Delft University of Technology

The Monadic Approach to Serverless Applications

Master Thesis

Distributed Systems

CSE5000

Authors:
Rahul Kochar (4812522)

Thesis Committee:
Prof. dr. J. Rellermeyer (Daily supervisor)
Dr. ir. J. Pouwelse
Dr. Przemysław Pawełczak

April 12, 2024

Abstract

The serverless computing trend is steadily picking up steam over the last few years and is challenging the traditional microservices on Kubernetes model which included inefficiencies like idling. The big three cloud providers AWS, GCP and Azure have different opinions on what serverless computing and serverless applications should look like, how they should be designed and what their architecture should be leading to each cloud going their own separate way. Ultimately, this causes vendor lock-in, fragments the serverless cloud technology landscape, paralyzes other research and development efforts while also harming other stakeholders like users, developers and businesses that depend on these cloud services and products. The heavy reliance on often changing proprietary API that is not compatible with alternatives makes stability a serious concern and enables exploitation of users. This thesis proposes MSA - Monadic approach to Serverless Applications to enables developers to write cloud agnostic serverless applications and tackle vendor lock-in. Users can build complex serverless applications without polluting business logic and migrate from one cloud to another by pressing a button. This thesis makes multiple other contributions such as documenting and showing how to reconcile differences in features between the big clouds in various ways for telemetry, metrics, unifying interactions of equivalent services in different clouds, getting rid of tedious boilerplate code in business logic while remaining cloud agnostic among others. Three serverless applications are built with MSA and benchmarked against equivalent applications build with alternatives. Serverless applications built with MSA frequently outperformed in execution time and RAM usage. These experiments also yielded insights on situations in which certain types of serverless applications perform better on a cloud than others. Lastly, usability of MSA and the quality of solution (solving vendor lock-in) received good scores in a user survey.

Contents

1 Introduction

Serverless computing is a paradigm of writing code in which cloud providers like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure offer managed services so that developers focus solely on writing business logic while the cloud provider takes responsibility for hosting the application with high availability, managing and administering all related physical and virtual infrastructure, security, and providing near-instantaneous scaling to meet demand, cost-effectively [143], [141], [169].

Serverless functions or Function-as-a-Service are a small piece of code hosted somewhere on the Internet that can be executed by making a curl request to an endpoint. These functions are designed to be small, disposable, and cheap such that they can be created en-masse rapidly and after completing their duty, can be discarded just as fast. This inherently requires serverless functions to be stateless and in turn significantly lowers the threshold for writing highly scalable APIs, stateless applications and small pieces of code while taking away granular control from developers [137], [98]. The cloud provider abstracts away almost all of the control into a service where developers trade away customisability and configurability for convenience and economics of scale to collectively reduce costs (for users of serverless and public cloud providers). Traditionally, developers had to maintain their own servers and other infrastructure (networking, security, cooling, etc.) which gave developers the freedom and ability to customise the minutest details about where and how their code runs.

AWS Lambda is the first serverless product offering by a large public cloud provider in 2014 [65]. Other public cloud providers and vendors followed soon after with their own offerings with various interpretations of serverless that fragment the serverless technology landscape and ultimately hinders research and development of cloud technologies. [45], [156], [174], [113], [16] describe that vendor lock in characterized by lack of common tooling, compatibility across ecosystems and vendor specific configurations are a serious concern for users of cloud technologies, tools and services. The multi-cloud strategy is frequently employed by consumers to escape from the clutches of vendor lock-in, but it runs into the problem of lack of a common framework or protocol for serverless functions and applications. Developers are forced to write code specific to the serverless function and the cloud provider that hosts the serverless function and other infrastructure such as databases, etc. Migrating the function to a different cloud provider or setup can mean a severe re-write of the code [124] - even though serverless is un-

derstood to be as something that can run any code and is therefore agnostic of everything i.e. it is just business logic which in theory stays the same in every context and situation in which the code may be hosted and run on.

To free developers from the shackles and constraints of vendor lock-in described above, this thesis takes inspiration from the monadic approach [4] of abstractions to abstract away and encapsulating the differences, and explicitly stating conditions and requirements that a monad must satisfy so that a compiler (or another program) can verify and give a guarantee that the specified conditions hold and requirements are indeed satisfied.

1.1. Research Questions

This thesis has the following Research Questions (RQ) that satisfy many of the criteria in the list of conditions for determining a grand challenge [77]. Vendor lock-in is a serious problem recognized by practitioners and researchers alike in cloud computing but also widely in software development [52], [117], [147], [91], [176].

1. **Monadic Approach:** *Can monads help the user explicitly declare requirements for their serverless functions without polluting/compromising functional nature of code?*

   Can developers neatly state where the serverless function is hosted and executed (being mindful of where other resources the serverless function interacts with are hosted) and other configurations like telemetry without bloating the code base or other ill effects that tend to solidify vendor lock-in.

2. **Cloud Agnostic:** *Can the monadic approach be applied to Infrastructure as Code to provision cloud agnostic infrastructure*

   Will the monadic approach enable developers to explicitly describe the cloud components and interactions in an arbitrary cloud environment without polluting the functional nature of code while being cloud agnostic at the same time.

3. **Clean Packaging:** *Can code synthesis be a standardized solution for writing generic cloud-agnostic business logic (that can be run in the serverless function of choice)?*

   Will the monadic approach enable serverless functions to have the required capabilities such that developers can write cloud-agnostic business logic and simultaneously the serverless function is not bloated with unrelated packages and unnecessary capabilities that weigh down the serverless function.

4. **How do serverless applications built with a monadic and cloud agnostic approach compare to serverless applications built with the traditional approach?**

5. **Is MSA easy to use?**

1.2. Thesis Structure

Chapter 2 and 3 will provide additional background leading to the main matter in chapters 4 and 5 followed by an Evaluation of the monadic approach and User Survey in Chapters 6 and 7. The thesis concludes with a Discussion, Conclusion and Future Work in chapter 8.

2

Background

This chapter gives the reader an overview of the current state of serverless computing, a closer look at the challenges in cross-cloud compatibility of serverless applications and describes the appealing properties of monads that can solve the aforementioned challenges. Other concepts needed for the solution like Infrastructure as Code are also introduced for the reader's benefit.

2.1. Serverless Compute

Serverless technologies are gaining momentum, with all major cloud providers announcing investments which suggests that serverless will play a critical role in the future of cloud computing. Serverless is attractive because of its fine-grained billing and low maintenance required. Developers pay only for resources actually used and do not pay for idling which is hard to escape in traditional forms of compute. Serverless products and tools like Serverless Functions (AWS Lambda, GCP Cloud Function, etc.) are clean and convenient to use because they sit very high on the stack for instance AWS Lambda sits on top of over 80 other AWS services [19]. All the details of the stack under the serverless function are hidden and managed by the cloud, i.e. this is the cost of convenience as developers have no control, visibility, or knowledge about what is happening inside the function. Developers simply submit code to the cloud provider, and the cloud provider runs the code in a Serverless Function while giving benefits like security, availability, and dynamic elasticity for free. Compared to traditional means, developers give up control in serverless functions in numerous ways [137], [74].

2.1.1. Loss of Control

- **Configuration:** Developers are limited to the configurations provided by the cloud provider. This means developers are limited to supported languages and specific supported versions [94], [139], [30] by the cloud provider. Developers are forced to upgrade the language/version of the language to a supported language/version if the cloud provider decides to halt support. For example, AWS supports four versions of Python at a time [95] and Python has an annual release cycle of often breaking changes [154]. This means that a serverless application written in Python on AWS must update their code every few years (not including maintenance required for other packages and dependencies of the business logic in

that serverless function).

Additionally, there are no configuration options/control over compilers and runtimes like JVM on AWS, GCP and Azure, nor any run-time parameters of Operating System that run the serverless function. The inability to configure JVM prohibits the attachment of agents and limits the monitoring of Java serverless functions [41]. AWS allows monitoring garbage collection with all options can that can be passed to `JAVA_TOOL_OPTIONS`, CPU, memory, etc. [59] while GCP and Azure allow developers to view some metrics like CPU, memory, and sampling requests [87], [85].

To try to solve these problems, cloud providers allow submitting a container (eg: Docker) which is run as a serverless function. Code can be run in any language and version as long as the developers can build a container [53], [172], [48], [46]. This can be a temporary workaround for the shortcomings of serverless functions, but it is inconsistent with the idea of serverless functions. Container-as-a-Service is not serverless because cloud providers will then play the role of providing Infrastructure-as-a-Service (IaaS) to developers i.e. the cloud will host the container and provide some benefits (of IaaS) but developers are responsible for critical aspects like maintaining and patching the operating system in the container, networking/security to facilitate communication in and out of the container.

Serverless functions on different cloud providers have different properties; for instance, an AWS Lambda and Azure Function Application can run for at most 15 and 10 minutes, respectively, while GCP Cloud Functions timeout at 10 minutes (event-driven) or 60 minutes (http functions) [39], [66], [27]. Similarly, other characteristics such as RAM, system architecture, operating system, importing libraries for business logic, size of code in the serverless function, etc. have seemingly arbitrary and unique values on AWS, GCP, and Azure. Factors like GCP Cloud Functions running for 4 and 6 times the max timeout of AWS Lambda and Azure Application Functions [28] respectively or that AWS Lambda supports arm64 architecture while GCP and Azure do not or that Azure Application Functions allow running a serverless function on a Windows Operating System but AWS and GCP do not support Windows out of the box among numerous others deliver a fatal blow to the general case for cross-cloud compatibility of serverless functions and applications [40], [37], [105].

- **Performance:** Program execution of serverless functions is a black box. Physical hardware, core operating system and runtimes are unknown and significantly limit the ability to optimise software for peak performance for that stack of hardware and software. Serverless functions running on virtual hardware are even harder to optimise code for because predicting behaviors is hard. Inconsistencies/variance in performance characteristics can be observed because the cloud provider is responsible for selecting the type of hardware the serverless function runs on, and developers have negligible to no control over this.

  Traditionally, latency can be improved by placing the two components that need low latency nearby in the server rack. Serverless technologies do not have such a concept because the cloud provider schedules the hardware that runs the function. The hardware may be on the same rack but could also be at opposite ends of the data center building. Data center networks are hierarchical because of their vast size forcing non-co-located servers to use layer 3 network to communicate. There is a significant difference in latency and bandwidth between co-located servers in the same rack compared to two servers placed in arbitrary

places in the data center [75]. Cold starts can also affect latency, but cloud providers now offer some keep-warm mechanism at a low cost to mitigate this.

- **Concurrency and Scalability:** Serverless boldly claims as much as you want on demand elasticity. However, cloud providers place limits on maximum resources that can run at any given moment eg: 1000 AWS Lambda and GCP Cloud Functions by default. On request, AWS may increase the number up to 50,000 Lambdas which is significantly more than traditional means of compute given the price. Different cloud providers have different limits and constraints on various resources.

- **Logging and Monitoring:** Monitoring and logging tools are unique to the cloud provider. All of the basic necessities and more are easily available on AWS, GCP and Azure (more in Section 2.1.2). However, these are not compatible with the other cloud platforms and require developers to re-create monitoring and logging on another cloud should developers choose to run a serverless function on another cloud. Developers can choose to use third party tools like DataDog but that introduces extra complexity in managing the logging service compared to using the default out-of-the-box logging tools provided by cloud providers because they do not require management.

- **Security:** Developers are responsible for ”Security in the cloud” i.e. securing the code in a serverless function, networking/firewall, Role Based Access Control (RBAC), etc. while cloud providers usually take responsibility for ”Security of the cloud” which is securing the physical hardware, patching the Operating System etc. [106]. Developers are restricted to the security tools offered by the cloud provider and are suggested to follow the best practices recommended by the cloud provider. These practices, security tools and sometimes principles are often unique to the cloud provider and not transferable to other cloud providers and services [97]. It may not be possible to implement custom security policies and protocols because cloud providers do not grant control/access to stack/underlying tools and technologies of the serverless function. Therefore, it is not guaranteed that the security and logging/monitoring requirements required by the developer/application will always be met. Cloud providers conveniently solve various security challenges for developers using serverless functions but [36], [78] show the emergence of new attack vectors for serverless architecture and serverless code.

- **Debugging/Testing on Local Machine:** Testing serverless functions locally is always a challenge because functions often need to interact with a SQL database or message queue which does not exist locally. However, cloud providers have made efforts to enable developers to locally test and debug code that does not require other components in the cloud. Azure supports invoking a serverless function on command line or with a VS Code extension. AWS [99] and GCP [80] have developed alternatives in the last year. AWS uses SAM [142] on the command line to simulate the function, while Google's `functions_framework` allows running a serverless function like a normal program. Google goes a step further in allowing the code to access components on the cloud with appropriate client libraries. The three major cloud providers have their own unique local testing/debugging ecosystem and are not compatible with other alternatives. The same holds for testing/debugging on the cloud - the available tools and methods are specific to and restricted to what is allowed by the cloud provider.

- **Testing on the Cloud:** An individual serverless function can be easily tested on the cloud by following the instructions of the respective cloud provider. However, none of the cloud providers support tests that have a greater scope than unit tests, i.e. it is not possible to do end-to-end testing or integration/regression tests etc. To avoid this, developers create a replica environment that is used only for testing. It would be beneficial if cloud providers had a ”test” mode granting developers with greater insight for easier debugging than a regular cloud environment.

- **Stateless vs Stateful Applications:** Serverless functions are by design stateless and (semi)-functional programming allows developers to take full advantage of parallel processing on serverless functions. Some applications like MapReduce would benefit greatly from serverless functions but require a state. To get around this, developers often dump the state into a database or cheap storage/cache, and then the following serverless functions contain some logic to figure out which data from the dump/cache it needs and loads it. This is an unclean approach because the heavy lifting of managing state is done programatically by the developer in the serverless function and there are extra dependencies (and costs) to the database which are restricted by latency of input/output speeds and geographies/localities. This approach is loaded with naked side effects and is not functional/pure in nature. [76] introduces disorderly programming where procedural programming does not contribute to scaleable business logic in serverless functions.

- **Issue Resolution:** The cloud provider takes complete ownership for many things as defined in the SLA - Service Level Agreement. Developers are limited to the legalities and fine-prints, possibly without a course of remediation except opening a support ticket. Cloud providers usually have a paid tier system with guaranteed support within a certain number of hours depending on tier. On the lowest/free tier, cloud providers may actively respond and assist developers but may also choose to blatantly ignore the support ticket or feature requests. Developers can control little more than the code running in the serverless function.

While serverless functions pamper developers with conveniences of scalability and cost-efficiency [55], it comes with drawbacks. The fundamental problem with serverless functions is that developers often have other requirements like monitoring, configuration, and security, to name a few, that serverless functions struggle to fully satisfy. It is for the developer to realize which of the developer’s requirements are met, and it is the responsibility of the developer to find a workaround to meet unmet requirements.

The next section will show through more examples that serverless functions by various cloud providers and the ecosystem around the serverless functions have sharp contrasts. Intuitively, the business logic inside a serverless functions should be the same if the same programming language is used - after all that is what serverless functions are all about. Developers give code to the cloud provider and the cloud provider hosts the code. However, a serverless function by itself is rather static and is not very useful. To be useful, the serverless function needs to interact with other cloud components and resources, which are unique to every cloud. The next section describes the major differences in writing code for serverless functions on AWS, GCP and Azure in Python. Similar analogies hold for other languages. In this thesis, we propose MSA (Monadic Approach to Serverless Functions) to ease some of these problems.

2.1.2. Differences Between Commercial Serverless Function Offerings

There are significant differences between functionality and features of the three serverless function offerings (AWS Lambda, GCP Cloud Functions and Azure Function Applications) examined in this thesis. This section describes the support for functionality and features such as logging and monitoring to middleware and input validation that AWS provides developers on Lambda that do not have equivalents (or have weak equivalents that partially match AWS) on GCP Cloud Function and Azure Function Application.

In the desire to be cloud agnostic, some features are not straightforward to offer such as init functions and powertools of AWS Lambda. Init functions [92] are similar to init containers in Kubernetes [84] and allow developers to run some code when the serverless function is triggered or invoked for the first time to prepare for running the business logic in the serverless function. Powertools [136] allows developers to set up a cache for their serverless function and re-use the data stored in the cache in future invocations because the data is persisted to other invocations of the serverless function [34] (speed up serverless function and save costs in some cases like accessing secrets from AWS Secrets Manager [22]). A cold start of the serverless function will require running the init and setting up the cache again - this means developers need to do extra work to manage these resources and is a pollution of business logic.

AWS Lambda also has an extensive telemetry suite consisting of tracing, logging and metrics built in for developers to plug-in and use ([163], [100], [110]). Developers can add annotations to functions in business logic to use the telemetry features. In the listing 2.1, the annotations capture_lambda_handler and capture_method sends a trace of the handler and of other methods to AWS X-ray [24]. This can be used to capture traces of both synchronous and asynchronous methods, capture the values returned (or exceptions thrown), include annotations and metadata to filter logs during analysis, aiohttp, capture traces of threads [25] and concurrent asynchronous functions.

```
@tracer.capture_lambda_handler
def lambda_handler(event, context):
    ...
    update_item(”user1”, ”item1”)

@tracer.capture_method
def update_item(user_id, item):
    ...
```

Listing 2.1: Tracer example in AWS Lambda

Listing 2.2 shows an example of logging in AWS Lambda. The annotation inject_lambda_context should be used for prod environment because it gives vital information such as if the Lambda was a cold start, ARN (Amazon Resource Number) to uniquely identify the function etc. Adding log_event = True is recommended for non-prod environments because it logs additional information like the event that is given fed to the lambda as input. Developers can add metadata and annotations for log analysis to ease searching log files, choose between five different log levels, specify policies for logging such as minimum log level [38], set a debug sampling rate and a broad formatting suite to make useful logs [102]. Lastly, loggers can be extended by inheriting and overriding to build custom serializers and formatters [101].

