which will be built from this folder).

CHANGELOG.md The changelog for that version.

main.swift Contains just a sample function, used for testing and an initial swift build.

Dockerfile The actual docker file for building the image. It is responsible for including the ActionLoop proxy.

Package.swift Sample package file used for testing.

swiftbuild.py This file is used as the /bin/compile tool, as specified in the ActionLoop proxy. It is responsible for creating the action binary.

swiftbuild.py.launcher.swift This is used as the /bin/compile.launcher file, as specified in the ActionLoop proxy. This file contains all supported signatures for deploying functions.

swiftbuildandlink.sh In the case of Swift, this just build the Swift executable

In order to actually update the runtime, a copy of any of the existing runtime folder will do, and requires minimal changes:

Dockerfile Simply replace the swift image tag to the latest one (FROM swift:5.4 -> FROM swift:5.8)

build.gradle Change the image tag to a new one: ext.dockerImageName = 'action-swift-v5.4' -> ext.dockerImageName = 'action-swift-v5.8'

swiftbuild.py.launcher.swift Replace the callback-based functions to async based ones

```
func _run_main<Out: Encodable>(mainFunction: ( @escaping (Out?,
 ↪  Error?) -> Void) -> Void, json: Data) {
        let resultHandler = { (out: Out?, error: Error?) in
                if let error = error {
                        _whisk_print_error(message: "Action handler
                         ↪  callback returned an error:", error:
                         ↪  error)
                        return
                }
                guard let out = out else {
```

```
                        _whisk_print_error(message: "Action handler
                        ↪  callback did not return response or
                        ↪  error.", error: nil)
                        return
                }
                do {
                        let jsonData = try
                        ↪  Whisk.jsonEncoder.encode(out)
                        _whisk_print_result(jsonData: jsonData)
                } catch let error as EncodingError {
                        _whisk_print_error(message: "JSONEncoder
                        ↪  failed to encode Codable type to JSON
                        ↪  string:", error: error)
                        return
                } catch {
                        _whisk_print_error(message: "Failed to
                        ↪  execute action handler with error:",
                        ↪  error: error)
                        return
                }
        }
        let _ = mainFunction(resultHandler)
}

func _run_main<Out: Encodable>(mainFunction: ()async -> (Out?,
↪  Error?), json: Data) async{
        let (out, error) = await mainFunction()
        if let error = error {
                _whisk_print_error(message: "Action handler callback
                ↪  returned an error:", error: error)
                return
        }
        guard let out = out else {
                _whisk_print_error(message: "Action handler callback
                ↪  did not return response or error.", error: nil)
                return
        }
        do {
                let jsonData = try Whisk.jsonEncoder.encode(out)
```

```
                _whisk_print_result(jsonData: jsonData)
        } catch let error as EncodingError {
                _whisk_print_error(message: "JSONEncoder failed to
                ↪ encode Codable type to JSON string:", error:
                ↪ error)
                return
        } catch {
                _whisk_print_error(message: "Failed to execute action
                ↪ handler with error:", error: error)
                return
        }
}
```

Now,

swiftbuild.py needs to be updated so that the code it injects calls the async function with the await keyword:

`code += f" _run_main(mainFunction: {main}, json: nil)\n"` becomes: `code += f" await _run_main(mainFunction: {main}, json: nil)\n"`

2.7 Challenges Encountered

While updating the runtime in hindsight did not require many changes, understanding how it all works together with the ActionLoop proxy to produce and run a binary required considerable effort. Issues arose frequently. Small mistakes rarely yield useful feedback. As mentioned before, a small change such as nesting the source file just one folder further, leads to confusing behavior.

For instance, consider how the current swift runtime processes an input. It reads a line from standard input, which is the JSON object OpenWhisk provides. That object, contains the parameters of the activation under the key 'value'.

```
let jsonData = try JSONSerialization.data(withJSONObject:
↪ parsed["value"] as Any, options: [])
```

Deploying and activating function without parameters is supported by openwhisk and very much possible. What happens when a Swift action is activated with no parameters? The runtime tries to serialize the [value] key which in this case does not exist. The result is that JSONSerialization's data function throws an error, failing the activation.

Chapter 3

Performance Comparison

3.1 Value Types vs Reference Types and Copy-On-Write

In programming languages, value types and reference types represent two ways to handle data. Understanding the difference between them and their performance implications is crucial to making informed decisions when choosing a language for serverless applications. Swift not only distinguishes between value and reference types, but also employs Copy-On-Write optimization for most of its value types, including collections such as Arrays, Dictionaries, and Sets.

3.1.1 Value Types

Value types are types for which the value is stored directly in the memory. When a value type is assigned or passed to a function, a separate copy of the value is created. The main

benefit of value types is that they are generally more efficient in terms of memory usage and can result in more predictable performance. In Swift, the distinction between value and reference types is handled through structs (value types) and classes (reference types).

3.1.2 Reference Types

Reference types represent data where the memory location (or reference) is stored, rather than the data itself. When a reference type is assigned or passed to a function, the reference (memory address) is copied, not the actual data. This means that different variables can point to the same data, potentially leading to unexpected behavior if the data is modified.

3.1.3 Copy-On-Write (CoW)

Swift uses Copy-On-Write (CoW) for most of its value types, including collections such as Arrays, Dictionaries, and Sets. Collections whose Element type is also a value type essentially leverage CoW for free. CoW is an optimization strategy where the copying of a value type is delayed until it's necessary, i.e., when the value needs to be modified.

This approach has several performance benefits when compared to simply copying the data every time:

- Sharing the same data among multiple instances of a collection is more memory efficient, as multiple copies of the same data are not required.
- When a value is not modified, the overhead of data copying is eliminated, resulting in improved performance.
- Copying is performed only when modifications are made, avoiding unnecessary data duplication.

3.1.4 Performance Benefits

The distinction between value and reference types, as well as Swift's usage of CoW, has several implications for performance:

- Value types with CoW can lead to more predictable performance, as the compiler can make better optimization decisions when it knows that data cannot be shared or modified externally. Furthermore, the actual data copying is delayed until it's necessary.
- Copying value types can be faster, as the memory layout is more predictable and can often be associated with better cache utilization.

- Avoiding the overhead of managing memory references for reference types can result in performance improvements, as fewer indirections are needed for memory access.

In summary, the distinction between value and reference types in a language like Swift, combined with the use of Copy-On-Write optimizations, provides developers with fine-grained control over memory handling and potentially leads to better performance when compared to languages that do not offer such a distinction. These aspects make Swift an attractive choice for serverless applications where performance is critical.

3.2 Qualitive Comparison with Go, Java, Javascript, and Python

Is the distinction between value types and reference types achievable in each language? Is any benefit from the distinction possible in each language? In this section, we will compare the distinction between reference and value types in Swift to Go, Python, Node.js (JavaScript), and Java. We will examine whether these languages offer a similar distinction, and if any benefits from the distinction are achievable in each language.

3.2.1 Go

Go supports both reference types and value types. In Go, structs are used as value types, while slices, maps, and channels are some examples of reference types.

Value types in Go: When a value type variable is assigned or passed to a function, a copy of the value is created. Consider the following example:

```
package main

import "fmt"

type Point struct {
        X int
        Y int
}

func main() {
        p1 := Point{1, 2}
```

```
        p2 := p1
        p2.X = 3

        fmt.Println(p1) // Output: {1 2}
        fmt.Println(p2) // Output: {3 2}
}
```

In this example, assigning `p1` to `p2` creates a separate copy of the value, hence changing `p2.X` does not affect `p1`.