Instead of building their own tools, GCP supports external telemetry tools like Cisco AppDynamics, DataDog, DynaTrace, New Relic and Splunk [121] which can be used to improve logging and tracing in GCP Cloud Functions (in addition to default logging provided out-of-the-

box with all serverless functions). At this moment, AWS officially supports DataDog, while Azure Function Applications has their own custom and native to Azure tool called Application Insights [79] that needs to be set up to receive some basic logging and monitoring features that AWS Lambda developers enjoy with little to no setup. In addition to all this, tools like Baselime [119] and OTEL [21] offer a large logging and telemetry suite for AWS but do not support GCP and Azure at the time of writing.

```
logger = Logger()

@logger.inject_lambda_context  # For prod
# @logger.inject_lambda_context(log_event=True)  # For non-prod
def lambda_handler(event: dict, context: LambdaContext) -> str:
    logger.info("Collecting payment")

    # You can log entire objects too
    logger.info({"operation": "collect_payment", "charge_id": event["charge_id"]})
    return "hello world"
```

Listing 2.2: Logger example in AWS Lambda

Tracer and Logger are for debugging and AWS also provides developers with an extensive monitoring suite for Lambda. Listing 2.3 shows how developers can request the monitoring of specific methods, capture cold starts, and add tags and metadata to improve visibility, allowing a graphical user interface to consume and plot statistics conveniently. The last line shows how developers can define custom metrics and measurements. Additionally, developers can set the resolution time for metrics [6], give additional weightage, or gain fine-grained control over the reporting of metrics through multivalued metrics in which the same metric can be given multiple values (AWS will put them on a list), add default tags so that it is persisted across lambda invocations, raise errors if some conditions for metrics are not met, and isolate multiple instances of the same metrics [110].

```
STAGE = getenv("STAGE", "dev")
metrics = Metrics()

@metrics.log_metrics  # ensures metrics are flushed upon request completion/failure
# @metrics.log_metrics(capture_cold_start_metric=True). # This will also capture cold
    start
def lambda_handler(event: dict, context: LambdaContext):
    metrics.add_dimension(name="environment", value=STAGE)
    metrics.add_metric(name="SuccessfulBooking", unit=MetricUnit.Count, value=1)
```

Listing 2.3: Metrics example in AWS Lambda

Developers can validate the input and output (or part of the input or output) of AWS Lambda against predefined JSON schemas by adding validator decorators [166]. To parse data, AWS Lambda provides Pydantic [132] which can also be added through decorators [123]. AWS has developed a vast prebuilt collection of parsers that can be quickly plugged in to parse input effortlessly or the parsers can be extended by overriding. Several envelopes are provided to unwrap, extract, and type cast the required object/value from complex input. AWS provides a neat solution for abstracting away polluting boilerplate code to verify input by validating objects against data types or specific criteria. Custom conditions including dynamic conditions based on input among numerous other possibilities can be applied quickly. AWS Event Source Data Classes [57] allows developers to create self-describing Lambda event sources and helper functions to deserialize and/or decode nested fields to simplify working with other cloud components

and services in AWS. In short, repetitive error prone pain points of receiving and validating input are made simple by providing ready made and highly customizable solutions.

AWS Lambda can batch process streams from various sources like SQS, Kinesis, and DynamoDB. AWS provides various primitives that can be easily used and extended by developers to fit specific use cases like batch processing asynchronous messages, integrating with Event Source Data Classes, input validation with Pydantic, and error handling including partial failure mechanisms.

AWS allows developers to use middleware in serverless functions (shown in listing 2.4) to run custom logic before and after every Lambda invocation synchronously [111]. Listing 2.4 shows an example that runs some code both before and after the business logic in the serverless function is executed [148]. The middleware can be quickly integrated with other features described above, such as tracing, logging, envelopes, validations, and batch processing. Middy is another open source tool for writing middleware on AWS Lambda [162] that offers developers an alternative with similar features. GCP does not officially have such features for middleware, but a user on StackOverflow suggests a workaround for GCP Cloud Function [153].

```
from aws_lambda_powertools.middleware_factory import lambda_handler_decorator

@lambda_handler_decorator
def middleware_before_after(handler, event, context):
    # logic_before_handler_execution()
    response = handler(event, context)
    # logic_after_handler_execution()
    return response

# @middleware_before_after(trace_execution=True) Optionally trace your middleware execution
@middleware_before_after
def lambda_handler(event, context):
    ...
```

Listing 2.4: Middleware example in AWS Lambda

2.1.3. A Closer Look at the Anomalies in Azure Function Application

Many cloud services and components of AWS, GCP and Azure are supported in MSA however, this thesis is primarily aimed at serverless applications and ecosystem of tools and technologies around serverless. AWS Lambda and GCP Cloud Functions have multiple things in common, such as hosting small pieces of code on lightweight infrastructure like MicroVMs Firecracker [61], [2]. AWS Lambda and GCP Cloud Functions are merely that, a URL that can be used to trigger the serverless function (execute the code). Microsoft Azure claims that Function Applications are the equivalent serverless function in Azure to AWS Lambda [144] and GCP Cloud Function [67]. However, in the documentation of Function Applications, Azure describes it as a ”serverless solution” [29] instead of the well-defined term ”serverless function”. This directly contradicts the two previous claims of Azure. StackOverflow describes Azure Function Applications as ”serverless compute” [170]. Azure also claims that WebJobs are equivalent to AWS Lambda [144], but WebJobs appears to have been made part of Function Applications now [15].

Functionally, hosting multiple small pieces of code in AWS and GCP requires multiple serverless functions as this naturally follows from the definition. On Azure Function Application (Python), every piece of code is placed under an endpoint annotation in the same Function Application.

Thus, Azure Function Application also has the capabilities of AWS and GCP API Gateway[5] and [14]. This forms the next major contradiction with the ideas of serverless - being light-weight and economical by having exactly the infrastructure and resources needed. A single serverless function can not be triggered or scaled as they all live together in Azure Function Application.

Azure's claim that Function Applications are serverless functions is unfounded due to limitations on performance, cost and resource efficiency. As a result Microsoft Azure does not have serverless functions and if Azure fixes this in the future, it will also be possible for developers to migrate applications built with MSA to and from Azure. The rest of the chapter will explore the differences in creating serverless functions on the a cloud and writing business logic for the serverless function with code examples.

2.2. Infrastructure as Code

There are multiple ways to create serverless functions and other infrastructure in the cloud. The easiest way is manually through the website (called Console) of the cloud provider, but it is prone to human error in selecting configurations. A reliable and scaleable approach is to automate by using an API of the cloud provider or Infrastructure as Code tools like Terraform and Pulumi. [83] describes Infrastructure as Code as "process of managing and provisioning computer data center resources through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools". In simpler words, it is a declarative description (in yaml, json, etc.) of the components in a cloud environment. It is written in code and can therefore scale, be automated, and the code can be semantically versioned. Versioning and tagging a cloud environment and services/components in the environment removes dynamic elements like human errors that limit reproducibility, reliability and scalability. Combined with CI/CD pipelines, Infrastructure as Code will give a complete end-to-end solution for reliably creating, managing, and deleting cloud resources.

Pulumi [127] is an Infrastructure as Code tool that offers an API to provision and manage infrastructure in the cloud. Pulumi works in two steps, a Preview stage and then an Up stage. This is similar to a compile and run/execute step in many programming languages. The preview step constructs a Directed Acyclic Graph with specific details (also called a plan) of the actions Pulumi will perform in a cloud environment. The up stage is to apply the plan to the cloud environment. Pulumi stores a state on a local machine or in the cloud on a S3 bucket for example. The state is a description of the cloud environment according to the actions performed previously by Pulumi. When creating the directed acyclic graph, Pulumi compares the actual state of the cloud with the stored state of the cloud (local machine or S3 bucket) to detect a drift which is differences between the two states. Developers must necessarily resolve drifts by either adding/deleting the appropriate services and components to infrastructure code to match those that exist in the cloud or by asking Pulumi to create/edit/delete the resources on the cloud that do not match the infrastructure code. This can be a highly destructive and sometimes difficult to reverse operation such as when a database without backups is deleted. Pulumi has other useful features like CI/CD [128] and secret management [130] to ease the development of applications.

There have been attempts in the past to unify cloud infrastructures of AWS, GCP and Azure into a common framework that did not succeed (discussed in Chapter 3) for various reasons. Listing 2.5, 2.7 and 2.8 show the creation of a simple serverless function using Infrastructure as Code in Pulumi [127]. Terraform [157] is another popular Infrastructure as Code tool in which infrastructure is declaratively described in yaml files. Listing 2.6 shows an example of Listing

2.5 in Terraform. The other two listings are written similarly in Terraform.

```
import pulumi_aws as aws
import pulumi_archive as archive

aws_serverless_function = aws.lambda_.Function(<name>,
    code=pulumi.FileArchive(<path-to-code>),
    role=<iam role>.arn,
    handler=<serverless function handler>,
    runtime=<serverless function runtime>
)
```

Listing 2.5: AWS Serverless Function

```
resource "aws_lambda_function" "<resource-name-for-lambda>" {
    function_name    = <name>
    filename         = <filename.zip>
    source_code_hash = <archive file object of filename>.lambda.output_base64sha256
    role             = <iam role>.arn
    handler          = <serverless function handler>
    runtime          = <serverless function runtime>
}
```

Listing 2.6: AWS Serverless Function

Among these code examples to provision a simple serverless function in AWS, GCP and Azure, Azure is the most plagued by complexity because of the tertiary details required to create a serverless function that are unrelated to the code that runs in the serverless function and the serverless function itself. AWS and GCP are relatively straightforward with AWS requiring an IAM role to grant lambda permission to execute itself and other permissions if the lambda needs to interact with other services and components. GCP Cloud Function similarly require an "invoker". Building complex serverless functions requires many more configurations and often other components from the cloud. Just like this example of a simple serverless function, the other services and components with complex configurations are similarly unique and non-trivial to unify.

```
import pulumi_gcp as gcp

gcp_serverless_function = gcp.cloudfunctions.Function(<name>,
    runtime=<serverless function runtime>,
    source_archive_bucket=<code bucket>.name,
    source_archive_object=<code object in bucket>.name,
    trigger_http=True,
    entry_point=<<serverless function handler>
)
```

Listing 2.7: GCP Serverless Function

```
import pulumi_azure as azure

aws_serverless_function = azure.appservice.FunctionApp(<name>,
    resource_group_name=<resource group>.name,
    location=<resource group>.location,
    app_service_plan_id=<plan>.id,
    storage_account_name=<account>.name,
    storage_account_access_key=<account>.primary_access_key
```

```
)
```

Listing 2.8: Azure Serverless Function

These examples show that creating a serverless function on different clouds has significantly different requirements. Similar differences in infrastructure code also hold for almost all other cloud services and components such as databases, api gateways, message queues, compute and networking hardware related services, security, access and permissions related components, etc. In MSA, developers can write infrastructure code for an arbitrary cloud, and Monads with the help of Pulumi are used to create the corresponding infrastructure on a specific cloud. The next section looks at the differences (polluting boilerplate code) imposed upon business logic by the cloud provider.

2.3. Specific Code for Serverless Functions

A developer should ideally write the same business logic in any serverless function, irrespective of where it is run but the signature of the handler (starting point of the serverless function, similar to main method in many languages) in the big three cloud providers has unique annotations and different parameters. Often, serverless functions by GCP have different handler signatures depending on the use-case (business logic in the serverless function). Additionally, AWS, GCP and Azure provide boto3 [17], functions_framework [68] and Azure-Functions [31] libraries to interact with other services (e.g. message queue, SQL database, etc.) in their respective clouds. The behavior, capabilities, and interactions with these libraries are specific to the cloud provider. Thus there are multiple hurdles in migrating a serverless function from one cloud provider to another - although the business logic and computations the developer wants to perform do not change.

The handler in AWS Lambda has a consistent signature shown in Listing 2.9. GCP and Azure serverless functions have different handler signatures depending on how the function is triggered, GCP http 2.10, GCP message queue 2.11 and Azure HTTP 2.12 are shown below. The parameters in the handler contain different information in the three cloud providers that are shown by accessing HTTP headers and query parameters passed when triggering the serverless function with a curl request. There are some similarities between some of these examples but clearly, an uniform and consistent standard for the handler of serverless functions is absent. Cloud providers provide a SDK with their serverless functions to ease development (writing business logic and interacting with other services and components in the cloud), but they too are incompatible with each other.

```
from json import dumps

def <function-name>(event, context):
    # HTTP query parameters can be accessed with
    query_parameters = event.get("queryStringParameters") or {}
    # HTTP headers can be accessed with
    headers = event.get("headers") or {}

    # Write business logic here

    return {
        "statusCode": <http status code>,
        "body": <output of code>
```

```
    }
```

Listing 2.9: AWS Lambda Handler

```
import functions_framework

@functions_framework.http
def <function-name>(request):
    # HTTP query parameters can be accessed with
    query_parameters = request.args
    # HTTP headers can be accessed with
    headers = request.headers

    # Write business logic here

    return body
```

Listing 2.10: GCP Cloud Function Handler for HTTP trigger

```
import functions_framework
from base64 import b64decode

@functions_framework.cloud_event
def <function-name>(cloud_event):
    # The message put into message queue can be accessed with
    message = base64.b64decode(cloud_event.data["message"]["data"]).decode("utf-8")

    # Do something with message (business logic)
```

Listing 2.11: GCP Cloud Function Handler for Message Queue trigger

```
import azure.functions as func
import datetime
import json
import logging

app = func.FunctionApp()

@app.route(route="<route>", auth_level=func.AuthLevel.ANONYMOUS)
def <function-name>(req: func.HttpRequest) -> func.HttpResponse:
    # HTTP query parameters can be accessed with
    query_string_parameters = req.params
    # HTTP headers can be accessed with
    headers = req.headers

    # Write business logic here

    return func.HttpResponse(
        status_code=<http status code>,
        body=<output of code>
    )
```

Listing 2.12: Azure Function Application Handler for HTTP trigger

There are sufficient compatibility challenges in writing non-trivial code for serverless functions for AWS, GCP and Azure. Putting this together with the previous section about specific code for infrastructure code on different cloud providers, the picture is bleak for developers and

the wider software and technology communities when it comes to cross cloud compatibility and vendor lock-in. To paint a fuller picture, making a serverless function with a hello world program is trivial because it does not solve a non-trivial problem. Building useful applications on the cloud has various aspects, such as scalability, security, reliability, and maintainability of the infrastructure, but also the cloud environment. This requires numerous cloud components and services to work together such as API Gateways, message queues, serverless functions, IAM Roles/permission sets, various kinds of databases and caches, logging, alerting and monitoring, etc. and the business logic in the serverless function must also align appropriately with the other cloud services and components. Some discrepancies between cloud providers are major and can only be solved by the cloud providers themselves. Other discrepancies are minor, but putting all of them together forms a formidable problem of vendor lock-in for developers and consumers of cloud services and technologies. [58] concluded that monads can be used for serverless computing and abstracting state in serverless functions. This thesis goes further on to build cloud agnostic serverless applications using monads which are the basis of this thesis for solving vendor lock-in.

2.4. Monads

The additional code required by a serverless function for business logic code pollutes the code in the serverless function as a whole. Cloud providers having unique polluting code worsens the situation but is a frequently occurring problem in other parts of software development. Monads were first introduced in Haskell [60], [4], a master planned functional programming language to neatly solve numerous problems while simultaneously keeping business logic clean and functional in nature. [171] mentions some examples in which Monads are used to abstract away control flow and thereby reduce boilerplate code for common operations. This enables developers to always remain in control of the boilerplate code while maintaining the purity of business logic. The primary application of monads in MSA is encapsulating side-effects, state, and keeping the code clean, functional, and stateless. Monads are structures that combine functions and wrap their return value into a type with some additional computation. This additional computation allows developers to encapsulate tedious polluting code behind a wrapper and keep the business logic code clean. A simple example of a monad is Either that encapsulates the type of a value [51]. The type can be any of the two explicitly specified types and developers can simply use the value of a specific type. Similarly, the Cloud Monad wrapper in MSA encapsulates the polluting boilerplate code for that respective cloud provider so that developers can write pure business logic code that will be portable and compatible across any cloud provider. MSA combines this concept with Infrastructure as Code to solve the serverless functions themselves having specific infrastructure code on different cloud providers by encapsulating the specifics of a cloud provider in a Cloud Monad.

Another feature of Haskell is function composition in which the output of running a computation of a function is passed to another function i.e. compose two functions into a single function. Listing 2.13 shows in Line 2 that the functions f and g can be chained with the dot operator. The example below it computes the odd numbers in the list of integers from 1 to 9 [103]. Such a composition is convenient to build complex programs with serverless functions and it is critical for this composition to be cloud agnostic to solve vendor lock-in.

```
1 # Mathematical notation
2 f (g x) = (f . g) x
```

```
3
4 # Example
5 map (\x -> not (even x)) [1..9]
6 # can be re-written to
7 map (not . even) [1..9]
```

Listing 2.13: Function Composition in Haskell

To summarize this chapter, even though the three biggest cloud providers - AWS, GCP and Azure have similar features and functionalities, there are severe discrepancies between the ecosystems, supporting tooling and libraries. The developers have paltry control over how, where, and on what their code runs on, resulting in the inability to give functional and non-functional SLA/QoS guarantees for a serverless application. The few that are still possible stem directly from the cloud provider, such as uptime and security. Additionally, serverless functions impose requirements on developers such as timing out in few minutes with limited resources and writing pure code (functional programming) to take advantage of the concurrency and scale the cloud can offer. The control flow of serverless functions simply do not have any provisions for enabling developers to reliably perform such actions in a cloud agnostic setting because it either means changing the control flow in a breaking and disruptive way or a smarter approach such as Monads which encapsulates these potentially problematic aspects and elegantly provides a backward compatible solution.

3

Related Work

This chapter details previous efforts to solve the problems mentioned above. There has been a distinct delineation in responsibilities and use-cases between infrastructure and application. Thus, all previous efforts have focused either on abstractions for application/code or feeble attempts to unify infrastructure.

Apache jClouds [12] and Dasein-Cloud [50] are multi-cloud toolkits for creating portable across cloud applications in Java. [3] checks portability of AWS EC2 and GCP Compute Engine workloads. [112] claim to test jCloud's serverless application portability and usability but are merely evaluating putting jar files into a serverless function. The code in jar files and interactions with other cloud components are elementary. Neither JClouds nor Dasein help developers create portable infrastructure and their documentation says that the best developed abstractions are for Blob Storage and Compute Instance Management. Dasein has not been maintained since 2014 while jClouds supports few GCP and Azure components. [125] and [11] try to create an abstraction library, have problems similar to jClouds, and have not been maintained for over 4 years.

Taibi et al. [152] describes FaaSifying arbitrary python code for an AWS Lambda. A similar approach can be taken and extended for serverless functions on other clouds and templates. The selected approach is inefficient and leaves the task of integrating a serverless function with the cloud infrastructure to the developer (unlike native serverless functions). [138] uses object oriented programming principles like inheritance to abstract away the tedious boilerplate code in serverless functions. However, this means that an overhead (extra library) is created, and the authors discuss neither infrastructure nor smooth integration of the serverless function with other cloud services and components.

Apache Libcloud [13] is a Python library for infrastructure and supports a wide range of cloud providers for infrastructure related to Compute (instance management), Load Balancer, Object Storage, Container, Backup and DNS. Terraform [157], Pulumi [127], and Serverless Framework [145] are Infrastructure as Code tools with a wide variety of cloud providers and, unlike Libcloud, supports everything on AWS, GCP, and Azure with a separate API for each cloud. However, each cloud has a unique infrastructure code with no compatibility between clouds on both. Terraform, Pulumi and Libcloud are simply a common platform from where infrastructure on a cloud can be

described in declarative syntax and then provisioned on various clouds.

OpenFaaS [118], KNative [90], Fission [62], OpenWhisk [122] provide a common interface for serverless functions running on top of containers or a Kubernetes cluster that can be run on any cloud. Apart from not integrating smoothly with other cloud components (like native serverless functions do), this also introduces extra complexities like Kubernetes, containers, etc. that need to be managed by developers. The idea of running containers inside containers is suboptimal and hence all of these are considered invalid solutions.

[175] suggests an analytical approach SEAPORT (SErverless Applications PORtability assessmenT) to determine if a serverless function can be ported to another cloud. This strategy is applicable to a subset of serverless functions, whereas any serverless function should be portable. [177] proposes a model in which developers create a serverless function multiple times for every cloud, a benchmark suite evaluates all the serverless functions to find the cloud that delivers the best performance, and then selects that cloud for that serverless function in the future. To avoid duplicate infrastructure in multiple clouds, the authors also created a common interface so that components in a cloud can talk to components in another cloud. This can create efficient multi-cloud systems where each components can be deployed on a cloud best suited for it but the problem lies in the fact that each serverless function (and component) needs to be created multiple times, once for each cloud.

Pulumi-cloud [129] (different from Pulumi) comes the closest to solving the challenges outlined. Pulumi-cloud creates their own API that abstracts away infrastructure, and the business logic in serverless function can interact with infrastructure through a common API. The challenge with this approach is the existence of a common API which introduces additional complexity compared to writing business logic in a way that is native to a cloud provider. Pulumi-cloud is still a Work-in-Progress at the time of writing and has released support for AWS in public preview, Azure in early stage preview, and has announced that GCP will be supported in the future. MBrace [1] is built for data scripting in a cloud agnostic fashion in F# and C# with support for Azure [108] and soon AWS [107]. It is possible to write scripts for specific tasks but is not a general purpose library for solving vendor lock-in. Modal [161] is another serverless platform for data, AI and ML tasks. They do not make any claims for agnosticity and instead run all workloads on a proprietary cloud environment.

There are other efforts in rapidly building serverless applications like Vercel [167], Netlify [42] and Dark [49] but they are limited to frontends and hosting websites with serverless functions. They do not offer equivalents of the vast array of services and resources like AWS, GCP and Azure. These tools do not expose the underlying infrastructure and are hence not designed for cloud agnosticity but rather for rapidly building specific serverless applications in the specific platform of the tool. AWS SAM [23] provides templates to quickly build serverless applications. The templates are a short-hand syntax for optimized Infrastructure as Code that only works on AWS.

Shuttle.rs uses Infrastructure from Code (IfC) to generate backends and other serverless applications in AWS. In IfC, developers write their business logic and tools like Shuttle [32], Klotho [88], SST [33] and Ampt [64] generate required infrastructure code for AWS using Infrastructure as Code tools like Terraform and Pulumi. Klotho suggests to have support for GCP and Azure in their introduction and tutorials but at this moment their GitHub repository [89] does not show evidence of support for either GCP or Azure in main branch. [160] claims that Ampt (previously Serverless Cloud) deploys to an AWS environment owned by Ampt whereas all other tools deploy to the AWS environment of the developer. Ampt takes away transparency and visibility of

the resources on a cloud environment but also demands developers trust Ampt to secure the cloud environment. Further, Ampt they also claimed that Ampt makes abstractions that restrict the ability to create/manage individual resources.

Nitric [159] another IfC tool supports AWS, GCP and Azure. Nitric documentation claims twice that Nitric is ”cloud agnostic” [131] and [158] but the example code on GitHub [116] does not have Azure and has duplicate infra code, once each for AWS and GCP. Writing code multiple times, once for every cloud is not cloud agnostic. Further, [160] reports that Nitric has significantly increased delays in cold start invocations of serverless functions. Nitric admitted this [43] and explained that the cold starts happen because business logic for the serverless function is containerized and then run in an AWS Lambda. Cloud Native Computing Foundation (CNCF) [141] states multiple times that serverless is achieved through virtualization.

4

Design of Monadic Serverless Applications

This chapter takes a top-down model to explain the design and working of MSA - Monadic approach to Serverless Functions. Section 4.1 and 4.2 will introduce how Pulumi (Infrastructure as Code) is used and monads are implemented in Cloud Monads with an example while Section 4.3 explains the design of Cloud Monads. Section 4.4 shows how monads are used to bridge differences between features offered by cloud providers, and Sections 4.5 and 4.6 provide some commentary on other aspects of MSA.

4.1. Tooling

A resource in Pulumi is a class (of Object Oriented Programming) whose object represents a service or component in the cloud. Instantiating such an object or resource involves Pulumi making an API call(s) to the respective cloud to create the actual service, and deleting the object deletes the actual resource in the cloud. If the object stores information such as data or maintains tertiary responsibilities such as permission sets, then the appropriate policies set explicitly by the developer or the default policies of the cloud provider apply to the creation and deletion of that resource.

Pulumi has a native resource for every component and service of AWS, GCP, and Azure that can be used to provision, maintain, and destroy infrastructure on the cloud. A cloud monad is made by creating a wrapper around the native resource of Pulumi. Developers program against the wrapper, which can contain extra logic to give developers extra features and ease the development of specific and custom resources. In MSA, monads are implemented in a similar way. The extra logic in the wrapper is used to translate cloud agnostic infrastructure code to the specific code required for a particular cloud. This cleanly abstracts away the specific code of a cloud provider and assists the developer in evading vendor lock-in because developers are never exposed to the specific code of any cloud provider. A limiting factor of MSA is the desire to create perfectly equivalent serverless applications on various cloud providers because this can create potential side effects. Chapter 2 lists numerous differences, many of which can be resolved neatly through monads. Although monads have a reputation for solving complex prob-

lems in easy, clean and convenient ways, some problems, such as a common architecture for a serverless function, can be resolved by either GCP and Azure adding support for arm64 architecture or AWS pulling support for arm64 (to match GCP and Azure which only have x86_64). At the moment, MSA only supports features that are commonly supported by both AWS and GCP.

A benefit of using Pulumi resources to implement the cloud monads instead of making direct API calls to the cloud is to leverage the extensive CI/CD tools that come with Pulumi [128]. To migrate a serverless application from one cloud to another, Pulumi first creates the equivalent serverless application in the other cloud and then deletes the resources in the cloud the migration is done from. Thus, by using MSA, developers gain the ability to use modern end-to-end CI/CD tooling out-of-the-box.MSA has three parts, 1) Cloud Agnostic Infrastructure Code 2) Cloud Agnostic Business Logic (code in serverless function), and 3) other features offered as monads to users to solve the problems introduced. Writing cloud agnostic infrastructure code can be complex because in serverless functions because of differences like trigger mechanisms of serverless functions and other events however the idea of the trigger is largely the same across all cloud providers. This can be leveraged by using monads to encapsulate these differences so that developers are not exposed to these differences.

The key challenge preventing developers from writing cloud agnostic business logic are the differences centered on the interactions with other cloud services and components. The first of two possible solutions is to create a superset of frameworks and services offered by the various clouds so that developers write code against the superset that at runtime selects and executes the appropriate code for that context. The drawback is that developers may not use all the features in the superset which bloat the serverless function. This is inconsistent with the ideals of serverless computing. Another approach is to ask developers to state the requirements of a serverless function (and interactions with other cloud services and components) which can then be used to synthesize the code inside the serverless function (business logic provided by developer + other things required for cloud agnosticity). The synthesis is unique for every cloud and situation. Every interaction a serverless function can have with another cloud service or component has a prebuilt template that code synthesis uses to create the cloud native code that can be executed by a serverless function. Templates can be easily extended, replaced, and new templates can be added to meet specific niche edge cases. This approach ensures that all the requirements of the serverless function are met and that the serverless function is efficient and lightweight (inline with principles of serverless computing) without exposing developers to the specifics of any cloud. The next section uses a simple example to build a cloud agnostic serverless application.

4.2. FooBar

FooBar (code available here[1]) is a simple API with two endpoints /foo and /bar that have a serverless function containing a small piece of business logic at the end of each endpoint. To build FooBar, two serverless functions, an IAM role and an API Gateway (shown in listing 4.1) are required. The IAM roles grant the necessary permissions so that the serverless function can be invoked and executed and the API Gateway exposes the serverless functions to the public internet.

```
1 from pulumi import ResourceOptions
```

[1] https://github.com/rkochar/msa/blob/master/utils/foobar.py

```
2  from utils.monad import Monad
3
4
5  def apigw_foobar():
6      m = Monad()
7
8      apigw_faas_iam_role = m.create_iam(role_name="a-faas-iam-role", role_file="faas-
       basic-role", attachment_name="faas-role-attachment", policy_name="faas-iam-policy"
       , policy_file"faas-basic-policy")
9
10     foo_function = m.create_faas(name='foobar-foo', code_path="foobar/foo", handler="
       foo.foo", runtime="python3.10", template="http", role=apigw_lambda_iam_role)
11     bar_funtion = m.create_faas(name='foobar-bar', code_path="foobar/bar", handler="bar
       .bar", template="http", runtime="python3.10", role=apigw_faas_iam_role, imports=["
       pydantic", "numpy"], is_timed=True)
12
13     routes = [
14         ("/foo", "GET", foo_function, "foobar-foo", "Serverless function for endpoint /
       foo"),
15         ("/bar", "GET", bar_function, "foobar-bar", "Serverless function for endpoint /
       bar"),
16     ]
17     m.create_apigw('foobar', routes, opts=ResourceOptions(depends_on=[foo_function,
       bar_function], replace_on_changes=["*"], delete_before_replace=True))
```

Listing 4.1: FooBar: Infrastructure as Code

For convenience, a single IAM role is re-used for both serverless functions in line 8. Lines 10 and 11 make the serverless function with names "foobar-foo" and "foobar-bar". The template parameter informs MSA how the serverless function will be triggered and how the results computed by the serverless function are to be returned to the triggerer of the serverless function. In this case, the functions are triggered by HTTP through an API Gateway and results are returned in the same way. The handler tells the serverless function which method in which file is the starting point of execution of business logic when the serverless function is triggered. Line 11 also declares three specific requirements of that serverless function. Numpy and Pydantic libraries are declared as requirements for the business logic in the serverless function and the developer has requested to measure the execution time of the business logic in the serverless function. The routes pentuple in line 13 defines the endpoints, HTTP request type and the serverless function that will be triggered. The API Gateway on line 17 simply implements this pentuple for every endpoint and conveniently hides the internals and plumbing of the serverless application from the outside world. This infrastructure code is generic and does not contain specific code or configurations that are specific to a cloud provider. This is critical in enabling migrations from cloud to another to solve vendor lock-in.

Listing 4.2 shows the business logic in the Bar serverless function. It generates a random number using numpy and checks the headers made with the curl request for a few conditions to make a valid transaction. Just like the previous listing on Infrastructure Code, the business logic is regular Python 3.10 code. This enables the business logic to be used in any serverless function in any cloud without specific requirements or extra conditions other than support for Python. The FooBar program proves the capability of MSA to write elementary programs that are cloud agnostic in infrastructure code and business logic. Later in this chapter, other useful capabilities like message queues and different types of databases are introduced.

```
from numpy.random import randint
from pydantic import BaseModel, Field, ValidationError, validator


def bar(headers, query_parameters):
    try:
        transaction = Transaction(sender=headers.get("sender"), receiver=headers.get("
    receiver"), amount=headers.get("amount"))  # The Transaction class is defined in
    Section Input Validation
    except ValidationError as e:
        return f"Errors found: {e.error_count()}. {e.errors()}"
    result = randint(0, 5)
    return f"Transfer {transaction.amount} from {transaction.sender} to {transaction.
    receiver}. result: {result}"
```

Listing 4.2: FooBar: Business Logic of Bar

To complete the serverless functions, the cloud agnostic business logic needs to be combined with other code that is written in a cloud native manner to meet the specific requirements for that serverless function (Section 2.1.2) and other specific requirements for a serverless function to interact with specific cloud services and components. MSA abstracts away the specifics and mundane requirements from developers to preserving the purity of business logic. This is inline with monadic principles of not restricting developers but rather, safely exposing the features of the cloud to the developers. Code synthesis is used to achieve this and MSA is able to offer extreme amounts of customizability to developers (just like monads) while maintaining the sanctity of business logic.

Listings 4.3 and 4.4 show the synthesized code for the serverless functions on AWS and GCP respectively. An example of how features that are not native to a specific serverless function in a specific cloud can be added in a cloud agnostic manner is shown. Lines 11, 17 and 18 for AWS and lines 10, 16 and 17 for GCP add a timer monad is added to measure execution time of the business logic in the serverless function (as requested in line 11 of 4.1).

```
import json
import boto3
from os import getenv
import re
from time import time
from uuid import uuid4


def template(event, context):
    name = "foobar-bar"
    start_time = time()
    telemetry = None
    query_parameters, headers = event.get("queryStringParameters") or {}, event.get("
    headers") or {}

    body = bar(headers, query_parameters)

    end_time = time()
    execution_time = str(end_time - start_time)
    print(f"execution_time: {execution_time}")

    return {
        "statusCode": 200,
```

```
        "body": json.dumps({"body": body, "execution_time": str(time() - start_time),
    })
    }

# <Business logic of Bar from abvove is here.>
```

Listing 4.3: FooBar: Bar in AWS

```
import functions_framework
from os import getenv
from time import time
from uuid import uuid4


@functions_framework.http
def template(request):
    name = "foobar-bar"
    start_time = time()
    telemetry = None
    query_parameters, headers = request.args, request.headers

    body = bar(headers, query_parameters)

    end_time = time()
    execution_time = str(end_time - start_time)
    print(f"execution_time: {execution_time}")

    return {"body": body, "execution_time": str(time() - start_time), }, 200

# <Business logic of Bar from above is here.>
```

Listing 4.4: FooBar: Bar in GCP

4.3. Cloud Monads

This section describes the working of Cloud Monads, each of which abstract a cloud services or component across multiple clouds. Each cloud monad is unique because it is carefully crafted to match the infrastructure configurations on the various cloud and simultaneously taking into account the relevant business logic required to interact and/or work with that cloud service or component. All of the below monads work on AWS and GCP but have varying degrees of applicability on Azure because Azure at this moment does not have sufficient and eligible serverless services and offerings to build nontrivial serverless applications (Section 2.1.3).

4.3.1. Serverless Functions

Serverless functions are the core of serverless applications and are responsible for performing computations. The Serverless Function Cloud Monad encapsulates AWS Lambda, GCP Cloud Functions, and Azure Function Applications. The default trigger (template) of every serverless function is HTTP(S) and can be changed by explicitly stating a different template in infrastructure code. This is done along with stating the other services and components on the cloud the serverless function wishes to interact with. A serverless function on AWS can be triggered in dozens of different ways such as when a value that satisfies a regular expression is updated in a DynamoDB table or when a message is sent to AWS's AI chatbot Lex [20], [165]. GCP Cloud

Functions can only be triggered in two different ways, 1) through a HTTP(S) request either directly to the serverless function or through an API Gateway or 2) by a message queue.

This difference where serverless functions in GCP can not be triggered directly by events is solved by using a Trigger Cloud Monad such that developers are not exposed to this complexity. All events such as updating a value in a database will publish a message to a dedicated message queue that will trigger the serverless function. The Trigger Cloud Monad creates all the required infrastructure and code synthesis ensures required extra code is synthesized without any extra effort from the developer. In AWS, the triggers are straightforward and the parameters are simply passed on to Pulumi. Regular expressions can be used to filter triggering events but this is not supported on GCP. A similar approach as Section 4.3.5 can be taken to resolve this and other differences in trigger mechanism. The Trigger Cloud Monad encapsulates all the complexities but the approach taken for GCP causes a drop in performance compared to AWS. This is unavoidable and expected because of the extra message queue and other code introduced in GCP. Ideally GCP should offer features like clean and efficient trigger mechanisms to developers.

The business logic inside a serverless function may require libraries to ease development of complex functionality. Dependencies can be made available inside a serverless functions by explicitly stating the dependencies in infrastructure code. GCP Cloud Functions and Azure Function Applications conveniently import libraries from pip with a requirements.txt generated by the Serverless Function Cloud Monad. AWS complicates importing libraries because they have made optimizations to make serverless functions lighter. A Lambda Layer [173] is similar to a layer in a container such as Docker. The Layer is designed to contain read-only binaries of libraries so that the Layer can be reused by other serverless functions. An additional benefit is that a Lambda strictly contain business logic making a Lambda lighter than an equivalent GCP Cloud Function and Azure Function Application. The Layer is built with specified libraries in the developer's local machine (or CI/CD pipeline if used) and uploaded to a S3 bucket for the Lambda to access. Another complexity that is abstracted away from developers is the architecture of AWS Lambda which can be x86-64 or arm64. This affects the layer when the architectures of local machine used to create the layer is arm64 which is at odds with the serverless function of x86_64.

The Serverless Function Cloud Monad has several other responsibilities that relate to interactions with other services and components on various clouds that are discussed in the next paragraphs. The next subsection will discuss exposing the serverless function to the public internet and HTTP(S) triggers including code synthesis.

4.3.2. API Gateway

AWS allows serverless function to be exposed naked to the public internet [93] and GCP gives from authentication which can be disabled [81]. The default endpoints generated for serverless functions are unique which make it inconvenient to build an API with serverless functions. The API Gateway fixes this. A pentuple provided to the API Gateway Cloud Monad is used to generate Swagger documentation according to the specifications of AWS [5] and GCP [14]. The pentuple allows developers to describe their endpoints and the serverless function that will be triggered upon hitting the endpoint without exposing the internals (different semantics of the Swagger yaml file), routing and serverless functions to the users.

Serverless functions that are triggered through an API Gateway should use the handler

shown in Listing 4.5. MSA synthesizes code that is native to the cloud on which the serverless function exists. To effortlessly write cloud agnostic business logic, MSA provides developers easy access to headers and query parameters passed along with the HTTP(S) request. MSA extracts these fields by parsing the specific inputs received by serverless functions and these inputs are different on different clouds.

```
def <function-name>(headers, query_parameters):
    # Business logic
    return <string>
```

Listing 4.5: Cloud agnostic HTTP trigger example

4.3.3. Message Queue

Triggering serverless functions with message queues achieves two objectives. Complex computations can be performed in serverless architecture by chaining serverless functions in serverless function composition and allows computations to be performed asynchronously. This is critical in serverless architecture because idling in synchronous calls is against the principles of serverless that postulate efficient usage of resources by deleting resources when they are not needed. One approach for this can be a serverless function making an asynchronous HTTP(S) call to the next serverless function but this suffers from unreliability in networking such as message not delivered or delivered with corrupted data. Message queues such as AWS SQS (Simple Queue Service) [8] and GCP Pub/Sub [126] are an excellent asynchronous alternative because message queues guarantee the delivery of a message which can then be used to asynchronously trigger the next serverless function in the serverless function composition.