Reference types in Go: While Go also has reference types, like slices, they behave differently than reference types in Swift. Here's an example using slices:

```
package main

import "fmt"

func main() {
        s1 := []int{1, 2, 3}
        s2 := s1
        s2[0] = 9

        fmt.Println(s1) // Output: [9 2 3]
        fmt.Println(s2) // Output: [9 2 3]
}
```

In this case, assigning `s1` to `s2` creates a reference to the same slice. Modifying `s2[0]` affects `s1` as well.

Although Go has value and reference types, it does not support Copy-On-Write (CoW) optimizations like Swift, limiting its potential for similar performance improvements.

3.2.2 Python

Python, being a dynamically typed language, does not explicitly provide separate value and reference types. However, the distinction can still be made between mutable and immutable objects, which relates to the concept of value and reference types.

Immutable objects: Immutable objects like tuples, strings, and frozensets behave similarly to value types:

```
t1 = (1, 2)
t2 = t1
t2 += (3,)

print(t1)  # Output: (1, 2)
print(t2)  # Output: (1, 2, 3)
```

In this example, updating `t2` by adding a new element does not affect `t1`.

Mutable objects: Mutable objects like lists, dictionaries, and sets behave more like reference types:

```
l1 = [1, 2, 3]
l2 = l1
l2[0] = 9

print(l1)  # Output: [9, 2, 3]
print(l2)  # Output: [9, 2, 3]
```

Here, assigning `l1` to `l2` creates a reference to the same list. Modifying `l2[0]` affects `l1` as well.

Python's performance characteristics are quite different from languages like Swift and Go. As Python does not have a built-in CoW mechanism and is generally slower due to its dynamic typing and interpreted nature, the benefits achievable from the distinction between value and reference types are comparatively limited.

3.2.3 Node.js (JavaScript)

JavaScript, the language used in Node.js, is also dynamically typed and principally uses reference types. All objects in JavaScript are reference types, including arrays and functions.

Reference types in JavaScript: When an object is assigned or passed to a function, it's the reference that is passed, not the actual data:

```
let arr1 = [1, 2, 3];
let arr2 = arr1;
arr2[0] = 9;

console.log(arr1); // Output: [ 9, 2, 3 ]
console.log(arr2); // Output: [ 9, 2, 3 ]
```

In this example, assigning `arr1` to `arr2` creates a reference to the same array. Modifying `arr2[0]` affects `arr1` as well.

Simulating value types in JavaScript: Although JavaScript does not have native value types, developers can simulate value-like behavior using techniques such as object cloning:

```
function clone(obj) {
    return JSON.parse(JSON.stringify(obj));
}

let obj1 = { x: 1, y: 2 };
let obj2 = clone(obj1);
obj2.x = 3;

console.log(obj1); // Output: { x: 1, y: 2 }
console.log(obj2); // Output: { x: 3, y: 2 }
```

Despite being able to simulate value types, the distinction between value and reference types in JavaScript is not as natural as it is in Swift. Furthermore, JavaScript lacks built-in CoW optimizations, limiting the performance benefits that can be derived from the distinction.

3.2.4 Java

Java, being an object-oriented language, primarily deals with reference types like classes and interfaces. Primitive types in Java, like `int`, `float`, and `boolean`, are value types.

Value types in Java: Java's primitive types are value types:

```
int a = 1;
int b = a;
b = 3;

System.out.println(a); // Output: 1
System.out.println(b); // Output: 3
```

In this case, assigning `a` to `b` creates a separate copy of the value. Changing the value of `b` does not affect `a`.

Reference types in Java: Java's objects behave as reference types:

```
List<Integer> list1 = new ArrayList<>(Arrays.asList(1, 2, 3));
List<Integer> list2 = list1;
list2.set(0, 9);

System.out.println(list1); // Output: [9, 2, 3]
System.out.println(list2); // Output: [9, 2, 3]
```

In this example, assigning `list1` to `list2` creates a reference to the same list. Modifying an element in `list2` affects `list1` as well.

Java has a more explicit distinction between value (primitive types) and reference types (objects). However, since Java does not have CoW optimizations like Swift, the potential performance benefits from such distinctions are more limited in comparison.

3.3 Summary

In summary, all five languages – Swift, Go, Python, JavaScript (Node.js), and Java – offer some degrees of distinction between value and reference types or related concepts (mutable/immutable objects). The distinction is often more explicit in statically-typed languages like Swift, Go, and Java, while dynamically-typed languages like Python and JavaScript handle the distinction via mutable/immutable objects or using programmer-provided methods for simulating value types.

However, the primary differentiator for Swift is its built-in Copy-On-Write (CoW) optimization for value types. This feature allows Swift to offer performance benefits for serverless applications that may not be as easily achievable in other languages under comparison. Performance improvements, both in terms of memory usage and execution time, can be derived from the proper understanding and application of the distinction between value and reference types, as well as making the most of the CoW optimization in Swift.

Chapter 4

Ease of Development

How easy is it to develop serverless functions in Swift compared to other languages? We can evaluate the development experience of Swift by comparing it with other serverless languages.

4.1 Language Simplicity and Syntax

Swift is a modern, expressive, and safe programming language designed for performance and ease of use. Its simplicity and syntax make it a strong candidate for serverless functions. In this section, we will examine how Swift's syntax compares to other languages and how it contributes to a better development experience.

Simplicity in Swift

Swift's syntax aims to be concise and clear, which can lead to shorter and more readable code. For instance, consider the following example of declaring a constant in Swift:

```
let numberOfDays = 7
```

This is similar to Python:

```
number_of_days = 7
```

However, in Java, the declaration is more verbose:

```
final int numberOfDays = 7;
```

Swift's simplicity becomes more evident when dealing with more complex constructs, such as optional values. In Swift, optional values are handled using the ‘¿ and ‘¡ operators, making it easy to declare and unwrap optional values:

```
let optionalValue: Int? = 42
let unwrappedValue: Int = optionalValue!
```

In comparison, Java uses ‘Optional‘ containers, which leads to more verbose code:

```
Optional<Integer> optionalValue = Optional.of(42);
int unwrappedValue = optionalValue.get();
```

Safety Features

Swift's syntax includes features that promote safe programming practices and reduce the likelihood of errors. One such feature is the guard clause. The guard statement allows developers to perform early exits from a function or loop, simplifying the code and making it more readable. With the compiler enforcing early exits, developers are less likely to introduce errors.

Here's an example of a guard clause in Swift:

```
func processInput(_ input: String?) {
    guard let unwrappedInput = input else {
        print("Invalid input")
        return
    }

    // Continue processing with unwrappedInput
}
```

A similar function in Python might use a conditional statement:

```
def process_input(input: Optional[str]):
    if input is None:
        print("Invalid input")
        return

    # Continue processing with input
```

While both versions are readable, the Swift version explicitly enforces the early exit, making it less prone to errors.