Pub/Sub of GCP gives strict ordering guarantees of messages compared to a regular message queue of AWS SQS. To compensate, SQS has a FIFO (First In First Out) mode, which is enabled when the name of the topic has the suffix `.fifo`. FIFO queues also force AWS to have at most one instance of an AWS Lambda running [35] at a time. In AWS, serverless functions can batch process messages in a message queue, but not in GCP. Thus, for simplicity, this thesis limits the scope to each message in a message queue triggering a serverless function. The Message Queue Cloud Monad returns a Python dictionary that should be passed along as an environment variable to a serverless function so that the serverless function can communicate with a message queue.

Similar to the HTTP(S) template discussed above, Listing 4.6 shows the unified function signature of a serverless function that will be triggered by a message in a message queue. GCP automatically deletes a message after it is consumed by a subscriber, but AWS does not. The code required to equalize this behavior between AWS and GCP is synthesized in the business logic of AWS serverless functions. If a serverless function states in the infrastructure code that it wishes to publish messages to a message queue, a method `publish_message` is made available to do so. Developers can simply call this method and pass the data to be published in the queue as a string.

```
def <function-name>(message):
    # Business logic
```

Listing 4.6: Cloud agnostic Message Queue trigger example

4.3.4. SQL Database

Databases are particularly complex because in addition to the differences in design decisions by different cloud providers, they also suffer from discrepancies in consistency guarantees and data models between alternative databases on various cloud providers. MSA has limited the scope to the cheapest instance of MySQL databases of AWS RDS (Relational Database Service) [7] and GCP CloudSQL [63] for the purposes of this thesis. The engine version is another challenge in building cloud agnostic serverless applications because all versions of MySQL are not supported by all cloud providers. This problem can be properly solved by the cloud providers themselves and MSA sidesteps it by selecting MySQL v8.0.34 because it is supported on both AWS and GCP. Another complication is that AWS demands that every RDS live inside a Virtual Private Cloud (VPC) but GCP allows a CloudSQL instance to live outside a VPC. Thus, the SQL Database Cloud Monad also sets up the VPC and required networking and communication infrastructure behind the scenes with no extra effort required by developers. CloudSQL requires creating an User [71] and so the cloud monad also creates a default user on both AWS and GCP.

Another challenge in writing pure cloud agnostic business logic is connecting to SQL databases. AWS RDS and GCP CloudSQL have their own specific ways and there are extra networking complications like VPC and firewalls. The code synthesizer carefully synthesizes required code to connect to the SQL database and also adds a method `execute_sql_query` so that developers can simply write their SQL query without polluting business logic. `execute_sql_queries` enables writing efficient programs and designing complex business logic by batching queries into a transaction before committing or flushing to the database. The SQL Database Cloud Monad is an example of how monads add complexity in the form of layers and abstractions to deliver several benefits without sacrificing on the ability to design complex and secure architecture involving plumbing and security but also writing complex and efficient business logic.

4.3.5. Blob Storage (S3)

The S3 Cloud Monad is used for various purposes. When a developer submits code to a serverless function, MSA zips the code and puts the zip file into a S3 bucket (Cloud Storage in GCP and Azure Storage Service in Azure) behind the scenes. The serverless function then picks up the zip file and loads the business logic. Some other usecases are storing data for data intensive computations, persisting state of a serverless function and using S3 to trigger other serverless functions. In AWS, 28 different events can trigger a Lambda [56] but there are only 4 [72] event types in GCP. Additionally, AWS allows filtering events with regular expressions [26] but in filters are not supported. This means a serverless function in GCP can be triggered on false positive events (note: in GCP events are published to a message queue that trigger a serverless function and MSA cleanly abstracts all of this away). Lastly, not only do the Python libraries of AWS S3, ”boto3” [140] and ”storage” of GCP [133] match these differences in properties, but they also have their own specific API calls that are not portable from one cloud to another.

The differences in filtering events used as triggers and the Python libraries have been solved by the S3 API Cloud Monad by abstracting away the two Python libraries and most event trigger types (some event trigger types like lifecycle rules in AWS must first be supported by GCP). The S3 API Cloud Monad unifies interactions with S3 on AWS and GCP such as inserting and (conditionally) retrieving data so that developers can program against a common API. Code synthesis ensures the required code is available in the serverless function. In addition, regular

expressions on a trigger have also been standardized. In AWS, no extra work is required as it is supported by the cloud out of the box. When conditions are specified in GCP, each potential triggering event is checked by the unified S3 API and the trigger is terminated if all conditions are not satisfied. This is not efficient because a serverless function is triggered in GCP on all events (GCP charges developers for this), but AWS developers save money because the situation of false negatives does not arise. Thus, in GCP, a combination of S3 Cloud Monad that creates a message queue to enable triggering a serverless function with S3 events and the S3 API Cloud Monad that provides developers additional critical features, such as regular expressions to filter triggering events, provide developers with a rich experience that is comparable to AWS with the additional benefits of escaping vendor lock-in.

4.4. Other Cloud Monads

Some other Cloud Monads are offered as features to developers and serve as a proof of concept to show how differences in features of serverless functions such as built-in telemetry can be resolved. All of these monads need to be explicitly requested in infrastructure code by a serverless function to receive that specific feature delivered through code synthesis. Many of these features can be implemented directly into the business logic of a serverless function. It is not recommended because implementing these features will pollute the business logic.

4.4.1. Telemetry

In OpenTelemetry, an observability framework for telemetric data such as traces and logs [168], a span is the smallest unit that contains information about an unit of work (serverless function in this thesis), and a tree of spans are together called a trace [164]. It is useful for debugging complex architectures and workflows. Section 2.1.2 describes the different levels of support for telemetry in serverless functions by various cloud providers. All of those (non native) solutions are unclean/bulky (a central server and other resources are needed such as networking, storage, etc) and some are paid services requiring a subscription. AWS Lambda has its own native Telemetry API [96] and some open source projects like AWS's OTEL Lambda [21] and OpenTelemetry Lambda [120] (in addition to X-Ray). Both of these are added as a prebuilt layer to a AWS Lambda which is a neat and clean solution that does not complicate infrastructure code nor does it pollute business logic. However, these tools are specific to AWS Lambda which solidifies vendor lock-in.

The cloud agnostic Open Telemetry Cloud Monad generates traces. At the moment, the information available in a span is the parent_name (previous span in the trace), parent_span_id (span id of parent's span) and span_depth (number of executions in the chain so far). This is extensible and developers can easily add their own specific use cases into it (besides making their own Open Telemetry Cloud Monad in MSA). Every serverless function in a function composition must explicitly request the opentelemetry monad otherwise the chain is broken and the span is reset.

4.4.2. Execution Time

This Cloud Monad times the execution time of the business logic in a serverless function. If the serverless function is triggered by a HTTP request then the execution time is returned as output in the json under the key "execution_time" with rest of the output. For message queue or other triggers it is, the execution time is obtained in the logs. It is passed on with telemetry for a

complete view of the serverless function composition.

4.4.3. RAM

Similar to the Execution Time Cloud Monad, the RAM Cloud Monad measures the RAM used by the serverless function and the result is returned with the HTTP response or can be retrieved in the logs of the serverless function.

4.4.4. Input Validation

AWS Lambda has an elegant input validation mechanism (discussed in Section 2.1.2). A simple approach to offer clean input validation in through by Pydantic [132] in the business logic. In line with the monadic principles, the parameters' requirements are explicitly stated and delivered to the developer. Listing 4.7 shows an example in which a serverless function takes three parameters as input from the headers of the http request and verifies the conditions specified in the transaction class. `strict=False` coerces the datatype if it does not match (string to int in the example) and unique_sender_receiver ensures that the sender and receiver are unique to make a do a valid transaction. This snippet is cloud agnostic, frees business logic from tedious boilerplate code and the only requirement is that Pydantic be declared as a dependency so that MSA can import Pydantic.

```
from pydantic import BaseModel, Field, ValidationError

try:
    transaction = Transaction(sender=headers.get("sender"), receiver=headers.get("
    receiver"), amount=headers.get("amount"), strict=False)
except ValidationError as e:
    return "Errors found: " + str(e.error_count()) + "\n" + str(e.errors())

class Transaction(BaseModel):
    sender: int = Field(ge=1, le=5)
    receiver: int = Field(ge=1, le=5)
    amount: int = Field(ge=1, le=10)

    @validator("receiver")
    def unique_sender_receiver(cls, receiver, values):
        if "sender" in values and receiver == values["sender"]:
            raise ValueError("Sender and receiver must be unique")
        return receiver
```

Listing 4.7: Pydantic: input validation

4.5. Supported Programming Languages

MSA has been implemented entirely in Python and Python is the only language supported in a serverless function at the moment. Python was selected because it is highly malleable and can be quickly and easily molded into what is required to build MSA. With the insights gained, adding support for ductile and stricter languages like Java and Go is relatively straightforward. The language used for cloud agnostic infrastructure code is less important because it is merely a description of the services and components in a cloud environment. Languages such as Go, Java, JavaScript and others supported by Pulumi (except YAML) can be used for infrastructure code without major challenges.

The degree of malleability of the language affects the complexity and maintainability of the implementation but not the capability of the serverless functions in a language different from Python. A constraint to keep in mind is that the serverless function templates use APIs provided by the cloud provider to interact with various cloud services and components. Writing and running cloud agnostic business logic in Python serves as a reference for making MSA programming language agnostic. The templates need to be adapted to the new language, and an equivalent for packaging/importing of libraries and dependencies needs to be implemented. For Python, it is sufficient to list the names of libraries for pip to install them. Languages such as Java using Maven or Gradle or Dotnet will require an appropriate solution. Code synthesis and monads such as the timer monad do not require significant changes but they need to be adapted for the new language.

Pulumi can also be swapped with another Infrastructure as Code tool, or even implementing the API calls to the cloud provider directly are viable. The monadic approach is independent of all tools and programming languages and other criteria.

4.6. Policy as Code

Policy as Code allows developers to describe rules in code for cloud services and components such as[2] blob storages such as AWS S3, GCP Cloud Storage and Azure Storage can not be accessed from the public Internet. Naturally, Policy as Code is plagued by the same problems of vendor lock-in and specific configurations. MSA solves it by making monadic abstractions over Pulumi's policy as code tool [73]. This is merely an interesting side feature and not part of the core of this thesis.

4.7. Conclusion

The chapter started by looking at how the Cloud Monads are constructed and used to provision, manage, and destroy cloud services and components using Pulumi. A simple serverless application FooBar showed both the ease of creating and migrating infrastructure from one cloud to another. The business logic in a serverless function is cloud agnostic because the interactions with other cloud services and components such as message queues and SQL database are done through a unified API provided by MSA. The individual differences between a service and it's near equivalents on other clouds are abstracted away. Code synthesis is used so that a serverless function always interacts in a native manner with other services and cloud components. This ensures that developers are able to take advantage of all features on a cloud that might not be possible with a non-native approach. Cloud Monads such as S3 release developers from the tedious tasks of implementing their own filters for queries to S3 and filtering out false positive triggers, but there is little that MSA can do to deliver permanent relief from such exploitative designs by any cloud provider. Monads such as the execution time, telemetry, RAM and input validation show how useful features for telemetry, monitoring, debugging, etc. can be made cloud agnostic and used to overcome the lack of compatibility between serverless functions across various cloud providers.

[2] `https://github.com/rkochar/msa/blob/master/policy-as-code/__main__.py`

5

Building Cloud Agnostic Applications

This section describes the applications made with MSA and are later used in chapter 6 for benchmarking.

5.1. MVCC

A challenge with building serverless applications is the requirement for serverless functions to be idempotent programs (for parallelization) and have negligible side effects and state. Some solutions like Apache Airflow [10], Step Functions [155] and Workflow [70] are frequently used for complex stateful computations. This MVCC program builds a Multi-Version Concurrency Control (MVCC) database by using first principles and function composition to perform complex stateful computations in a functional programming approach instead of using previously mentioned tools.

MVCC is a type of non-locking database [114] in which users who wish to write to the database submit a new version of the data with the user's changes. This means that the original data are immutable and allow strong read consistency and scalability/parallelization, while write consistency and scalability/parallelization of writes are determined by the conflict resolution strategy. In our use case, there are 10 bank accounts with 10 euros each at the start and transactions are made from one account to another of between 1 and 5 euros (these numbers are selected randomly and can be changed). According to MVCC, making a transaction involves creating a new version with the updated amounts. According to MVCC, the write operation of making a transaction does not affect reads at all because reads will use the last available amounts until a new version is released.

Figure 5.1 shows the architecture to implement this program in AWS and GCP has the same diagram without the VPC at bottom left. One SQL database is made with multiple tables to save cost and the tables are not shared. Read and write serverless functions have multiple instances of serverless functions, each with their own SQL table as depicted by the three lambdas and three AWS RDS tables under the Read-only instance. Init and Control have an instance of the serverless function each. There are three endpoints, init, read (check amount in account) and write (send x euro to account y). Init sets up the SQL tables to prepare for the first transaction. The read endpoint will fetch the latest versions available and write will accept transactions. If a transaction is valid (sanity check, input parsing, etc.), the transaction is proposed to Control

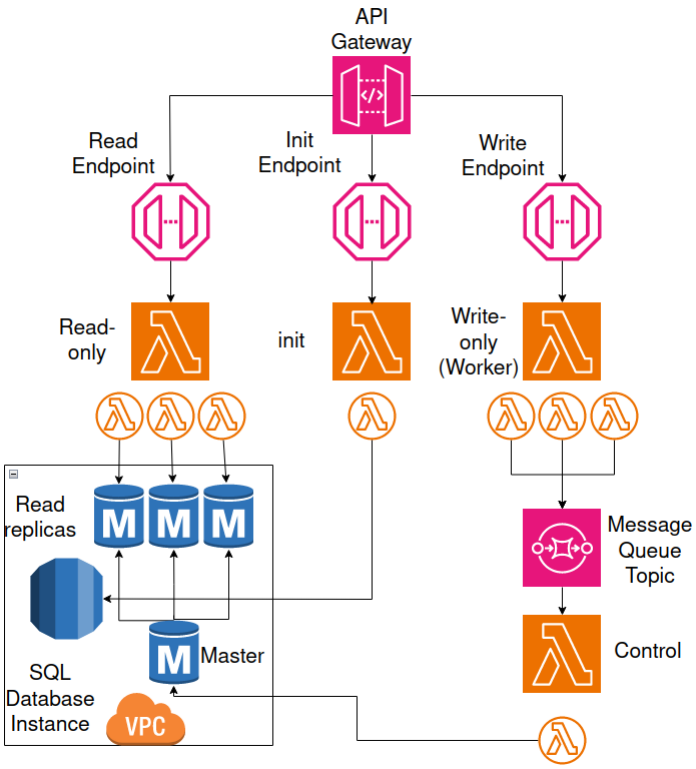

Figure 5.1: MVCC Diagram for AWS

though a message queue. If Control can successfully verify that the transaction is possible by checking the balance of the sender, the transaction is acknowledged and a new version is published and the relevant tables are updated with the latest version.

5.2. Serverless MapReduce

Berkeley view on serverless computing [86] identifies 5 problems (including MapReduce) that stretch serverless computing to its limits. MapReduce is a complex distributed program to write in stateless functional programming because it involves steps such as splitting the work and data into chunks that are given to workers for parallel computations, and then the results are collected again in the map stage. The reduce stage has similar steps which require tracking and aggregating upon the results of the mapper to yield the final answer. Serverless functions are short-lived, which further complicates this. AWS first proposed a serverless reference architecture (Refarch) using S3 and Lambda [146] and later started offering their own serverless spark offering called Glue [18]. GCP followed with a serverless spark offering [151] as well.

Refarch is re-created with MSA so it will be cloud agnostic. The original architecture and business logic are mostly preserved, but there are some changes. Refarch had a driver program

that acted as a coordinator and ran on the local machine. This allowed Refarch to make http calls to serverless functions but our mapreduce will run completely on the cloud, which means the driver program needs to be triggered by a curl request to an API Gateway and it needs to be split into smaller pieces because of the timeout on serverless functions. In addition, all http calls need to be replaced with appropriate triggers.

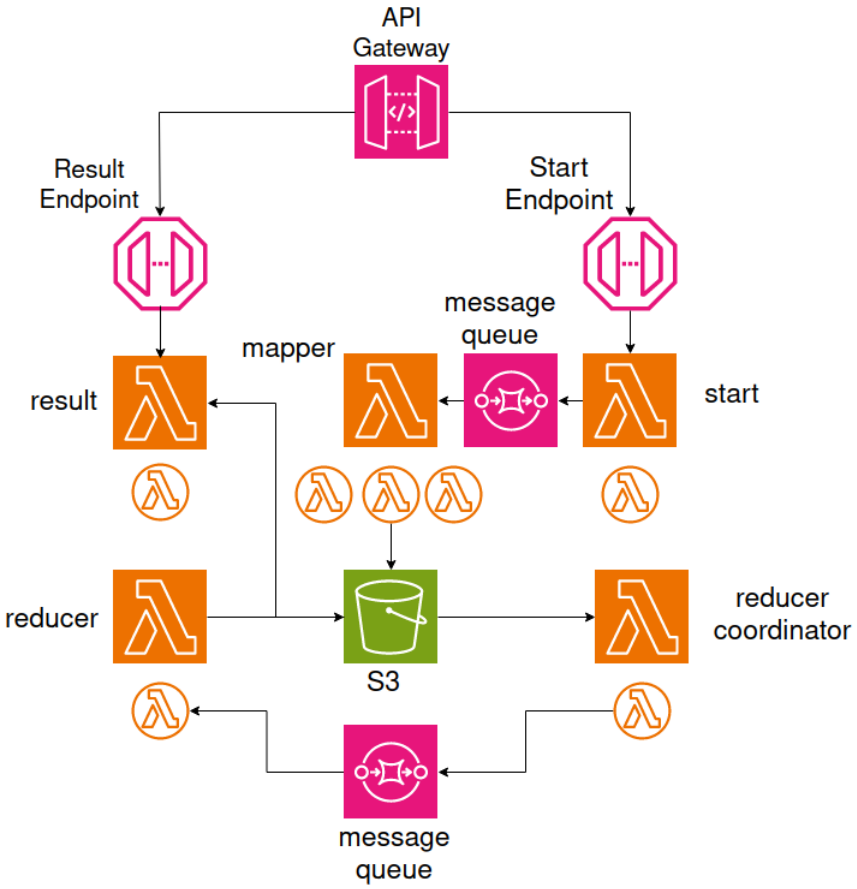

Figure 5.2: MVCC Diagram for AWS

Figure 5.2 shows the architecture of mapreduce built with MSA. The start endpoint triggers the start serverless function that distributes work to the mappers. Start calculates the amount of work, size of data and RAM of the serverless functions to make chunks and marks each chunk with an id. Every chunk is published along with the assigned id in the message queue to trigger multiple mappers in parallel. Additionally, the distribution of work called jobinfo is also persisted by writing to S3. The mapper performs the map step and writes the results to S3 using the assigned id in the name of the file and each file uploaded to S3 triggers the Reducer Coordinator that checks if every chunk in the map step has been processed using jobinfo to trigger the Reducer that completes the MapReduce step and writes the final answer to S3. The

result endpoint simply fetches the final answer from S3 for convenience of the user.

5.3. Serverless ZooKeeper

Copik et al. [44] made a serverless implementation of ZooKeeper in AWS called FaasKeeper. At the time of writing, there is a pull request [69] with a serverless implementation of Zookeeper on GCP that reuses neither business logic nor infrastructure code. An attempt was made to rebuild FaasKeeper with MSA first on AWS and then migrate to GCP. There are non-trivial challenges because FaasKeeper uses DynamoDB triggers (trigger an action when a value is added/updated in DynamoDB database) which are defined in infrastructure code through EventSourceMappings [47]. The equivalent of DynamoDB in GCP is Google Spanner [149] but the philosophy of GCP is that triggers should be in business logic [150]. This is a serious hindrance in creating a cloud agnostic serverless ZooKeeper because of the hard delineation of responsibilities between infrastructure code and business logic. Four potential solutions are to create a monad to solve this difference in philosophy, use a different database in GCP that supports triggers in infrastructure code (like DynamoDB on AWS), migrate DynamoDB triggers defined in infrastructure code to business logic in AWS or to migrate from DynamoDB to a database that is supported on both AWS and GCP such as MySQL or PostgreSQL with a common philosophy for triggers.

A monad can be made to bridge the gap between the responsibilities of infrastructure code and business logic however eliminating or diluting this gap destroys the logical isolation between infrastructure code and business logic. The isolation/loose coupling gives much needed confidence to developers, businesses and other users that their services will not be severely impacted by small trivial mistakes because the contagion is localized. Although a monad can solve the problem, it will introduce a greater evil. The challenge with databases is that they tend to have unique query languages to help users take maximum advantage of the trade-offs made in the design of a database. Using different databases in different clouds forces developers to interact with databases in a query language independent approach. Building a one-to-one mapping of query languages can be a solution but this is unnecessarily complicated and bloats business logic and/or negatively impacts performance. Solving compatibility and vendor lock-in in databases is outside the scope of this thesis, and it is desirable to use an existing solution instead of building one ourselves. The last two potential solutions have similar drawbacks - ZooKeeper is used for low latency atomic writes. Putting triggers in business logic (and also checking for false positives in triggers) will increase latency, which is overwriting a core property/feature of ZooKeeper [82]. Should GCP support defining triggers in the infrastructure code for Spanner, similar to what AWS does on DynamoDB, this is expected to become feasible. The last solution is to use a common database such as SQL. MySQL and PostgreSQL are not NoSQL like DynamoDB and Spanner, however they can give similar properties on a single machine but that totally defeats the purpose of ZooKeeper which is to be a ”Distributed Coordination Service for Distributed Applications” [178]. Thus, all solutions either considerably damage the highly desirable features of Zookeeper, or the solution introduces a greater evil than the problem we initially start with.

6 Evaluation

This chapter describes the various experiments performed to evaluate MSA and answer the research questions. To determine the quality of serverless applications built with MSA, they are compared with equivalent counterparts (referred to as native) built according to best practices recommended by respective cloud providers. This will reveal the downsides if any of MSA. Section 6.1 explains the setup and assists the reader in reproducing the experiments by describing how to prepare the infrastructure and the steps taken before making measurements to solve unpredictable behavior such as cold starts of serverless functions. Sections 6.2, 6.3, and 6.4 evaluate the performance of three serverless applications. MVCC is a non-locking, scaleable and distributed database, MapReduce is a data intensive serverless application, and Zookeeper is a latency serverless sensitive application. Section 6.5 discussed the latency of MVCC and Zookeeper and Section 6.6 gives the reader a gist of the insights gained in this chapter.

6.1. Procedure

To solve the unreliability of cold starts in serverless functions on AWS and GCP, serverless functions are warmed up by sending a random large number of requests for about a minute and then a request is sent every 5 seconds for a total of 10 requests. The serverless functions invoked by these 10 requests are measured with the execution time and ram usage monads described in Section 4.4. This procedure is used for all experiments involving serverless functions on AWS and GCP including baselines from other sources. The number of concurrent executions of serverless functions is set to the default value of 1 and the autoscalar is enabled, which means that if a default threshold selected by the cloud provider is crossed, another instance of the serverless function will be created (or destroyed when the load falls below the threshold). When the program uses function composition to perform complex computations over multiple serverless functions, timestamps from logs are used to calculate the exact end time of operations. All experiments are done within the free tier of AWS and GCP except the VPC and networking needed for SQL databases in MVCC is paid. Unless explicitly stated, the default parameters are selected for all configurations in the cloud. The experiments are carried out using Python 3.10 during working hours on working days at the AWS and GCP data centers in Ireland and Belgium, respectively.

6.2. MVCC Database

To study the overheads introduced by MSA, three baselines are selected for comparisons. MVCC is implemented on Kubernetes and natively on AWS and GCP [115]. Kubernetes is selected because it is frequently used in numerous industries for hosting applications. The implementation on Kubernetes uses microservices in Python to replace serverless functions, and a MySQL database for AWS RDS and GCP CloudSQL. The message queue is Kafka and run locally on a Docker Desktop Kubernetes cluster [54] on a MacBook M2. The values in this experiment are calculated by taking the average of 10 consecutive valid and accepted transactions after the microservices have stabilized (same as described in Section 6.1). The box plots show the 10 measurements for each variant, and the number used for the calculations below is the mean, unless otherwise specified. In every experiment, Kubernetes is compared with MSA implementations on AWS and GCP followed by a comparison of MSA with the native implementation in that cloud.

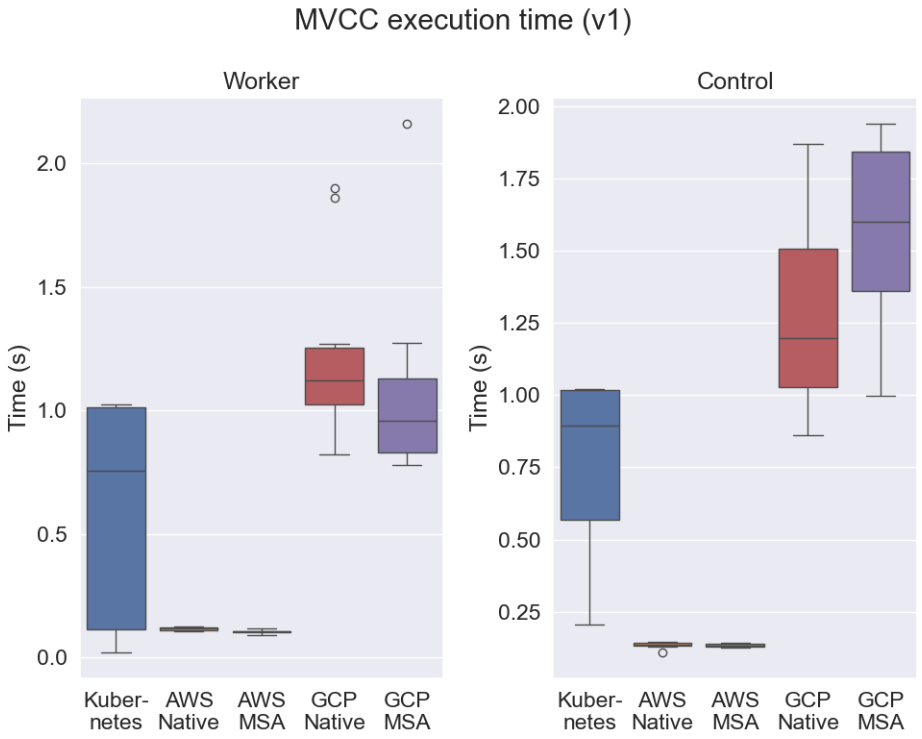

Figure 6.1: Execution time of Worker and Control

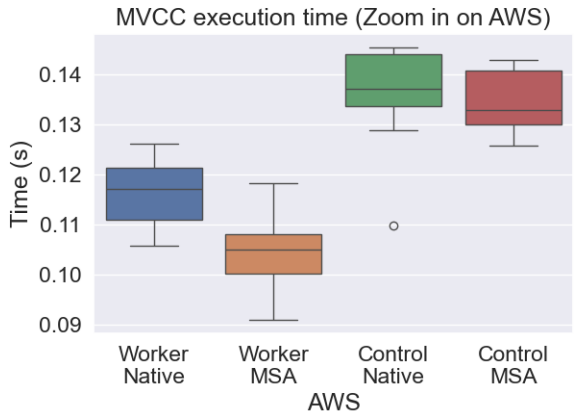

Figure 6.2: Execution time of Worker and Control of AWS

Figure 6.1 compares execution time in seconds of Worker and Control serverless functions and Fig. 6.2 zooms in on AWS. In Worker, the implementation on Kubernetes the fastest responses in the tails which is expected because it is running on a local machine instead of an enormously complex public data center of a cloud provider. AWS has a mean execution time of 0.104632s and has the smallest range observed while Kubernetes has a mean of 0.601930s (475% more than AWS) and the largest range among the 5 measured variants. This is also expected because of the implementation of Kafka listeners polling a queue waiting for an event in classical microservice fashion, whereas in serverless, a function is triggered near instantaneously after the triggering event occurs. GCP has a mean of 1.078641s (930% and 79% higher than AWS and Kubernetes, respectively). Looking at the two implementations in the respective cloud providers, the mean MSA is faster than native by 11.2% in AWS and 13.6% in GCP.

Control shows similar trends to Worker. Noteworthy difference is that there are longer tails in Kubernetes and in GCP. Control writes the transaction to the database so interactions with a SQL database can explain the longer and variable executions times. Kubernetes requires on average 0.770943s which is 473% more than AWS which needs on average 0.134539s. GCP is the slowest at 1.569475s, which is 1066% and 103% more than AWS and Kubernetes, respectively. AWS MSA of Control is faster by 0.9% than its native equivalent but GCP is slower by 25% and is the only instance in which MSA is not faster than its native counterpart in Figure 6.1. GCP MSA can be slower than Native in Control but faster than Worker because the code synthesis for writing to the database in GCP is suboptimal.

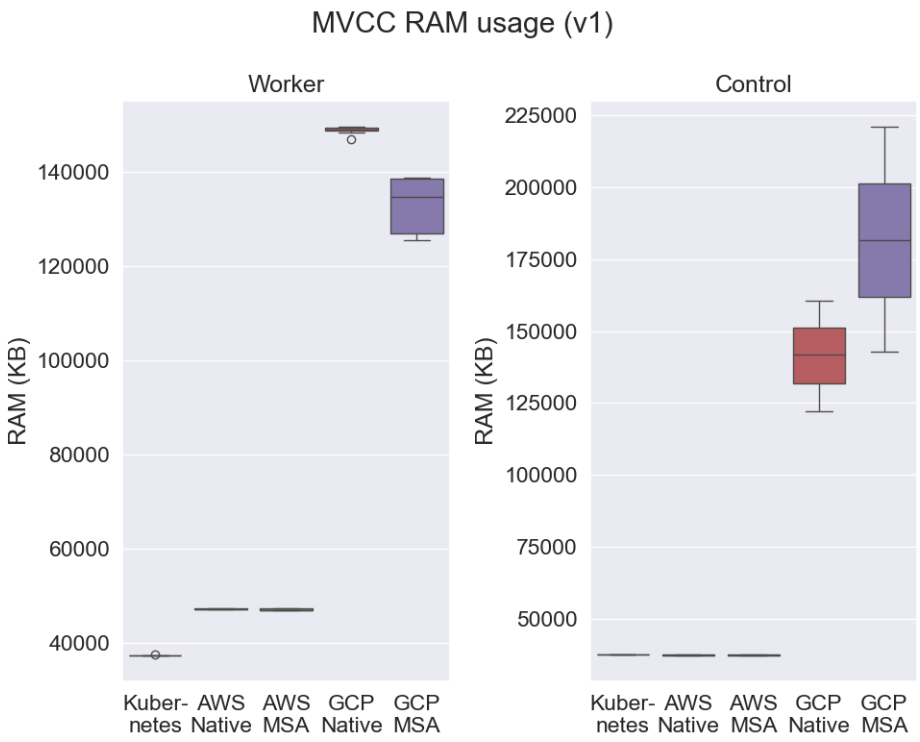

Figure 6.3: Ram usage of Worker and Control

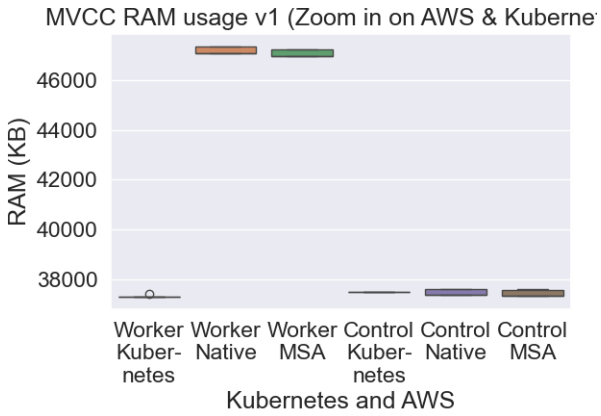

Figure 6.4: Execution time of Worker and Control of AWS

Figure 6.3 examines the ram usage in KB of Worker and Control serverless functions on Kubernetes, AWS and GCP (MSA and native implementations), and Fig. 6.4 is a zoom in on the AWS and Kubernetes boxplots. GCP requires more RAM than AWS and Kubernetes in both Worker and Control. In Worker, GCP MSA required 182% and 256% more than AWS and Kubernetes, respectively. Kubernetes needed the lowest at a mean of 37324 KB and AWS requires an extra 26% at 47069 KB. AWS Native requires an additional 0. 26% on average over AWS MSA, while GCP Native needed an additional 11.8% over GCP MSA. In Control, AWS MSA required the lowest amount of RAM but AWS Native and Kubernetes are both within the margin of error. GCP MSA is 385% higher than both Kubernetes and AWS MSA. It is interesting that Control GCP MSA requires 28% extra RAM than Native (whereas it was 11.8% lower in Worker). This trend is also consistent with execution time where MSA outperforms Native in Worker but not in Control. GCP requires significantly higher RAM and execution time despite having the same business logic and programming language compared to Kubernetes and AWS (MSA and Native). A possible explanation for this is that the SQL library Google provides is significantly slower than the SQL libraries on AWS (boto3) and the MySQL Python Connector client used in Kubernetes.

Figure 6.5: Execution time of Worker and Control (latency optimized)

To test the fastest execution times, v2 is a latency optimized write-ahead approach can be used. The key difference from v1 described earlier, is that after Control receives and ac-

cepts a transaction, the transaction is logged and acknowledged. The transaction is later asynchronously written to the database by using a message queue to trigger another serverless function. In some edge cases, this can cause transactions to be incorrectly accepted or rejected, and this is a conscious design decision to optimize latency. Figure 6.5 shows the execution times of Worker and Control in seconds. The trend is similar to v1 with Kubernetes and GCP showing a large range of values but AWS consistently has a small range and is the fastest on average in this experiment while GCP is the slowest along with long tails. Interestingly, v2 is faster than v1 for Kubernetes and GCP but not AWS. In Worker, the execution times of latency optimized Kubernetes, GCP Native, and GCP MSA are 45%, 9% and 16% faster than v1 but AWS Native and AWS MSA are slower by 92%, 29.5% than v1. In Control, Kubernetes, GCP Native and GCP MSA are 68%, 25% and 36% faster than v1 while AWS Native and AWS MSA are 65% 100.3% slower respectively. Looking at Control, this suggests that for a serverless function, communicating with the SQL database RDS on AWS is faster than communicating with the message queue SQS. However, Kubernetes and GCP can publish a message on Kafka and Pub/Sub faster than updating a row in SQL databases, MySQL and Cloud SQL respectively. This can be because of optimizations (or tradeoffs) made in the respective clients, hardware, networking, or service (message queue and SQL Database) itself by cloud providers (and vendors for Kubernetes, Kafka, and MySQL). A more thorough study is required to provide more insight because it is possible that the AWS SQS Python client is slow, which makes AWS RDS look fast. It is not clear by Worker shows significantly different numbers in v2 than v1.

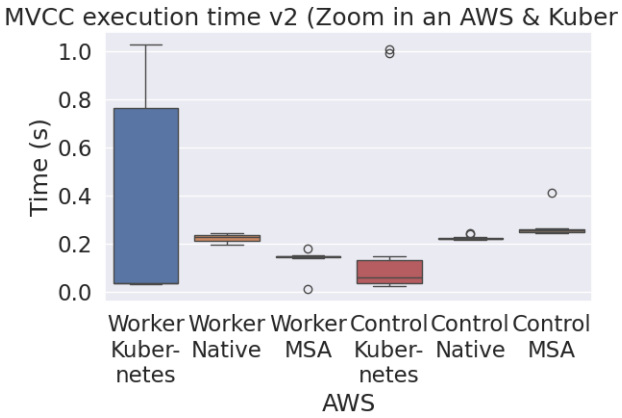

Figure 6.6: Execution time of Worker and Control (latency optimized) of AWS

Figure 6.7 shows the RAM usage in v2 which is roughly the same as before. In Worker and Control, Kubernetes needed 0.7% and 0.17% lesser RAM than v1. AWS shows a similar trend in Worker and Control where Native requires 0. 56% and 0. 9% lesser RAM than v1 and AWS MSA 1.16% and 1.7% respectively. GCP however shows wild swings with Worker needing -4.8% and 11% more RAM while Control needs 8% and 31% lesser RAM.