Readability

Swift's syntax and features contribute to more readable code. For example, Swift's type inference system allows developers to write cleaner code without explicitly declaring types:

```
let message = "Hello, world!"
```

In contrast, Java requires explicit type declarations:

```
String message = "Hello, world!";
```

Additionally, Swift's support for functional programming constructs, such as map, filter, and reduce, can make code more readable and expressive. Here's an example of filtering and transforming an array in Swift:

```
let numbers = [1, 2, 3, 4, 5]
let evenSquares = numbers.filter { $0 % 2 == 0 }.map { $0 * $0 }
```

A similar operation in Java is more verbose and less expressive:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenSquares = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
```

Copy-on-Write (CoW)

Copy-on-write (CoW) is an optimization technique that Swift uses to minimize memory usage and improve performance when working with value types like arrays, dictionaries, and strings. CoW delays the copying of a value until it is modified, reducing unnecessary copying and memory overhead.

Here's an example of CoW in action:

```
var array1 = [1, 2, 3]
var array2 = array1
array2.append(4)
```

In this example, ‘array1‘ and ‘array2‘ initially share the same underlying storage. When ‘array2.append(4)‘ is called, Swift detects that the storage is shared and creates a separate copy of the storage for ‘array2‘ before appending the value.

A similar example in Java does not benefit from the CoW optimization:

```
List<Integer> list1 = new ArrayList<>(Arrays.asList(1, 2, 3));
List<Integer> list2 = new ArrayList<>(list1);
list2.add(4);
```

In this case, Java creates a new copy of the list immediately, regardless of whether modifications are made, which can lead to higher memory usage and decreased performance.

Extensions

Swift’s extensions allow developers to add new functionality to existing types, such as classes, structures, or protocols, without modifying their original implementation. This feature makes it simple to enhance types in a clean and modular way.

For example, consider adding a method to the ‘Int‘ type that checks if a number is even:

```
extension Int {
    var isEven: Bool {
        return self % 2 == 0
    }
}

let number = 42
print(number.isEven) // Output: true
```

Java does not have an equivalent feature, so a utility method or wrapper class would be needed:

```
public static boolean isEven(int number) {
    return number % 2 == 0;
}
```

```
int number = 42;
System.out.println(isEven(number)); // Output: true
```

Swift's extension feature leads to more elegant and expressive code compared to Java in this example.

Custom Operators

Swift allows developers to define custom operators, providing flexibility and expressiveness in the language. Custom operators can be created for arithmetic, comparison, logical, and other operations, enhancing readability and simplifying complex expressions.

For example, consider defining a custom '**' operator for exponentiation:

```
infix operator **: MultiplicationPrecedence

func **(base: Double, exponent: Double) -> Double {
    return pow(base, exponent)
}

let result = 2.0 ** 3.0 // Output: 8.0
```

In Java, you would need to use the 'Math.pow' function directly, which may be less expressive:

```
double result = Math.pow(2.0, 3.0); // Output: 8.0
```

In summary, Swift's simplicity and syntax, safety features like guard clauses, support for functional programming constructs, copy-on-write optimization, extensions, and custom operators contribute to writing simple, safe, and efficient code. These features enable developers to create high-performance serverless functions while maintaining readability and expressiveness.

Property Wrappers

Property wrappers in Swift are a powerful language feature that can significantly simplify code by encapsulating common patterns and behaviors. They allow developers to create reusable, composable, and declarative abstractions for property access patterns.

One prominent example of property wrappers is SwiftUI, a user interface toolkit for building applications. SwiftUI extensively uses property wrappers, such as '@State', '@Binding', and '@ObservedObject', to manage state and data flow. These wrappers enable developers to create complex and reactive UIs with concise and expressive code.

Another instance where property wrappers are beneficial is the Swift ‘ArgumentParser‘ library. This library provides a straightforward way to parse command-line arguments. By employing property wrappers like ‘@Option‘, ‘@Argument‘, and ‘@Flag‘, developers can define command-line interfaces with minimal code, eliminating the need to manually handle argument parsing. This results in a more readable and maintainable codebase.

In the following Swift ‘ArgumentParser‘ example, a ‘transform‘ argument is used to define a URL option and validate the input:

```
import ArgumentParser

struct ExampleApp: ParsableCommand {
    @Argument(help: "Your name")
    var name: String

    @Option(help: "Your age")
    var age: Int

    @Argument(help: "URL of the PS (powersoft) Server", transform:
     ↪ urlTransformer)
    var psURL: URL

    func run() {
        print("Hello, \(name)! You are \(age) years old.")
        print("PS Server URL: \(psURL)")
    }
}

let urlTransformer: (String) -> URL = { str in
    guard let url = URL(string: str) else {
        fatalError("\(str) is not a valid URL!")
    }
    return url
}

ExampleApp.main()
```

For comparison, let’s look at similar functionality in Python using the ‘argparse‘ library:

```
import argparse
import sys
from urllib.parse import urlparse

def url_transformer(str):
    url = urlparse(str)
    if not url.scheme or not url.netloc:
        sys.exit(f"{str} is not a valid URL!")
    return url

parser = argparse.ArgumentParser(description="Example command-line
 ↪  application")
parser.add_argument("name", help="Your name")
parser.add_argument("age", type=int, help="Your age")
parser.add_argument("psURL", type=url_transformer, help="URL of the
 ↪  PS (powersoft) Server")

args = parser.parse_args()

print(f"Hello, {args.name}! You are {args.age} years old.")
print(f"PS Server URL: {args.psURL}")
```

And in Java using the ‘picocli‘ library:

```
import java.net.MalformedURLException;
import java.net.URL;
import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Option;
import picocli.CommandLine.Parameters;

@Command(name = "ExampleApp", description = "Example command-line
 ↪  application")
public class ExampleApp implements Runnable {
    @Parameters(index = "0", description = "Your name")
    private String name;

    @Option(names = {"-a", "--age"}, description = "Your age")
    private int age;
```

```
    @Parameters(index = "1", description = "URL of the PS (powersoft)
     ↪  Server", converter = UrlConverter.class)
    private URL psURL;

    public static void main(String[] args) {
        int exitCode = new CommandLine(new
         ↪  ExampleApp()).execute(args);
        System.exit(exitCode);
    }

    @Override
    public void run() {
        System.out.printf("Hello, %s! You are %d years old.%n", name,
         ↪  age);
        System.out.printf("PS Server URL: %s%n", psURL);
    }