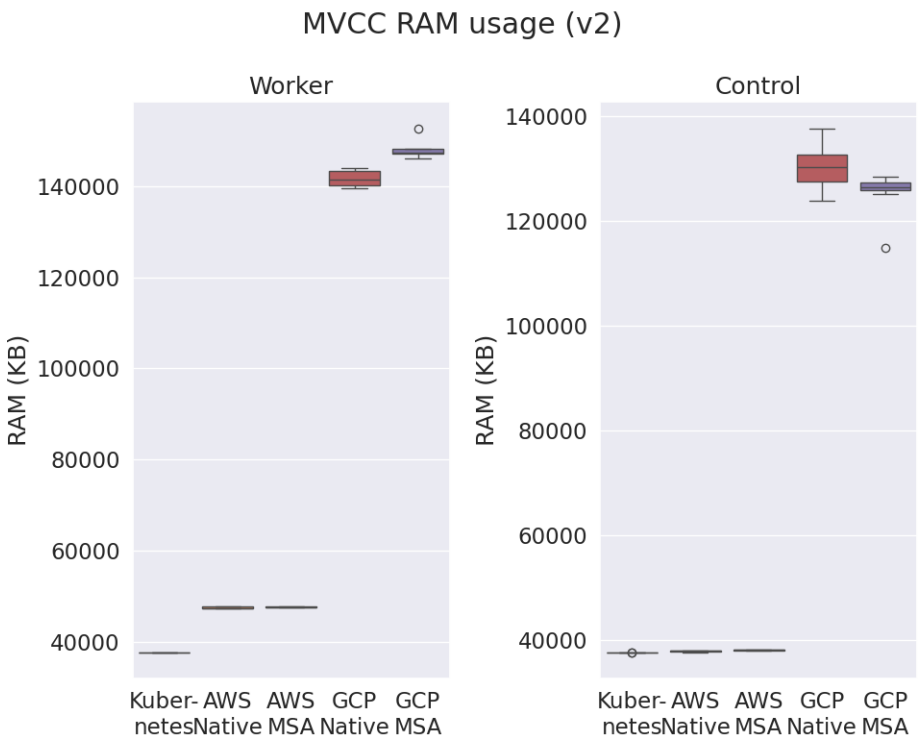

Figure 6.7: Ram usage of Worker and Control (latency optimized)

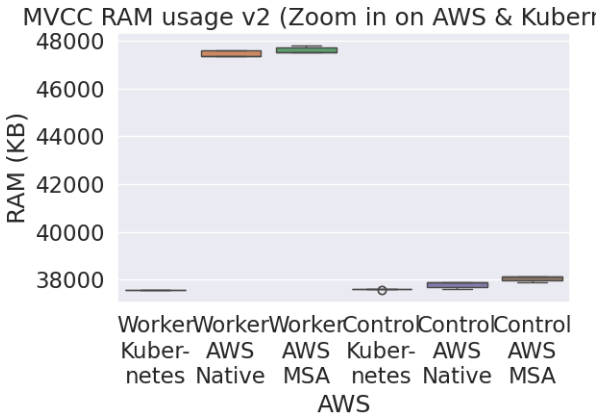

Figure 6.8: Ram usage of Worker and Control (latency optimized) of AWS

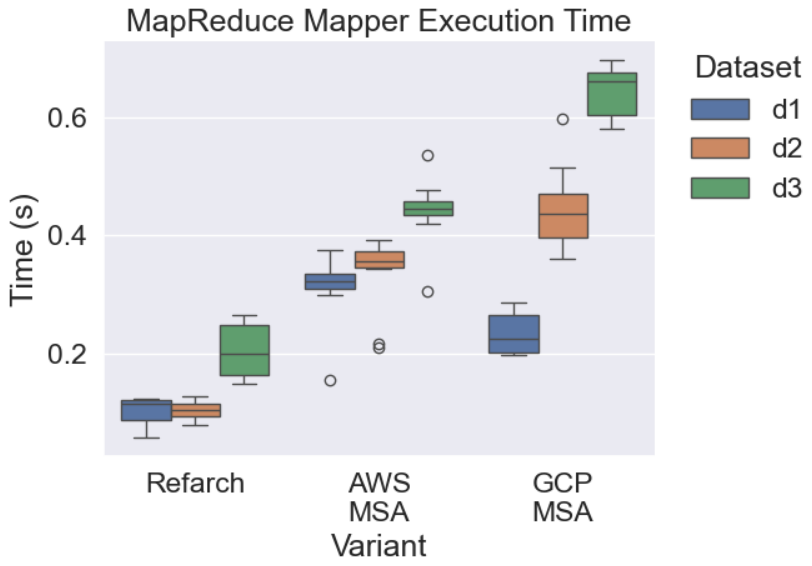

Figure 6.9: Mapper Execution Time

6.3. Serverless MapReduce

D1, D2 and D3 are randomly selected subsets of UC Berkeley's amplab1 dataset [9] where D1 is 10% (15KB) of the total dataset and D2, D3 are twice the size of D1 and D2 respectively. Mapper and reducer serverless functions correspond to the map and reduce steps in MapReduce. The selected baseline is the modernized version of AWS's reference architecture for mapreduce and is referred to as Refarch. Unfortunately, Refarch has not been maintained and predates numerous developments in the cloud because it is written in Python 2 and uses AWS configurations that are not supported. Refarch has been updated to Python 3.10, deprecations (on AWS configurations) have been fixed, and it's dependencies have also been modernized. The structure and logic of Refarch is preserved but it is important to note that it's driver program and reducer programs run locally whereas our mapreduce built with MSA runs entirely on the cloud. Refarch's default line count program is used and the measurements are the average of 10 consecutive invocations where the mapper and reducer step of mapreduce can involve multiple invocations. After each dataset is run, everything is torn down and deployed fresh for the next dataset.

Figure 6.9 shows the execution times (seconds) of Mapper. Refarch's implementations are the fastest, 66% and 55% faster than AWS and GCP on d1, 69% and 76.5% in d2 and 53.5% and 68% in d3. The size of the data set doubles in each experiment, and therefore a proportional increase in execution times expected. Refarch does not consistently show this and MSA AWS has a steeper slope going from d2 to d3 than from d1 to d2. GCP however shows a consistent linear increase. Interestingly, MSA AWS has longer tails than others showing some fast executions but they are still slower than Refarch. The measurements on the tails of MSA AWS are

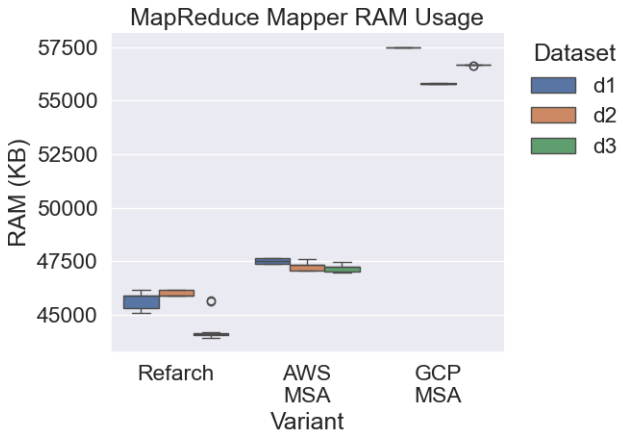

Figure 6.10: Mapper RAM Usage

faster than those of MSA GCP although on average MSA GCP outperforms MSA AWS by 25% on d1 and underperforms by 32.6% and 45.6% on d2 and d3, respectively.

Figure 6.10 shows curious results in the RAM usage (KB) of Mapper serverless functions. D1 is expected to have the lowest and d3 the highest but d3 is the lowest on all 3 implementations on serverless mapreduce. This is probably due to buffering loading of the data from S3. Better results of buffering are seen in larger datasets of d3 than d1. In Refarch, d2 does the worst while d3 has a long upwards tail but in AWS MSA, the RAM usage is consistent across the datasets and the benefits of buffering are neatly visible. In GCP, the RAM usage is more consistent than AWS MSA and Refarch but requires 21%, 18% and 20% more RAM than AWS MSA in d1, d2 and d3 respectively. There are long tails in Figure 6.9 for both AWS and GCP, but proportional tails are not seen in RAM usage (Fig. 6.10) suggesting that the tails are due to interactions with S3 on AWS and Cloud Storage in GCP.

Figure 6.11: Reducer Execution Time

Figure 6.11 shows execution times of Reducer. Since the line count program is used, the reducer needs to aggregate the results by reading from and writing the final answer to a S3 bucket (Cloud Storage for GCP). This is an interesting graph because on AWS, d2 and d3 take significantly more time than d1 (likely because of more interactions with S3 to aggregate a larger number of files containing results of map step). In AWS, d2 is on average faster than d3, d3 has a smaller range while in GCP the opposite holds - D3 is on average faster than d2. AWS outperforms GCP d1 by 11% but underperforms in d2 and d3 by 13.7% and 32.2% respectively.

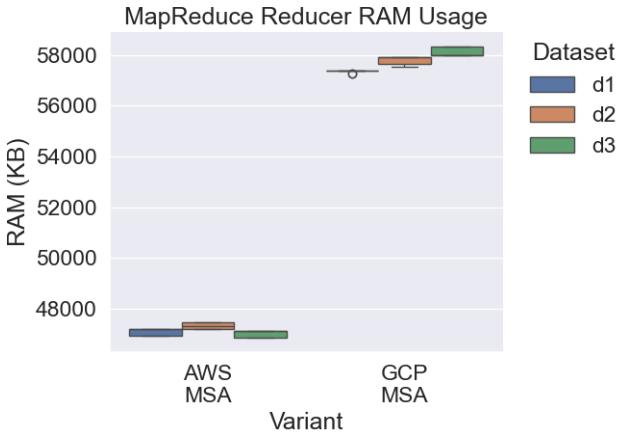

Figure 6.12: Reducer RAM Usage

Figure 6.12 does not add new insights but confirms what has already been seen. GCP requires significantly more RAM than AWS 21.8%, 22.1% and 23.6% on d1, d2 and d3 respec-

tively. Furthermore, Reduce does not load datasets from S3 but both AWS and GCP use similar amounts of RAM compared to Mapper. This suggests that it is the amount of RAM reserved by a serverless function by default is included in the measurement by the default resource library [135] built into Python, perhaps cloud providers have adapted the library to suit serverless functions or buffering is exceptionally good in Mapper. The experiments conducted for MapReduce would benefit greatly if GCP were to release an equivalent reference architecture for a serverless mapreduce in GCP so that MSA can be evaluated against it.

6.4. Serverless ZooKeeper

The original serverless Zookeeper, FaaSKeeper [44], is used as a baseline with an equivalent implementation of FaaSKeeper built with MSA. Two serverless functions Writer and Distributor are examined. Writer exclusively modifies system storage and does all the writing operations of ZooKeeper. Clients register watches on nodes that store data and Distributor is responsible for sending notifications to all Clients when Writer modifies a node. The execution time (seconds) and ram usage (KB) of Writer and Distributor are measured in Figure 6.13. FaaSKeeper Writer is usually tight and consistent in execution time while MSA Writer on average has twice the execution time of FaaSKeeper. However, the range of values observed in both implementations is the same because of long tails in FaaSKeeper Writer. In Distributor the mean of MSA 0.228478s, 88% larger than FaaSKeeper's 0.121204s. However, the MSA tails are significantly larger at the bottom, which means that some invocations of MSA Distributor are faster than the FaaSKeeper's Distributor. MSA underperformed in both Writer and Distributor suggesting that either code synthesis can not compete with Native implementations in latency sensitive serverless applications or that optimizations can be made in the synthesized code for DynamoDB. MSA implementations in Writer and Distributor require 2.5% and 1.6% more RAM than FaaSKeeper. Distributor has tightly bound RAM usage for both implementations but, in Writer, MSA has a wider range than FaaSKeeper. Overall, there are long tails in execution times of all implementations but not in RAM which can be caused by the interactions of the serverless function with DynamoDB. The same conclusions as before for optimizing synthesized code of DynamoDB also holds here.

6.5. Latency

In this section, the experiments measure latency, which is defined as the difference in time at which confirmation for an event is received and the time at which an event is received. In MVCC the event is a transaction of sending an amount from one account to another, while in Zookeeper it is creation of a new node in the file system.

Figure 6.14 shows the latency of MVCC in seconds, and Fig. 6.15 zooms in on AWS. In v1, AWS has the smallest range of values and the lowest average latency. Kubernetes and GCP (Native and MSA) are showing considerable variance with standard deviations of 0.731117, 0.508081 and 0.621536 while for AWS (Native and MSA) it is 0.014367 and 0.011322. In AWS, MSA is slightly better than Native but not in GCP although the latencies of GCP Native and GCP MSA are a whisker away from each other. In v2, AWS is once again the most consistent with the smallest range although Kubernetes shows significant improvement of 78% compared to v1, Kubernetes has the lowest average latency in v2 but it also has a long tail. Compared to v1, AWS Native and AWS MSA are 77% and 69.4% higher) while for GCP Native , GCP MSA and Kubernetes it has reduced in v2 by 17.4% and 28.2%. The standard deviation of latency in

Figure 6.13: Execution time and RAM usage of Writer and Distributor in ZooKeeper

GCP Native fell by 29.8% but increased for GCP MSA by 7.4% compared to v1. The standard deviation of latency in AWS MSA v2 is 230% higher than AWS Native (Fig. 6.15), while for GCP this number is 87% higher, suggesting that MSA can be better equipped to work with message queues.

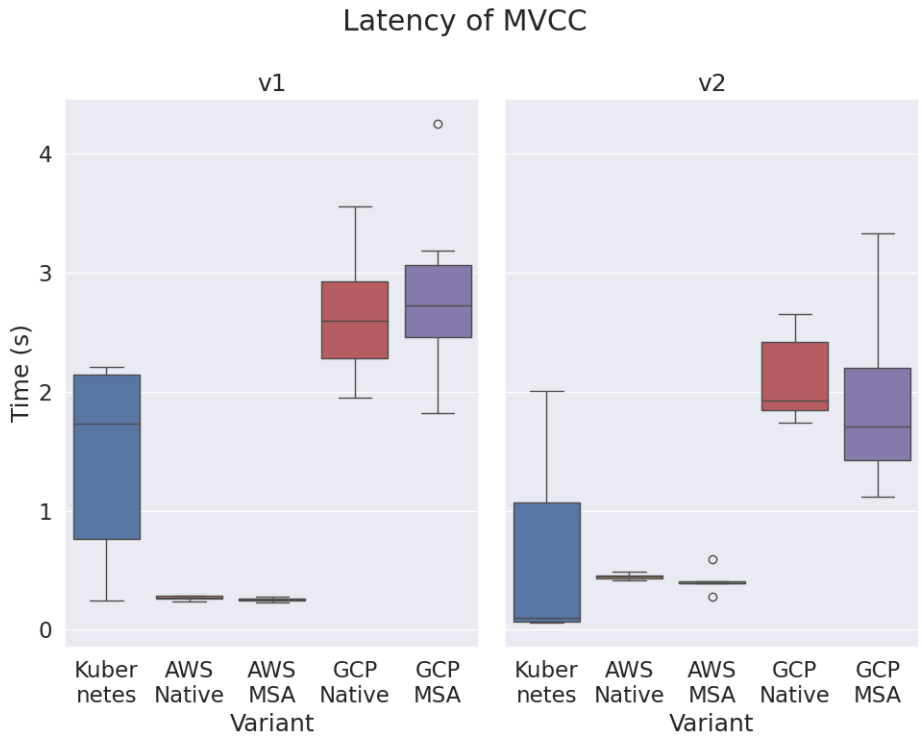

Figure 6.14: Latency in MVCC

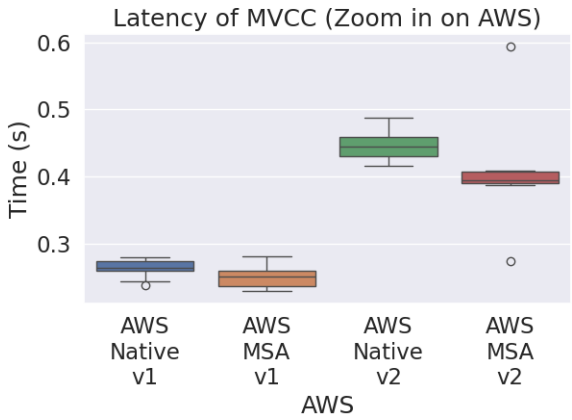

Figure 6.15: Latency in MVCC (Zoom in on AWS)

Figure 6.16 examines the latency in serverless Zookeeper built with MSA and FaaSKeeper. We find that FaaSKeeper outperforms by a substantial amount with a 80% lower standard deviation in latency compared to MSA. On average, FaaSKeeper latencies are half the time in seconds of MSA and does not have a long tail like MSA.

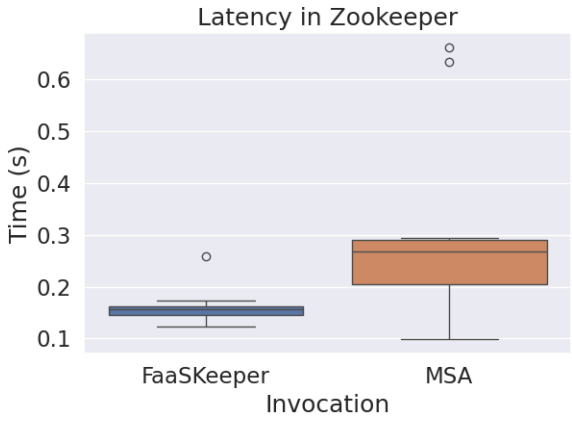

Figure 6.16: Latency in Zookeeper

6.6. Discussion

This chapter started by evaluating five MVCC implementations, one on Kubernetes, one built natively on AWS and GCP, and one built with cloud-agnostic infrastructure code and cloud-agnostic business logic using MSA on AWS and GCP. The experiment had two parts with two different implementations of MVCC, the first with strict consistency (v1) and the second, latency optimized with eventual consistency guarantees (v2). AWS MSA and GCP MSA beat their natively built

counterparts in average execution time in two different serverless functions, in v1 and v2. Kubernetes sometimes was better than MSA, but both serverless implementations on AWS (Native and MSA) had smaller tails and lesser variance than Kubernetes. AWS serverless functions had shorter execution times and lower RAM usage than GCP serverless functions suggesting that AWS may have optimized CRUD operations by a serverless function to a SQL database (AWS RDS) while GCP is yet to offer such benefits to their users. Next a serverless implementation of Zookeeper built with MSA is compared with FaaSKeeper and it was found that FaaSKeeper outperformed MSA in execution time and RAM in all tests. There were frequent interactions with DynamoDB and message queues which are likely to have caused long tails.

We also looked at a partially serverless implementation of Mapreduce by AWS called Refarch against 2 purely serverless implementations on AWS and GCP with MSA. This experiment is different from the previous experiments because it is data centric and revealed that there are situations in which a GCP Cloud Function can run faster than AWS Lambda. We also discovered that in Mapper and Reducer, GCP serverless functions had smaller tails than and outperformed AWS in execution time, but not in RAM usage. A more extensive study with a greater amount of resources is needed to study buffering done by serverless functions when loading data from S3 because RAM usage was roughly the same for all three datasets tested. Both MSA implementations reflected the size of datasets proportional to execution time, but Refarch which is built in AWS did not reflect this trend as it was faster than expected in d2. MSA was unable to beat Refarch in execution time of serverless functions highlighting that data intensive serverless functions are a blind spot of MSA and were not taken into account in the design. A more thorough evaluation of execution times for each statement of code in a serverless function can better explain the execution times and RAM usage and help identify the cause of the observed behaviors.