    public static class UrlConverter implements
     ↪  CommandLine.ITypeConverter<URL> {
        @Override
        public URL convert(String value) throws MalformedURLException
         ↪  {
            URL url = new URL(value);
            if (url.getProtocol() == null || url.getHost() == null) {
                throw new MalformedURLException(value + " is not a
                 ↪  valid URL!");
            }
            return url;
        }
    }
}
```

In the Python and Java examples, the code for validating and parsing the URL input is more verbose compared to the Swift version. The Swift @Argument property wrapper, combined with the transform argument, simplifies the code and enhances expressiveness.

In summary, Swift's property wrappers contribute to the language's simplicity and expressiveness, making it an attractive choice for serverless development. Their ability to

provide powerful abstractions and code simplification is an advantage that might be more difficult or impossible to achieve in other languages.

4.2 Available Libraries and Frameworks

Swift has a growing ecosystem that is constantly evolving and expanding. However, it might not be as extensive as more established languages like Python, Java, and Node.js. This could affect the availability of libraries and frameworks needed for specific use cases in the context of FaaS. In this section, we will discuss some of the notable libraries and frameworks available for Swift, as well as some of the challenges developers may face in certain scenarios. [13]

4.2.1 Web Development

One of the most prominent Swift web frameworks is Vapor, which allows developers to build web applications and APIs using Swift. While Vapor has gained popularity and offers many useful features, it comes with its own set of challenges when compared to popular frameworks or solutions available for Python, Node.js, and Java.

Popular Frameworks for Python, Node.js, and Java

In contrast to Swift's Vapor, more established languages such as Python, Node.js, and Java offer a wide range of popular frameworks and libraries that make web development easier and faster. Some of these frameworks have been around for many years and have extensive documentation, community support, and a large ecosystem of plugins and extensions.

Python

- **Django**: A high-level web framework that promotes rapid development and clean, pragmatic design. It includes an ORM, authentication support, an admin interface, and many other features out-of-the-box [1].
- **Flask**: A lightweight web framework that provides flexibility for developers to choose their own components, such as databases and authentication systems [3].

Node.js

- **Express**: A minimal and flexible web application framework for Node.js that provides a robust set of features for web and mobile applications. Express is widely used and has a large number of plugins and middleware available [2].

- **Koa**: A next-generation web framework for Node.js, created by the team behind Express. Koa is designed to be more expressive and robust while being smaller and more lightweight [6].

Java

- **Spring Boot**: A widely-used framework that simplifies the development and deployment of Java-based web applications. Spring Boot provides built-in support for embedded servers, security, data access, and more [9].
- **JavaServer Faces (JSF)**: A Java web application framework that simplifies building user interfaces for Java EE applications. JSF provides a component-based approach, allowing developers to build UIs by assembling pre-built components [5].

Example Use Cases

1. **User Authentication and Authorization**: When building a web application, it is common to require user authentication and authorization. Django includes built-in support for user authentication, while Express has the popular Passport middleware, and Spring Boot provides Spring Security. On the other hand, Vapor offers authentication support, but it might not be as mature or feature-rich as these other frameworks.
2. **Database Integration**: Many web applications require integration with databases. Django comes with a built-in ORM, while Flask has SQLAlchemy and Spring Boot offers JPA and Hibernate. In the Node.js ecosystem, Sequelize and TypeORM are popular choices for database integration. Vapor has its own ORM called Fluent, but it may lack the maturity, community support, and extensive documentation compared to the other solutions.
3. **Template Engines**: Rendering server-side templates is a common task in web development. Python's Django and Flask both offer built-in template engines, while Node.js's Express supports various templating engines like Pug and EJS. In Java, Thymeleaf is a popular template engine for Spring Boot applications. Vapor supports the Leaf template engine, but it may not have the same level of community support or plugin ecosystem as the alternatives.

In summary, while Swift's Vapor framework offers a powerful solution for web development, it may face challenges in terms of maturity, community support, and available plugins when compared to the popular frameworks and libraries available for Python, Node.js, and Java. Developers should consider the specific requirements of their serverless web applications and weigh the trade-offs between the available options.

Vapor and Swift Package Manager

Due to the way the Swift compiler and the Swift Package Manager currently work, resolving package dependencies can be time-consuming, as it happens sequentially. Even a boilerplate Vapor app requires the compilation of more than 1,500 files. Additionally, the Swift Package Manager has experienced stability issues, such as a bug related to updating dependencies from a specific branch of a Git repository. Although the bug has now been resolved, it demonstrates the growing pains Swift's ecosystem is experiencing.

Documentation and Community Support

As a newer language, Swift's documentation and community support might not be as comprehensive as more established languages. Developers may face challenges in finding appropriate resources or examples when working with specific libraries or frameworks in the context of FaaS. This could lead to slower development and increased reliance on trial and error to find solutions.

Despite these challenges, Swift's ecosystem continues to grow and improve, and many developers are contributing to its progress. With time and continued investment from the community, Swift is likely to become a more mature and stable language for serverless functions.

Example: Swift Package Manager Bug

One example that highlights Swift's infancy and lack of maturity for production use is a bug related to the Swift Package Manager. In the past, the Swift Package Manager had an issue where the package cache would not get invalidated for recent commits when a dependency was set to a branch of a Git repository (e.g., `.package(url: "https://github.com/andreas16700/OTModelSyncer_pub", branch: "main")`). This bug significantly hindered development, as developers were forced to resort to obscure workarounds to manage their dependencies.

In the specific case mentioned, the project was divided into several packages, and changes were committed frequently. Due to this bug, the development process became considerably more challenging. The developer had to run unit tests and use Docker containers to ensure operational consistency with Linux, which resulted in increased development time and effort.

Although the bug has now been fixed, it took more than three years since it first appeared on the Swift forums [14] . This example demonstrates that while Swift is a powerful and promising language, it still faces challenges due to its relative immaturity compared to more established languages.

As the Swift ecosystem continues to grow and mature, it is likely that such issues will become less common. However, developers should be aware of the potential challenges they may face when adopting Swift for serverless functions and be prepared to invest additional time and effort to address them.

4.2.2 Developer community and support

A robust developer community and available resources can contribute to the ease of development. While Swift has a strong community, the serverless aspect of it might not be as well-established as other languages.

4.2.3 Tooling and IDE support

Swift has great support in Xcode, which is the primary development environment for Apple platforms. However, developers who primarily work with other languages may not be familiar with Xcode. There is also support for Swift in other IDEs, such as Visual Studio Code, but the level of support may vary compared to languages with more extensive serverless development history.

4.2.4 Integration with serverless platforms

The ease of integrating Swift with serverless platforms like OpenWhisk, AWS Lambda, or Google Cloud Functions may impact the development experience. Consider the availability of templates, plugins, and other tools that facilitate serverless development and deployment.

4.2.5 Learning curve

The ease of development in a language also depends on an individual developer's familiarity and previous experience with that language. Swift is easy to learn for beginners, but developers who have never used Swift may need some time to become proficient.

4.2.6 Linux Support

Swift's support for Linux is relatively recent, and there are some features that are not available or have limited functionality due to the absence of the Objective-C runtime. Swift was designed to interoperate closely with Objective-C when it is present, but it was also designed to work in environments where the Objective-C runtime does not exist [8] . This means that some features that depend on the Objective-C runtime are not available on Linux.

For example, when a Swift class on Apple's platforms is marked `@objc` or subclasses `NSObject`, you can use the Objective-C runtime to enumerate available methods on an object or call methods using selectors. Such capabilities are absent on Linux because they depend on the Objective-C runtime [8].

Developers might encounter unexpected limitations when developing serverless functions due to these differences. The exact features that are not available on linux are not well documented. While a program may compile and run fine on a Mac, it may either encounter a compiler-time error on linux or, albeit sparingly, crash at runtime. Developers may need to resort to writing unit tests and running them in Docker containers to ensure operational consistency between platforms.

An Example

While developing a crucial serverless function, compiling and deploying it raised no concerns. But trying to invoke it elicited an unexpected error.

```
aloizi04@kube1:~$ python3 invoke.py -p 8083 init syncModel syncModel_action.zip
{"ok":true}

aloizi04@kube1:~$ python3 invoke.py -p 8083 run
{"error":"command exited"}
```

Figure 4.1: Invoking the action using Apache's invoke.py tool

Note: This is using the *invoke tool* [4] provided by OpenWhisk for debugging runtimes.

Every action runs on a language runtime. A runtime is a docker image. Attaching to the docker container and trying to run the binary to get more clues, we can see the following issue:

```
root@d0478e2b14c9:/swiftAction/action/1/bin# ./exec
{"value": {}}
Received: {"value": {}}
Syncing model965
fetching source PS data..
Illegal instruction (core dumped)
root@d0478e2b14c9:/swiftAction/action/1/bin#
```

Figure 4.2: Crash witnessed by manually running the action executable in the runtime container

Due to the lack of Swift Linux IDEs and debuggers, not many options besides using `lldb` exist.

Building the program again in debug mode (for easier debugging) and investigating with `lldb`, an offending function is found and a breakpoint is set.

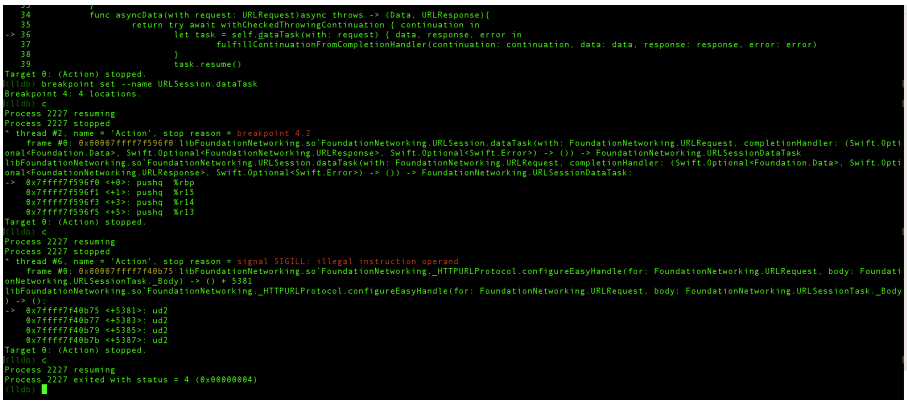


Figure 4.3: Debugging with LLDB to find the cause of the crash

The issue seems to arise after invoking the `dataTask(with:)` function of Foundation's `URLSession`. The networking part of `Foundation` made heavy use of Objective-C and the Swift team, in the transition away from Objective-C, made a separate package called `FoundationNetworking` just for Linux. Many networking parts of `Foundation` are either poorly documented about their Linux support or silently unimplemented. The true cause of the above crash is unknown.

This example demonstrates a significant hurdle in the development process that might arise with no compiler warning, and made exceptionally more difficult with the sheer lack of any sort of feedback.

Debugging serverless applications is already difficult enough, and Swift FaaS developers should expect it to be even more difficult (in the case of Swift) as they may find themselves using `lldb` unexpectedly.

Swift's Linux support continues to evolve, and the community is working on addressing these limitations. However, developers should be aware of the potential challenges they may face when adopting Swift for serverless functions in a Linux environment and be prepared to invest additional time and effort to overcome them.

Chapter 5

Synchronization System Case Study

In eCommerce, synchronization systems are often needed to ensure that the online store reflects the current state of the store's products' logistics. Brick-and-mortar stores manage their inventory with logistical software system. A synchronization system that ensures the consistency and truthfulness of the online store is crucial. The main focus of this chapter is on a practical comparison between serverless and monolithic implementations of such a synchronization system. Synthetic workloads are generated and two mock servers, simulating a logistics system and a Shopify store are used.

5.1 System Overview

Suppose a brick-and-mortar store sells products. Each product has different variants. The store keeps track of its inventory with a logistics system. That system provides an API endpoint with which one can query data and modify it. The store wishes to have an online eCommerce store. The platform of the store has an API endpoint for adding and modifying resources, such as the products. A system is needed that ensures that the online store reflects the current state of the physical store. That means that Every product that exists on the physical store, should exist on the online store with the correct quantity. If a product sells out on the physical store, it is the job of the system to ensure that the lack of stock is present on the online store.

5.1.1 Main Components

PS will denote the server that runs the physical store's entrypoint. SH will denote the server that runs the online store's endpoint. Syncer is the object that synchronizes a product between the two endpoints.

Chapter 6

Results

6.1 Experimental Setup

The experimental setup involved deploying the monolithic and serverless implementations on machines of type m510 on CloudLab in the Utah cluster. Benchmarks were conducted using increasing workloads to assess the performance of each system. A custom-built rate limiter was used to control the flow of requests to the systems.

6.2 Benchmarking Process

This section describes the workload and the main components involved in the benchmarking process. It also explains the methodology used in benchmarking the serverless and monolithic implementations.

6.2.1 Main Components

PS Server

The PS Server contains items and their stock count as shown in the following structures:

```
struct PSItem: Identifiable {
    ...
    let itemCode: String
    let modelCode: String
    var id: String { itemCode }
    ...
}

struct PSStock: Identifiable {
```

```
    let itemCode: String
    var id: String { itemCode }
    var stock: Int
}
```

Many *Items* belong to one *Model*. A `PSItem` is identified by its `itemCode`, and a model is identified by a `modelCode`.

SH Server

The SH Server contains products and their stock count as shown in the following structures:

```
struct InventoryLevel: Identifiable {
    let inventoryItemID: Int
    var id: Int { inventoryItemID }
    var stock: Int
}

struct SHVariant: Identifiable {
    let id: Int
    let productID: Int
    let sku: String
    let inventoryItemID: Int
}

struct SHProduct: Identifiable {
    let id: Int
    let handle: String
    ...
    let variants: [SHVariant]
}
```

ModelSyncer

A *Model* from the PS Server corresponds to an `SHProduct` from the SH Server. A *Model*'s `PSItems` correspond to the respective `SHProduct`'s `SHVariant`. A `PSItem`'s stock (`PSStock`) corresponds to the respective `SHVariant`'s `InventoryLevel`. The goal of the `ModelSyncer` is to ensure that every model from the PS Server is reflected by an equivalent product on the SH Server (including stock listings).

6.2.2 Workload

A workload can be characterized by two variables:

1. Total number of models: The total number of models that should be generated on the PS Server. For each model, a number of items are generated ranging from 1 to 20.

2. Seed: Two numbers (xSeed and ySeed) that will be used for randomization. A pseudo-random number generator (PRNG) will be seeded with these numbers and will be used for all functions concerning randomization. This ensures that given these two variables the generated resources on the two servers will always be the same. Any reference to randomness, from now on, refers to pseudo-randomness provided by the PRNG.

6.2.3 Benchmarker

The Benchmarker does three main things:

1. Setup the two servers

2. Sync all models

3. Return metrics concerning step 2. Metrics include, among other things, the time it took, memory and space usage.

Setting up the servers

The Benchmarker will be referred to as ‘B‘ to avoid repetition.

Generating models on PS Server ‘B‘ sends a GET request to PS Server at the path ‘/generate/modelCount/xSeed/ySeed‘. The PS Server generates `modelCount` models using the `(xSeed, ySeed)` seed.

Generating equivalent resource on SH Server For each model from the PS Server, the equivalent resource on the SH Server can be in one of three states:

1. Equivalent resources exist (SHProduct, SHVariant, InventoryListing) and are up to date.

2. Equivalent resources exist partially (SHProduct, SHVariants, InventoryLevels) and are not up to date. Partially exist means some items might not have an equivalent SHVariant on the SH Server. "Not up to date" means some properties of the resource on the SH Server might need updating (source of truth is always PS Server).

3. No equivalent resources exist for that model. That model does not exist on the SH Server. Its equivalent resources (SHProduct, SHVariants, InventoryLevels) have to be created from scratch.

SH Server has initially no data. **N** = total number of models generated on PS Server. P = number of models partially synced on SH Server (state 2). F = number of models fully synced on SH Server (state 3).

My apologies for the abrupt cut-off. Here is the continuation of the Benchmarking Process section:

‘B‘ randomly generates the numbers P and F given the following constraints: * $\mathbf{N} > 0$ * P+$F < \mathbf{N} * 0 \leq P \leq \mathbf{N} * 0 \leq F < \mathbf{N}$

The sum of the numbers of partially and fully synced models should not exceed the total number of models. All models could be partially synced, but all models should not be fully synced as there would be no work to be done.

‘B‘ requests P+F random models from the PS Server. Then, ‘B‘ constructs the partially and fully synced products from the P and F models, respectively, along with their respective stocks. The RNG is used to generate random data such as IDs and stock counts. ‘B‘ finally sends this data to the SH Server on the route ‘batchProductsAndStocks‘.

At this point, the two servers are set up and ready to run any benchmarks to sync any models.

Running a workload

The process of running a workload is very similar between the serverless and the monolithic implementations. The only way they differ is in the ‘syncModel‘ function. To encapsulate this behavior and simplify the benchmarking, the following protocol is defined:

```
protocol WorkloadRunner {
    init(psURL: URL, shURL: URL, msDelay: Int?)
    static var name: String { get }
    var psClient: MockPsClient { get }
    var shClient: MockShClient { get }
    func runSync(sourceData: SourceData) async throws -> (Int, Int,
     ↪ [UInt64])
}
```

Any type that fulfills these requirements can then be used with the Benchmarker and get results.

The ‘runSync‘ function takes some source data (the current data on the two servers) and returns S, F, L where S is the number of successful syncs, F the number of failed syncs, and L is an array that contains the latency observed for each sync (the amount of time it took for a single model sync to complete).

This allows the Benchmarker to easily get comparative results for a range of workload sizes for each ‘RunnerType‘. For instance, for a given workload, it can easily set up the servers and run an implementation (make it sync every model):

```
static func runOnce(workload: Workload, runner: WorkloadRunner) async
 ↪  throws -> (String, Duration, Int, Int, [UInt64]) {
    let name = type(of: runner).name
    print("Setting up servers for ", name)
    await setUpServers(for: workload)
    print("Retrieving source data...")
    let source = await getSourceData()
    print("Running..")
    var (successes, fails, durations) = (0, 0,
     ↪  [UInt64].init(repeating: 0, count:
     ↪  source.psModelsByModelCode.count))
    let duration = try await SuspendingClock().measure {
        (successes, fails, durations) = try await
         ↪  runner.runSync(sourceData: source)
    }
    print("\(name) took \(duration). Had \(fails) fails and
     ↪  \(successes) successes.")

    print("Resetting servers' state..")
    let _ = await runner.psClient.reset()
    let _ = await runner.shClient.reset()
    print("Reset both servers.")

    return (name, duration, successes, fails, durations)
}
```

The above function is for running one workload on one runner. We can leverage this to easily run the benchmark for every ‘RunnerType‘:

```
func runAll(minModelCount: Int, totalModelCount, increments: Int) {
    let workloadSizes = stride(from: minModelCount, to:
     ↪  totalModelCount + 1, by: increments)
    let resultsFileName =
     ↪  "bench_\(workload.totalModelCount)_\(runners.map { type(of:
     ↪  $0).name }.joined(separator:
     ↪  ","))_\(workload.xSeed)_\(workload.ySeed)"
```