Lastly, we looked at latencies in MVCC and Zookeeper and found that MSA is better equipped to communicate with databases than Native counterparts and there is room for improvement in publishing messages to a message queue. This is expected because in AWS, messages can be batch processed from a message queue but GCP serverless functions can not do this. Thus, in the design of MSA, AWS Lambda cannot batch process at the moment. Furthermore, if a serverless function needs to interact with a SQL database, it can be advisable for the serverless function to communicate directly with the database in AWS but asynchronously in GCP using a message queue that will trigger another serverless function to interact with a SQL database. In Zookeeper, we find that the code synthesis and abstractions made for DynamoDB can be improved because FaaSKeeper outperformed MSA in all experiments.

In general, AWS almost always outperformed GCP in execution time and RAM usage, sometimes by significant amounts and this is consistent with other previous works [109], [104], [45]. Another insight gained are that serverless components on the cloud offered by various cloud providers offer significantly different performance in execution time and RAM usage, but also latency of operations like publishing a message in a serverless message queues (AWS SQS and GCP Pub/Sub) and CRUD operations in a SQL database (AWS RDS and GCP Cloud SQL). Such operations are vital in time-sensitive applications (more so in serverless functions), and it would logically follow that, similarly to latency, there are probably also other scalability constraints and sweet spots of different services on different clouds. This knowledge will better inform developers in building cloud applications by leveraging the strengths and working around the weaknesses of a cloud and it's services. This highlights that migrating an application from one cloud to another has various easy-to-miss nontrivial nuances that can have considerable

impact on performance of that application and the QoS/SLA the application provides to its users.

7 User Survey

7.1. User Survey

To determine if MSA actually solves the vendor lock-in problem by enabling developers to write cloud-agnostic infrastructure code and business logic, a user survey is conducted. The following paragraphs explain the design and structure of the experiment and survey followed by the results and analysis.

7.1.1. Setup

To determine if MSA meets the objectives of the thesis such as cloud agnosticity, participants need to build a non-trivial serverless application where there are sufficient differences in the native implementations between AWS and GCP so that the utility of MSA can be discovered. It is also important for participants to have previously worked on preferably both AWS and GCP before hand or at least be familiar with the challenges of vendor lock-in specifically on the cloud. Without understanding the problem, it is hard to see how MSA solves the problem, and thereby limits the ability of participants to give useful opinions about MSA in the survey. To this end, participants experienced in building cloud infrastructure and writing cloud native applications on multiple clouds are required. This is because MSA has both components and one of the two can only give an incomplete picture of MSA to the participant. The eligibility conditions complicate finding a sizable number of participants belonging to such a specific niche group of participants. Possible biases are addressed by inviting participants with experience on various clouds and different seniority/years of experience coming from different domains of IT.

To bring out the problem of vendor lock-in and solve it with MSA, participants are asked to build a serverless application with multiple components on AWS or GCP and migrate to the other cloud. Participants examine and test the migrated serverless application on the other cloud to determine if the migration is successful. A migration is successful if the migrated application on the other cloud perform the same tasks as the original without additional side effects. This requires specific prior knowledge and experience with cloud applications to build such a serverless application in a short time and to know what to look for to fairly evaluate MSA.

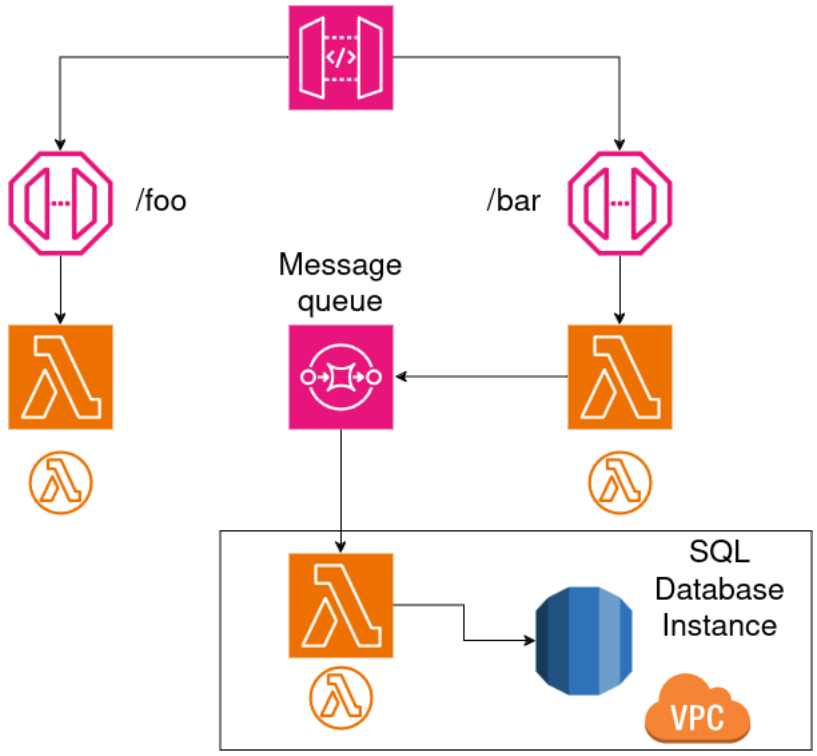

Figure 7.1: Example architecture of serverless application for user survey

7.1.2. The Experiment

Participants are given 45 minutes to build an API with two endpoints, ”/foo” and ”/bar”. Foo is a simple FizzBuzz implementation on a serverless function exposed through an API Gateway, while Bar stores the headers and query parameters received at the API Gateway in a SQL database. A complication for developers is that the SQL database can contain sensitive information, so serverless functions exposed to the public Internet can not have read or write access to the SQL database. Foo is a simple program, and participants are helped to get started and learn their way around MSA. Bar is built without assistance and developers have access to documentation of MSA, AWS and GCP. To save time, participants are also provided with a schema of the SQL table with the code required to setup the database.

A possible implementation of Bar can consist of two serverless functions (Figure 7.1), one that is exposed to the public Internet and the other in the VPC to interact with the SQL database. Function composition can be used where the first serverless function is triggered by an API Gateway endpoint with a POST request. The serverless function processes the headers and query parameters and publishes the computed results to a message queue that will trigger the other serverless function in the VPC. To enable communication between the serverless functions outside and inside the VPC, appropriate IAM roles and permissions are needed in addition to opening a port or some form of hole in the VPC. This must be done delicately because a wide open VPC is not a VPC anymore, so IAM roles should be used to further tighten the VPC. Other

solutions such as setting an API Gateway with authentication inside the VPC accepting traffic at only port 443 are also a valid solution, but is not supported by MSA at the moment. After setting up the infrastructure, developers need to ensure that their business logic can interact with two different flavors of SQL databases (AWS RDS and GCP Cloud SQL) and message queues (AWS SQS and GCP Pub/sub). The last step is to do a migration from one cloud to the other.

7.1.3. Questionnaire

Participants start the survey by answering important questions like the number of years of programming experience, multiple clouds, and monads, followed by more specific questions about MSA, quality of migration, and abstractions. 15 responses were collected for the survey of which 46% and 33% reported extensive and moderate experience with multiple clouds, 33% also reported more than 7 years of programming experience and another 40% with between 3 to 7 years suggesting the participants should have the required technical skills but are also relatively diverse with a healthy mix of seasoned veterans and others at various stages of mastery of cloud technologies. Most of the participants were unfamiliar with monads and therefore were given a brief introduction to monads before starting to build the application with MSA.

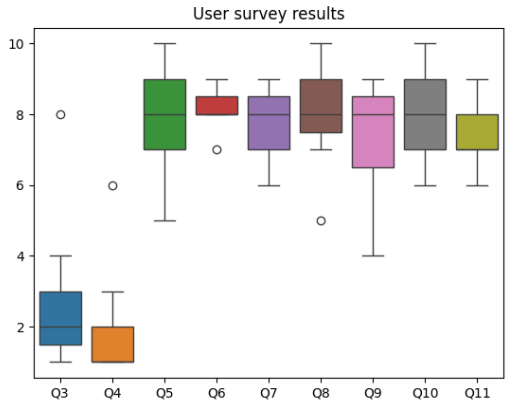

Figure 7.2: User survey results by question

All the following questions are answered by giving an integer score between 1 and 10 where 1 means strongly disagree and 10 means strongly agree. The overall average score of given to MSA is 7.8. Participants strongly agreed (average score 8) that MSA allowed developers to be true to the ideals of serverless programming, that is, a serverless function with Python code on a cloud of choice (AWS or GCP) can be hosted with specific configurations. The participants also reported an average score of 8.2 and 7.73 for abstracting tedious boilerplate code from infrastructure code and business logic. Some concerns raised by participants for the latter were about batch processing on AWS Lambda, and MSA taking a hard position on delineating responsibilities of business logic and infrastructure code. This design decision is at odds with personal preferences for some participants who prefer to dynamically configure infrastructure through business logic and better align with the philosophy of GCP with respect to triggers. De-

Years of programming experience	Experience with multiple clouds
26% (0 to 3)	20% No
40% (3 to 7)	33% (Moderate)
33% (>7)	46% (Extensive)

Table 7.1: Relevant experience of participants

spite this, participants reported an average of 8.13 to satisfactorily migrate cloud infrastructure from one cloud to another and a 7.4 to migrate business logic to another cloud. Participants were also pleased with MSA and gave an 8 for the ability to create components on the cloud like SQL databases, message queues, etc. with specific nontrivial configurations that are unique to a specific cloud through a common infrastructure code. However, ease of use received a 7.27 because MSA forces developers to write infrastructure code and business logic in specific ways, which is restrictive. Such criticism is typical for abstractions, specifically since the participants were already familiar with the various proprietary APIs of the cloud providers, which play a major role in causing the vendor lock-in this thesis aims to solve. Given that participants were unfamiliar with monads, using a design like MSA can feel like a burden because numerous design decisions are informed by monads.

Participants with experience on a single cloud were the least impressed by MSA giving an overall average of 7.1. This is expected because the severe divide between AWS and GCP is arduous to appreciate without first experiencing the problem firsthand. Participants with experience on multiple clouds reported an overall average of 8.36. Participants with greater than 7 years of programming experience (G7) and those with less than 3 years (L3) gave an overall average score of 8 and 7.1. For ease of use of MSA, both G7 and L3 gave lower scores with an average of 7.2 and 6.75; however, both groups were satisfied with the quality of migration of the infrastructure code (8.8 and 8.5) and the business logic (7.25 and 6). An interesting convergence in opinions between the two groups is that G7 reports an average of 7.8 and L3 7.75 for hosting a serverless function with Python code on a cloud of choice (AWS or GCP) with specific configurations. This suggests both groups are satisfied with the ability to create serverless functions with specific configurations but there challenging elements to MSA. It is possible that this opinion will change if the participants have more time to build more complex serverless functions, which will give MSA a better opportunity to show its strengths.

The difference in average score per question between participants with extensive and moderate experience with multiple clouds is always within 0.5 with the former usually being higher. Perhaps the tasks were too simple for these two groups. Lastly, in Figure 7.2 there are tails up an down in most questions suggesting that either more work needs to be done in explaining the problem and the solution or that there is a variety of opinions on the topics discussed suggesting that there are more subtle complexities and nuances that require a closer look.

Number	Question
1	How many years of programming experience do you have?
2	What is experience with multiple clouds (AWS, GCP, Azure, etc.)?
3	How familiar are you with the concept of monads from functional programming? (1: Not at all, 10: Know it like the back of my hand).
4	Have you used monads when you write code? (1: Never, 10: All the time).
5	Does this tool allow you to ”host any code, where ever you want, the way you want”? (Is this tool true to the ideals of serverless programming?)
6	Is boilerplate code successfully abstracted away from business logic in writing Infrastructure as Code? (Do monads abstract away the tedious boring code).
7	Is boilerplate code successfully abstracted away from business logic in writing code in the serverless function? (Do monads abstract away the tedious boring code).
8	Does MSA satisfactorily migrate cloud infrastructure to another cloud?
9	Does MSA satisfactorily migrate business logic in serverless function to another cloud?
10	Are you able to create other components like SQL database, Message Queue, etc. with configurations unique to a cloud with common code?
11	How easy/convenient to use is it?

Table 7.2: Questions asked in the Survey

8

Discussion and Future Work

In this thesis, we have introduced MSA - Monadic approach to Serverless Applications that takes inspiration from Monads to solve vendor lock-in on public clouds like AWS and GCP. In chapter 3 numerous previous attempts at solving this problem are listed and as of the time of writing this paper, no attempt as succeeded. All solutions solve either the agnostic cloud infrastructure problem or the cloud agnostic business logic problem, but none have created a complete and performant solution to solve both problems simultaneously. The monadic idea of encapsulating tedious polluting boilerplate code that is specific for a cloud provider behind a wrapper combined with Infrastructure as Code provides a potent tool for enabling developers to write pure and efficient business logic code to build performant cloud agnostic cloud native serverless applications. MSA has been designed to enable developers to build complex serverless applications that remain true to the ideals of serverless computing such as cost effective and near-instantaneous scalability with negligible maintenance. Section 8.1 answers the research questions in Section 1.1 with which this thesis started. Section 8.2 concludes the thesis with possibilities for future work.

8.1. Discussion

1. **Monadic Approach:** *Can monads help the user explicitly declare requirements for their serverless functions without polluting/compromising functional nature of code?*

   Yes. Section 4.2 shows an example and explains the various aspects in which monads are used to enable developers to declare requirements for their serverless functions in the monadic style. Layers of abstractions are used in Cloud Monads so that there is no loss in functionality and the quality of code is maintained. Chapter 5 describes more complex programs built in the same style that always adhere to the monadic principles. Further, by unifying the infrastructure code and business logic through layers of abstractions, developers do not need to learn or use specific code of a particular cloud because the specifics and other ugly details such as discrepancies between various clouds are abstracted away. MSA does not take away freedoms of developers by allowing usage of specific code of a cloud provider and but then MSA is not able to guarantee cloud agnosticity.

2. **Cloud Agnostic:** *Can the monadic approach be applied to Infrastructure as Code to provision cloud agnostic infrastructure*

   Yes. In sections 2.1.1 and 2.1.2 we highlight the various differences in configurations, ecosystems, and product offerings between the public clouds, and in chapter 4 we discuss how these differences are solved in infrastructure code by taking inspiration from monads. For instance, an event in S3 can directly trigger a Lambda in AWS however in GCP an event in Cloud Storage can only be published in a Pub/Sub message queue. This is severe difference between infrastructure code in AWS and GCP. The respective Cloud Monads solve this by creating the required infrastructure and providing an equivalent trigger mechanism in GCP as AWS already has to enable writing cloud agnostic infrastructure code. Chapter 5 provides evidence for using common generic infrastructure code to build complex cloud agnostic infrastructures on AWS and GCP without rewriting any code. The cloud agnostic infrastructure code guarantees can further be extended in a similar way to other clouds such as Azure, Oracle, IBM, etc. Lastly, the Cloud Monad also allows developers to state conditions and restrictions on their dependency and MSA ensures the conditions are validated and restrictions are actively obeyed.

3. **Clean Packaging:** *Can code synthesis be a standardized solution to write generic cloud-agnostic business logic which can be run in the serverless function of choice?*

   Yes. Section 2.3 shows the different requirements business logic code must satisfy to be run on different serverless functions in AWS, GCP and Azure. Although all three clouds make similar claims and give developers similar configurations on the surface, the devil is in the details. Two possible solutions to enable developers to write cloud agnostic business logic to are discussed, the first is to build a common framework or library for developers to code against, but that can means extra lines of code in dependencies that are unrelated to the program. The other solution is to synthesize code on the fly so that only the required code is synthesized and the serverless function is not bloated with irrelevant dependencies. This keeps the serverless function light and clean besides smoothly integrating with the monadic concept of asking for requirements to be explicitly stating upfront.

   Section 4.3 describes how developers can state the interactions that a serverless function will have with other cloud services and components. Code synthesis will then ensure that the specific things that are required are synthesized in a clean modular approach that does not burden developers with tiresome repetitive code. Prebuilt templates store the required code for every situation such as triggering a serverless function with HTTP(S) or a message queue. The code synthesis stitches together the required templates so deliver exactly what is required in the serverless function. The benefit of synthesis is that developers can code against a common API that will work on all supported clouds, but the drawback lies in the fact that a template for every cloud interaction needs to be built, maintained and it can add extra lines of code to the serverless function compared to a vendor specific cloud native approach that usually causes vendor lock-in.

4. **How do serverless applications built with a monadic and cloud agnostic approach compare to serverless applications built with the traditional approach?**

   Chapter 6 benchmarks and evaluates the three complex cloud-agnostic serverless applications built with MSA. The first application MVCC is a non-locking scaleable distributed

database, MapReduce and ZooKeeper are data intensive and latency sensitive serverless applications respectively. In the experiments it is found that serverless applications built with MSA often beat alternatives in execution time and provide comparable performance in ram usage along with the additional benefit of freedom from vendor lock-in by cloud providers.

In MVCC, AWS MSA and GCP MSA are benchmarked against their native implementations in respective clouds and also an equivalent architecture in Kubernetes. Kubernetes is selected because it is a widely used tool of choice. In v1 of MVCC, MSA outperformed native implementations in Worker in both execution times and RAM usage of serverless functions. MSA was usually ahead in Control too but there are some instances in GCP where MSA required more RAM and execution time than native. GCP MSA was always significantly behind Kubernetes in all experiments but AWS MSA often outperformed Kubernetes. A latency optimized variant (v2) of MVCC is also made that asynchronously writes transactions to the databases. V2 improved the latencies in Kubernetes and GCP but not AWS suggesting that AWS has optimized writing to a RDS whereas GCP optimized publishing a message on a message queue. A closer look is required to confirm this.

The ZooKeeper experiment benchmarked MSA against a serverless ZooKeeper called FaaSKeeper. MSA was on average unable to beat FaaSKeeper in all experiments which means there are areas of improvements in MSA in latency sensitive applications like ZooKeeper. Some samples of MSA did outperform FaaSKeeper and they could be chance or luck but it is hard to say given the enormous complexities and unknown scheduling algorithms of AWS data centers.

The last application is a serverless MapReduce in which AWS's reference architecture (Refarch) is modernized and used as a benchmark. AWS MSA was slower than Refarch in execution times of Mapper but had long tails and some samples were competitive with Refarch. Perhaps minor tweaking in MSA can fix this. In a first, GCP outperformed AWS in execution time on the first dataset of Mapper and in second and third datasets of Reducer. GCP however required significantly higher RAM than AWS in all experiments suggesting that GCP Cloud Functions trade off RAM for execution time in data intensive tasks. The datasets however are not a wide spectrum in size and complexity so perhaps AWS makes tradeoffs that show results at different sizes of datasets.