```
    let writer = initializeCSV(name: resultsFileName)
    for modelsCount in workloadSizes {
        var times: [Duration] = .init()
        var successes: [Int] = .init()
        var fails: [Int] = .init()
        var latencies: [Stats] = .init()
        let (name, time, succ, fail, lats) = try await
        ↪ Self.runOnce(workload: workload, runner: runner)
        print(name, "took \(time)")
        times.append(time)
        successes.append(succ)
        fails.append(fail)
        latencies.append(.init(from: lats)!)
        addResults(writer: writer, modelsCount: modelsCount, times:
        ↪ times, succ: successes, fail: fails, latencyStats:
        ↪ latencies)
    }
}
```

The above function ensures that for every workload size, every ‘RunnerType‘ gets to do the exact same type of work as the environment is exactly the same (thanks to using the same RNG seeds). The results are written to a CSV file. The technicalities of the CSV writing are beyond the scope of this chapter.

6.3 Results and Discussion

In a given workload, sending the model sync request raised the following question: how often should the model sync requests be sent? As the workload increases, should any adjustments be made? As it currently stands, the benchmarker uses Swift’s TaskGroup construct to send the requests.

6.3.1 Without a Rate Limit

First, we experimented with leaving it all up to the task group, which resulted in requests being sent roughly all at once. The serverless implementation’s execution time increased incomparably to the monolithic version, as seen on figure 5.1. While for both implementations the median latency did tend to increase, the serverless implementation’s latency increased considerably faster, as seen in figures 5.2 and 5.3.

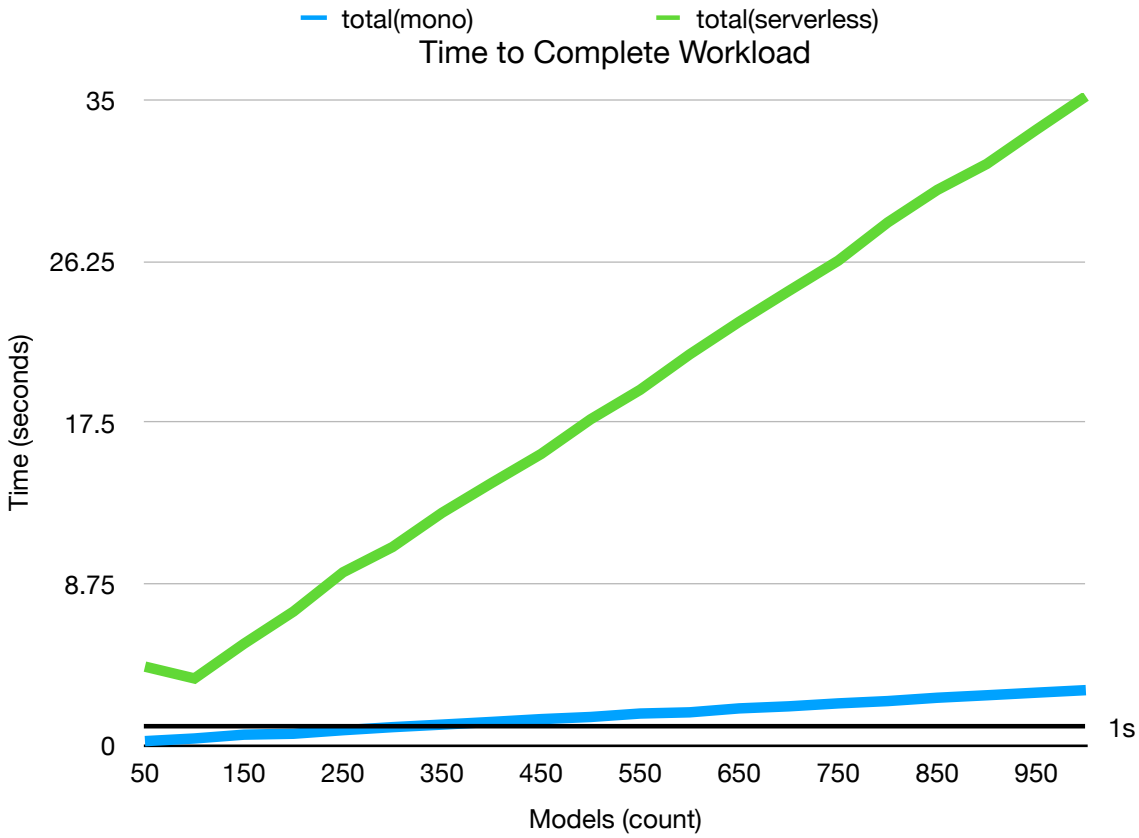


Figure 6.1: Execution time comparison between serverless and monolithic implementations

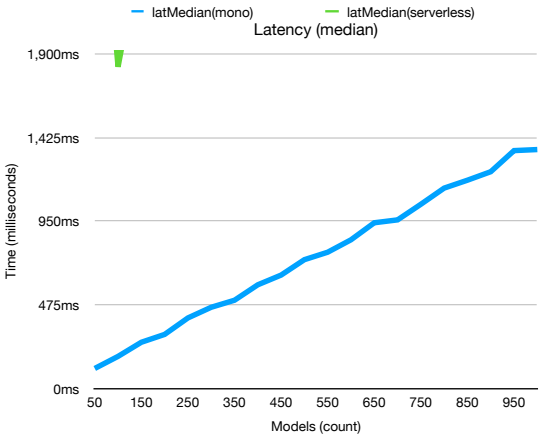


Figure 6.2: Median latency for a single model request, scaled for the monolithic implementation

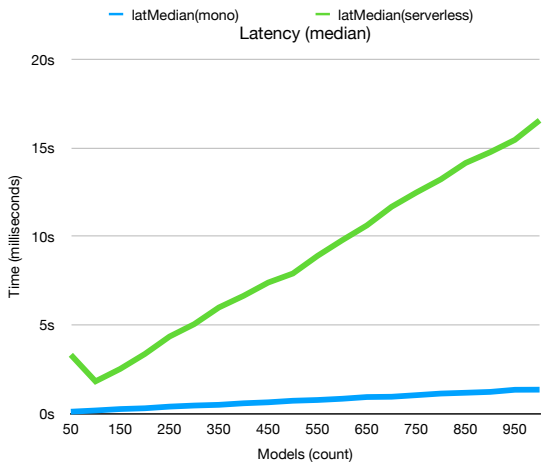


Figure 6.3: Median latency for a single model request

The above is explained by a much lower Queries per Second (QPS) by the serverless implementation. As seen on figure 5.4, while the serverless implementation seems to have

hit a limit of 28 QPS, the serverless implementation seems to hit a limit of 333, more than ten times bigger.

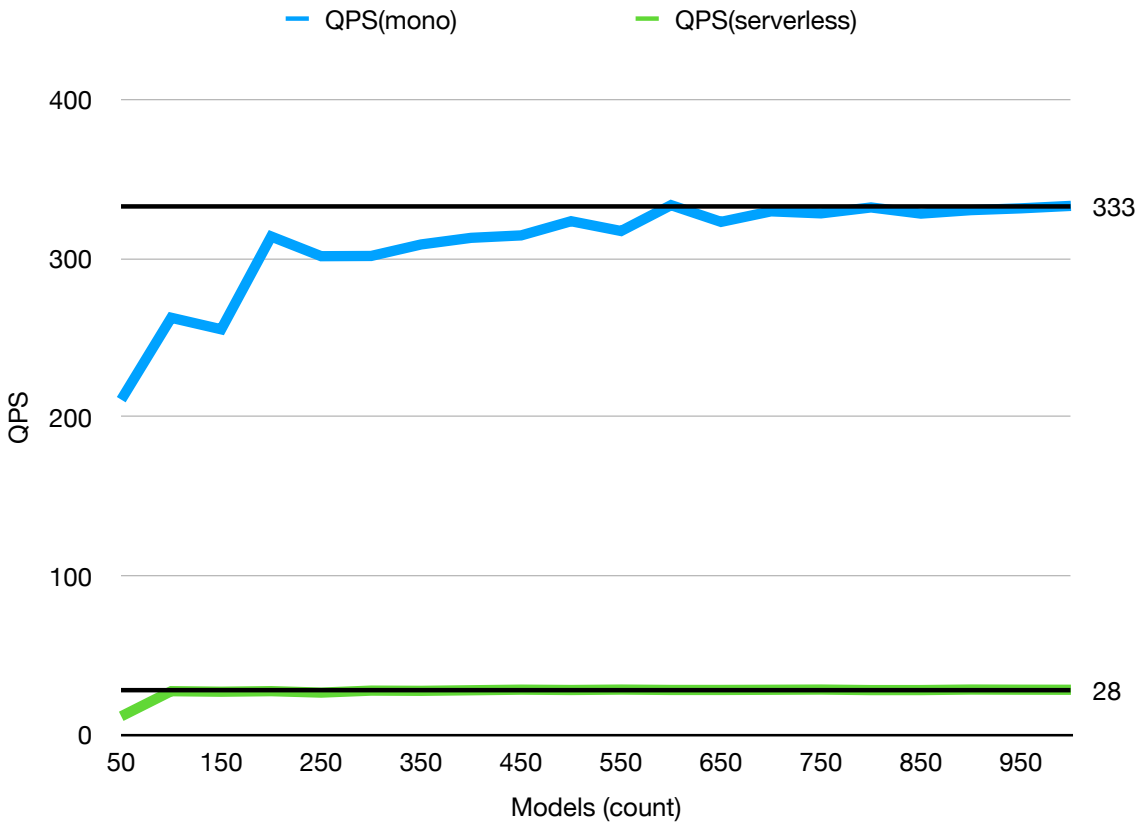


Figure 6.4: QPS comparison between serverless and monolithic implementations

6.3.2 Implementing a Rate Limit

We use the Swift Rate Limitter, which is a custom made Swift rate limitter we have developed [15], which is well tested, which allows us to send requests with a minimum set delay between them. We utilized it to send the requests every 100ms. This will cap the maximum QPS to 10 which ensures neither implementation will be stressed, hopefully allowing us to have a clearer picture of the situtation.

With the rate limit both implementations take the same amount of time and bot achieve roughly 10QPS. As evident from figure 5.5, a single model sync takes roughly 280ms and 60ms on the serverless and monolithic implementations respectively. An interesting finding, is that the P99.9% latency (figures 5.6 and 5.7), apart from being higher, it as less stable on the serverless implementation. This is despite utilizing less than the third of its maximum QPS.

Despite utilizing 18 machines for executing actions, the serverless approach did not

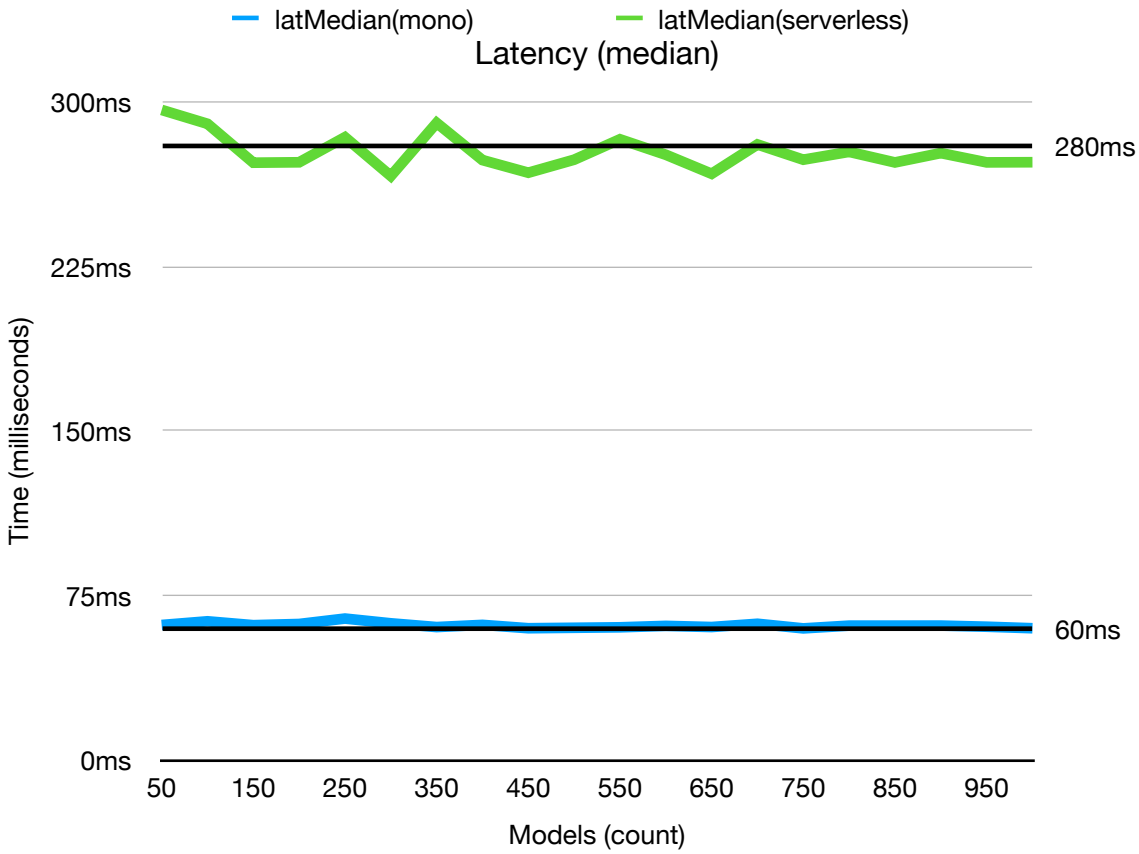


Figure 6.5: Rate-limited (100ms) comparison between serverless and monolithic implementations

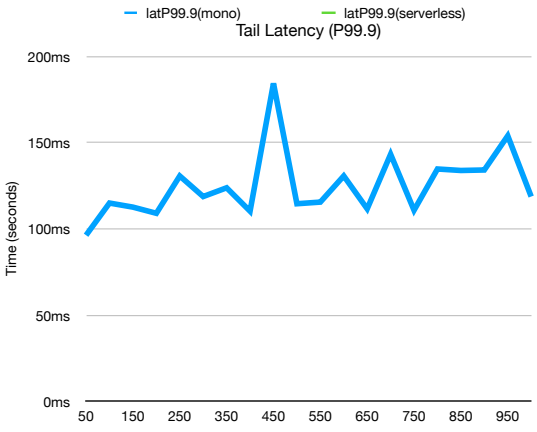


Figure 6.6: Median latency for a single model request, scaled for the monolithic implementation

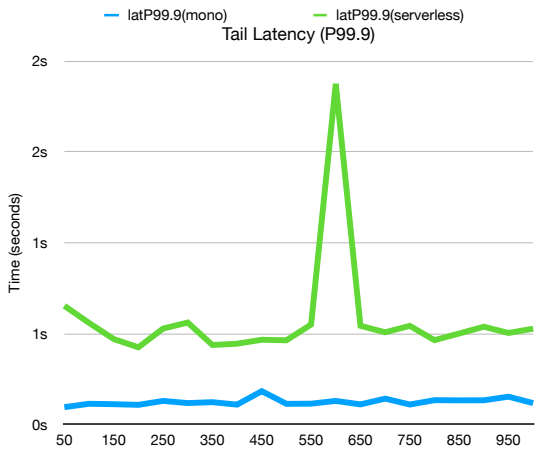


Figure 6.7: Median latency for a single model request

show any clear benefits. This may be attributed to the lack of support for "intra-concurrency" in OpenWhisk runtimes, excluding NodeJS. This limitation significantly affects the scal-

ing capabilities of the serverless implementation, which is a crucial aspect of the serverless or FaaS promise. 18 machines achieving 28QPS means that on average, a request takes 642ms to complete, which is more than double of the 280ms it takes, when the system is not stressed.

6.4 Improvements and Future Work

There are potential avenues for improvement in the serverless system. Notably, updating the Go proxy used by all OpenWhisk runtimes to support intra-concurrency could allow all language runtimes, including Swift, to support concurrent executions, potentially dramatically increasing the maximum QPS.

6.5 Conclusion

The case study findings contribute to the overall conclusion of this thesis, suggesting that the benefit of migrating to a serverless implementation is not always evident and should be carefully assessed for each workflow. The case study also highlights the importance of runtime support for intra-concurrency in realizing the full potential of serverless systems. Without intra-concurrency, the requests stay in memory until a container is available to serve them. The overheads of OpenWhisk may explain the 2 times slowdown of its request completion under load. Notably, the Swift runtime was updated to the latest version to leverage its native async/await features, which played a significant role in the serverless implementation. A total of eighteen identical machines achieving a throughput of ten times less than one single, identical machine is frankly unacceptable. Support for intra-concurrency for the ActionLoop proxy should be a priority for the OpenWhisk project as seamless, efficient scaling is the central promise of FaaS.

Chapter 7

Conclusion

In concluding, it is evident that Swift holds substantial potential as a serverless language, albeit with several areas necessitating further development and exploration.

Swift's expressiveness and simplicity render it a productive choice for developers, offering a unique blend of performance and usability. However, a more comprehensive practical comparison with other serverless languages such as Python, JavaScript, or Go is imperative to fully comprehend its strengths and weaknesses in realistic serverless scenarios. This will not only validate Swift's theoretical advantages but also provide valuable insights for its future development.

One important aspect to consider, is that the decision to utilize serverless architecture should not be made lightly. While serverless computing offers scalability and reduces the need for server management, in some cases, a shift from a serverless to a monolithic architecture may lead to improved scalability, resilience, and cost-effectiveness, as was the case with the Prime Video service [17]. This underscores the significance of a case-by-case analysis when deciding on the architectural choice, which is a subject of ongoing debate in the tech industry [16].

The current state of Swift as a serverless language is promising, but there are limitations that need to be addressed. One of the key challenges is the lack of helpful feedback for some errors, which contradicts Swift's core promise of safety. Developers often find themselves needing to use `lldb` on remote machines to troubleshoot, which can be a significant hurdle in the development process.

Moreover, the need for better documentation and Linux support is evident. The absence of a built-in mechanism to ensure that all APIs used are supported on the Linux platform is a significant drawback. Developers are currently left to rely on Docker tests or discover at runtime if a feature is unsupported. Enhancing Linux support and providing comprehensive documentation would greatly improve Swift's fitness as a systems language and its viability as a serverless language.

In the context of OpenWhisk's ActionLoop proxy, the importance of intra-concurrency

cannot be overstated. The case study results indicate that the lack of real concurrency wastes much of the hardware's potential and provides poor scaling, which is contrary to the essence of serverless. As most runtimes use the ActionLoop proxy, its support of concurrency is critically important for the efficient use of Swift in serverless settings.

Looking ahead, the future of Swift as a serverless language is bright, but it hinges on addressing these challenges and capitalizing on its strengths. The journey of Swift in the serverless landscape is just beginning, and with the right improvements, it can become a powerful tool in the hands of developers. In this journey, it is vital to remember that while serverless architecture offers many benefits, the decision between serverless and monolithic architectures should be made with careful consideration ofthe specific context and requirements of the project, as well illustrated by the case of Amazon's Prime Video service

Bibliography

[1] Django. https://www.djangoproject.com/.

[2] Express. https://expressjs.com/.

[3] Flask. https://flask.palletsprojects.com/.

[4] invoke tool. https://github.com/apache/openwhisk/blob/master/tools/actionProxy/invoke.py.

[5] JavaServer Faces. https://www.oracle.com/java/technologies/javaserverfaces.html.

[6] Koa. https://koajs.com/.

[7] Platform Support. https://github.com/apache/openwhisk-runtime-swift.

[8] Platform Support. https://swift.org/about/#platform-support.

[9] Spring Boot. https://spring.io/projects/spring-boot.

[10] Apache OpenWhisk - Action Interface. https://github.com/apache/openwhisk/blob/master/docs/actions-new.md#action-interface, 2023.

[11] Apache OpenWhisk - ActionLoop Proxy. https://github.com/apache/openwhisk/blob/master/docs/actions-actionloop.md, 2023.

[12] Apache OpenWhisk. Apache openwhisk documentation. https://openwhisk.apache.org/documentation.html, 2023.

[13] Chromium OS. Sandbox. https://www.chromium.org/developers/design-documents/sandbox.

[14] S. Forums. Dependency on (very) active branch doesn't pull new commits. https://forums.swift.org/t/dependency-on-very-active-branch-doesnt-pull-new-commits/33204, 2020.

[15] A. Loizides. Rate Limitter for Swift. https://github.com/andreas16700/RateLimitingCommunicator, 2022.

[16] V. Review. Serverless vs monolith, 2023. [Online; accessed May-2023].

[17] P. V. Tech. Scaling up the prime video audio/video monitoring service and reducing costs by 90 [Online; accessed May-2023].