AWS MSA outperformed AWS Native in latency on all MVCC experiments however in GCP MSA beat GCP Native only in v2. In time sensitive applications like ZooKeeper, MSA was unable to keep up with FaaSKeeper. Serverless applications built with MSA in many cases outperform natively built serverless applications in execution time and RAM usage (Section 6.6) and not only does MSA aid developers in escaping the clutches of vendor lock-in, but also improves code quality by enabling developers to write clean functional infrastructure code and business logic. Thus, developers benefit in various ways, sometimes without drawbacks.

5. **Is MSA easy to use?**

   The purpose of the survey is to determine MSA solves the problem of vendor lock-in by enabling developers to write cloud-agnostic infrastructure code and business logic. 15 participants are asked to build a nontrivial serverless application where there are sufficient differences in the native implementations between AWS and GCP so that the utility of MSA

can be discovered. A survey filled out by the participants after building the serverless application asked questions about building specific and customizable serverless functions, quality of migration in infrastructure and business logic, ease of use of MSA among others. The overall average score on scale of 1 (strongly disagree) to 10 (strongly agree) is 7.8 which is slightly lower than expected but acceptable given the challenges in conducting the survey such as recruiting participants with the required experiences (monads and experience with multiple clouds) and conducting the survey such that the problem of vendor lock-in is brought out, utility of MSA is brought out and there is time to reflect on what MSA does in a limited amount of time and resources.

Participants with experience on multiple clouds reported an average overall score of 8.36 while participants without the experience reported an average overall score of 7.1 showing a gap in information about the vendor lock-in challenges or the difficulties in explaining MSA as almost all participants were unfamiliar with monads. It is possible that the monadic aspect of MSA was complex and hard to evaluate for participants given their reported existing knowledge of monads. Abstractions are seen as a burden when a developer only knows one proprietary API and does not understand the need for the abstraction, which makes it hard to understand the utility of the abstraction. It is also possible that serverless application made by participants was trivial which robbed MSA of an opportunity to truly show what it can do.

Participants strongly agreed with a score of 8 that MSA allowed developers to stay true to the ideals of serverless programming, 8.2 and 7.73 for abstracting away tedious mundane boilerplate code from infrastructure code and business logic, 8.13 and 7.4 for migrating infrastructure code and business logic from one cloud to another (AWS and GCP) and a 8 for creating other cloud services and components such as message queues, SQL databases and so on with specific configurations that can be problematic in cloud agnostic settings. Lastly, ease of use of MSA scored 7.27 possibly because participants found monads and related aspects cumbersome.

8.2. Future Work

In this thesis, we presented MSA to build serverless cloud-agnostic applications on AWS and GCP which are automated and migrations can be made from one cloud to another simply by pressing a button. There are several possible improvements that are logical next steps for MSA or outside the scope of this thesis due to time or financial constraints.

1. **Data migration:** Serverless functions are stateless and hence cloud applications depend heavily on databases for many things like storing state or passing information from serverless function to another. MSA does not migrate data in a database from the cloud to another with the serverless application. The data is instead deleted to reduce costs by cleaning up the cloud environment from which the migration is done from. This is sufficient for this thesis but impractical for real life use cases.

2. **Security:** MSA allows developers to build highly customizable IAM roles to grant or deny access of cloud components to other resources and cloud components however it is not sufficiently fine grained to be used in production environments. Ideally, the IAM roles should be built in code dynamically instead of statically in json files to plug the holes

and grant tighter accesses and permissions to cloud components and resources. Another perspective of security is in networking, where, while VPC is a good idea, opening of ports or piercing the VPC with specific services can be done better and more securely. Lastly, it is vital to protect data in databases with encrypting at rest and in movement, frequent rotations of keys and regular backups. Lastly, serverless functions are exposed through an API Gateway on AWS and GCP however they do not offer sufficient protection against DDoS attacks and meet other requirements for securing and protecting resources required for critical infrastructure. [134] notes that security in Infrastructure as Code requires more attention to better protect cloud environments and developers/consumers/organizations using cloud technologies from malicious actors.

3. **Edge compute:** Edge computing has been picking up steam and can prove to be a game changer with other concepts like distributed machine learning, IoT and many more. Serverless applications can leverage the edge to further minimize latency in applications.

4. **IfC:** MSA did not explore the possibility of generating infrastructure code from the business logic. This will further simplify building serverless applications and combining Monads with the latest developments in AI can be promising.

5. **Cross cloud:** MSA currently allows developers to build a serverless application on a single cloud at a time. In this thesis, several instances were observed where particular services of a cloud are better than competitors. It could be interesting for developers to build a cross cloud serverless application that leverage the best tools and services of every cloud.

Bibliography

[1] *{m}brace the cloud*. url: http://mbrace.io/.

[2] Alexandru Agache et al. “Firecracker: Lightweight virtualization for serverless applications”. In: *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 2020, pp. 419–434.

[3] Udit Agarwal. “Cloud Abstraction Libraries: Implementation and Comparison”. In: (2016).

[4] *All about Monads*. url: https://wiki.haskell.org/All_About_Monads.

[5] *Amazon API Gateway*. url: https://aws.amazon.com/api-gateway/.

[6] *Amazon CloudWatch now supports high resolution metric extraction from structured logs*. url: https://aws.amazon.com/about-aws/whats-new/2023/02/amazon-cloudwatch-high-resolution-metric-extraction-structured-logs/.

[7] *Amazon Relational Database Service*. url: https://aws.amazon.com/rds/.

[8] *Amazon Simple Queue Service*. url: https://aws.amazon.com/sqs/.

[9] *Amplab, UC Berkeley*. url: https://amplab.cs.berkeley.edu/.

[10] *Apache Airflow*. url: https://airflow.apache.org/.

[11] *Apache Deltacloud*. url: https://deltacloud.apache.org/.

[12] *Apache jclouds*. url: https://jclouds.apache.org/.

[13] *Apache Libcloud*. url: https://libcloud.apache.org.

[14] *API Gateway*. url: https://cloud.google.com/api-gateway.

[15] *Are Webjobs depricated?* url: https://learn.microsoft.com/en-us/answers/questions/1117332/are-webjobs-depricated.

[16] Itzhak Aviv et al. “Infrastructure from code: The next generation of cloud lifecycle automation”. In: *IEEE Software* 40.1 (2022), pp. 42–49.

[17] *AWS Boto3 Github Repository*. url: https://github.com/boto/boto3/.

[18] *AWS Glue*. url: https://aws.amazon.com/glue/.

[19] *AWS Lambda: Resilience under-the-hood*. url: https://aws.amazon.com/blogs/compute/aws-lambda-resilience-under-the-hood/.

[20] *AWS Lex*. url: https://aws.amazon.com/lex/.

[21] *AWS OTEL Lambda*. url: https://github.com/aws-observability/aws-otel-lambda.

[22] *AWS Secrets Manager*. url: https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html.

[23] *AWS Serverless Application Model*. url: https://aws.amazon.com/serverless/sam/.

[24] *AWS X-Ray*. url: https://aws.amazon.com/xray/.

[25] *AWS X-Ray SDK for Python*. url: https://github.com/aws/aws-xray-sdk-python/#user-content-trace-threadpoolexecutor.

[26] *aws.s3.BucketNotification Supporting Types*. url: https://www.pulumi.com/registry/packages/aws/api-docs/s3/bucketnotification/#supporting-types.

[27] *Azure Functions hosting options*. url: https://cloud.google.com/functions/docs/configuring/timeout.

[28] *Azure Functions hosting options*. url: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#timeout.

[29] *Azure Functions overview*. url: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp.

[30] *Azure Functions runtime versions overview*. url: https://learn.microsoft.com/en-us/azure/azure-functions/functions-versions.

[31] *Azure Python Functions Github Repository*. url: https://github.com/Azure/azure-functions-python-library.

[32] *Build Backends. Fast.* url: https://www.shuttle.rs/.

[33] *Build modern full-stack applications on AWS*. url: https://sst.dev/.

[34] *Caching data and configuration settings with AWS Lambda extensions*. url: https://aws.amazon.com/blogs/compute/caching-data-and-configuration-settings-with-aws-lambda-extensions/.

[35] *Can I limit concurrent invocations of an AWS Lambda?* url: https://stackoverflow.com/questions/42028897/can-i-limit-concurrent-invocations-of-an-aws-lambda.

[36] José Manuel Ortega Candel et al. “Cloud vs Serverless Computing: A Security Point of View”. In: *International Conference on Ubiquitous Computing and Ambient Intelligence*. Springer. 2022, pp. 1098–1109.

[37] *Configure Cloud Functions*. url: https://cloud.google.com/functions/docs/configuring.

[38] *Configuring advanced logging controls for your Lambda function*. url: https://docs.aws.amazon.com/lambda/latest/dg/monitoring-cloudwatchlogs.html#monitoring-cloudwatchlogs-advanced.

[39] *Configuring Lambda function options*. url: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-timeout-console.

[40] *Configuring Lambda function options*. url: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html.

[41] *Configuring the monitoring agent*. url: https://www.ibm.com/docs/en/mon-diag-tools?topic=application-configuring-monitoring-agent.

[42] *Connect everything. Build anything.* url: https://www.netlify.com/.

[43] *Contributors to AWS Lambda container cold starts*. url: https://nitric.io/blog/lambda-container-coldstarts.

[44] Marcin Copik et al. “FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example”. In: *arXiv preprint arXiv:2203.14859* (2022).

[45] Marcin Copik et al. “Sebs: A serverless benchmark suite for function-as-a-service computing”. In: *Proceedings of the 22nd International Middleware Conference*. 2021, pp. 64–78.

[46] *Create your first containerized functions on Azure Container Apps*. url: https://learn.microsoft.com/en-us/azure/azure-functions/functions-deploy-container-apps?tabs=acr%2Cbash&pivots=programming-language-csharp.

[47] *CreateEventSourceMapping*. url: https://docs.aws.amazon.com/lambda/latest/api/API_CreateEventSourceMapping.html.

[48] *Customize the Cloud Functions build process*. url: https://cloud.google.com/function/docs/building#view_your_build_image_logs.

[49] *Darklang: just code*. url: https://darklang.com/.

[50] *Dasein Cloud*. url: https://jclouds.apache.org/.

[51] *Data.Either: The Either type and associated data operations*. url: https://hackage.haskell.org/package/base-4.19.1.0/docs/Data-Either.html.

[52] Ricardo Ramos De Oliveira, Rafael Messias Martins, and Adenilso Da Silva Simao. “Impact of the vendor lock-in problem on testing as a service (TaaS)”. In: *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2017, pp. 190–196.

[53] *Deploy Lambda functions with container images*. url: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/deploy-lambda-functions-with-container-images.html.

[54] *Deploy on Kubernetes with Docker Desktop*. url: https://docs.docker.com/desktop/kubernetes/.

[55] Simon Eismann et al. “The state of serverless applications: Collection, characterization, and community consensus”. In: *IEEE Transactions on Software Engineering* 48.10 (2021), pp. 4152–4166.

[56] *Event notification types and destinations*. url: https://docs.aws.amazon.com/AmazonS3/latest/userguide/notification-how-to-event-types-and-destinations.html#supported-notification-event-types.

[57] *Event Source Data Classes*. url: https://docs.powertools.aws.dev/lambda/python/latest/utilities/data_classes/.

[58] Jelle Eysbach. “Cloud Monads: A novel concept for monadic abstraction over state in serverless cloud applications”. In: (2022).

[59] *Field Notes: Monitoring the Java Virtual Machine Garbage Collection on AWS Lambda*. url: https://aws.amazon.com/blogs/architecture/field-notes-monitoring-the-java-virtual-machine-garbage-collection-on-aws-lambda/.

[60] Ismael Figueroa, Paul Leger, and Hiroaki Fukuda. “Which monads Haskell developers use: An exploratory study”. In: *Science of Computer Programming* 201 (2021), p. 102523.

[61] *Firecracker – Lightweight Virtualization for Serverless Computing*. url: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.

[62] *Fissiom*. url: https://fission.io/.

[63] *Focus on your application, and leave the database to us*. url: https://cloud.google.com/sql/?hl=en.

[64] *From code to fully-managed AWS workloads in seconds*. url: https://getampt.com/.

[65] *Function as a Service*. url: https://en.wikipedia.org/wiki/Function_as_a_service.

[66] *Function timeout*. url: https://cloud.google.com/functions/docs/configuring/timeout.

[67] *Functions*. url: https://learn.microsoft.com/en-us/azure/architecture/gcp-professional/services#functions.

[68] *GCP Functions Framework Github Repository*. url: https://github.com/GoogleCloudPlatform/functions-framework-python.

[69] *GCP implementation*. url: https://github.com/spcl/faaskeeper/pull/41.

[70] *GCP Workflows*. url: https://cloud.google.com/workflows?hl=en.

[71] *gcp.sql.User*. url: https://www.pulumi.com/registry/packages/gcp/api-docs/sql/user/.

[72] *gcp.storage.Notification Inputs*. url: https://www.pulumi.com/registry/packages/gcp/api-docs/storage/notification/#inputs.

[73] *Get started with Pulumi policy as code*. url: https://www.pulumi.com/docs/using-pulumi/crossguard/get-started/.

[74] Fotis Gonidis et al. “Cloud application portability: an initial view”. In: *Proceedings of the 6th Balkan Conference in Informatics*. 2013, pp. 275–282.

[75] Jasper A Hasenoot, Jan S Rellermeyer, and Alexandru Uta. “The Performance of Distributed Applications: A Traffic Shaping Perspective”. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 207–220.

[76] Joseph M Hellerstein et al. “Serverless computing: One step forward, two steps back”. In: *arXiv preprint arXiv:1812.03651* (2018).

[77] Tony Hoare and Robin Milner. “Grand challenges for computing research”. In: *The Computer Journal* 48.1 (2005), pp. 49–52.

[78] Sanghyun Hong et al. “Go serverless: Securing cloud via serverless design patterns”. In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. 2018.

[79] *How to configure monitoring for Azure Functions*. url: https://learn.microsoft.com/en-us/azure/azure-functions/configure-monitoring?tabs=v2#enable-application-insights-integration.

[80] *How to develop and test your Cloud Functions locally*. url: https://cloud.google.com/blog/topics/developers-practitioners/how-to-develop-and-test-your-cloud-functions-locally.

[81] *HTTP triggers*. url: https://cloud.google.com/functions/docs/calling/http.

[82] Patrick Hunt et al. “{ZooKeeper}: Wait-free coordination for internet-scale systems”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010.

[83] *Infrastructure as code*. url: https://en.wikipedia.org/wiki/Infrastructure_as_code.

[84] *Init Containers*. url: https://kubernetes.io/docs/concepts/workloads/pods/init-containers/.

[85] *Java Profiler for Azure Monitor Application Insights*. url: https://learn.microsoft.com/en-us/azure/azure-monitor/app/java-standalone-profiler.

[86] Eric Jonas et al. “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019).

[87] *JVM*. url: https://cloud.google.com/monitoring/agent/ops-agent/third-party/jvm.

[88] *Klotho*. url: https://klo.dev/docs/.

[89] *Klotho: develop for local, deploy for the cloud*. url: https://github.com/KlothoPlatform/Klotho.

[90] *KNative*. url: https://knative.dev/docs/.

[91] Hirohiko Kuramata, Mitsuo Okano, and Hikaru Obata. “Possible Solutions for the Vendor Lock-In Control Protocol in IP-Based Production Systems”. In: *SMPTE Motion Imaging Journal* 132.5 (2023), pp. 38–45.

[92] *Lambda execution environment*. url: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html#runtimes-lifecycle-ib.

[93] *Lambda function URLs*. url: https://docs.aws.amazon.com/lambda/latest/dg/lambda-urls.html.

[94] *Lambda runtimes*. url: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html.

[95] *Lambda runtimes*. url: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html.

[96] *Lambda Telemetry API*. url: https://docs.aws.amazon.com/lambda/latest/dg/telemetry-api.html.

[97] Xing Li, Xue Leng, and Yan Chen. “Securing serverless computing: Challenges, solutions, and opportunities”. In: *IEEE Network* (2022).

[98] Wes Lloyd et al. “Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, pp. 195–200.

[99] *Local development, testing, and debugging of serverless applications defined in Terraform using AWS SAM CLI*. url: https://stackoverflow.com/collectives/aws/articles/75585198/local-development-testing-and-debugging-of-serverless-applications-defined-in.

[100] *Logger*. url: https://docs.powertools.aws.dev/lambda/python/latest/core/logger/.

[101] *Logger: Bring your own Formatter*. url: https://docs.powertools.aws.dev/lambda/python/latest/core/logger/#bring-your-own-formatter.

[102] *Logger: LambdaPowertoolsFormatter*. url: https://docs.powertools.aws.dev/lambda/python/latest/core/logger/#lambdapowertoolsformatter.

[103] *Looking for an explanation of function composition*. url: https://stackoverflow.com/a/1475927/12555857.

[104] Pascal Maissen et al. “Faasdom: A benchmark suite for serverless computing”. In: *Proceedings of the 14th ACM international conference on distributed and event-based systems*. 2020, pp. 73–84.

[105] *Manage your function app*. url: https://learn.microsoft.com/en-us/azure/azure-functions/functions-how-to-use-azure-function-app-settings?tabs=portal.

[106] Eduard Marin, Diego Perino, and Roberto Di Pietro. “Serverless computing: a security perspective”. In: *Journal of Cloud Computing* 11.1 (2022), pp. 1–12.

[107] *MBrace on AWS*. url: https://github.com/mbraceproject/MBrace.AWS.

[108] *MBrace on Azure*. url: https://github.com/mbraceproject/MBrace.Azure.

[109] Garrett McGrath and Paul R Brenner. “Serverless computing: Design, implementation, and performance”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410.

[110] *Metrics*. url: https://docs.powertools.aws.dev/lambda/python/latest/core/metrics/.

[111] *Middleware factory*. url: https://docs.powertools.aws.dev/lambda/python/latest/utilities/middleware_factory/.

[112] Di Mo et al. “Addressing Serverless Computing Vendor Lock-In through Cloud Service Abstraction”. In: ().

[113] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. “An Evaluation of Open Source Serverless Computing Frameworks.” In: *CloudCom* 2018 (2018), pp. 115–120.

[114] *Multiversion concurrency control*. url: https://en.wikipedia.org/wiki/Multiversion_concurrency_control.

[115] *mvcc-db*. url: https://github.com/rkochar/mvcc-db/.

[116] *Nitric multi-cloud demo for PulumiUP 2023*. url: https://github.com/nitrictech/nitric-multi-cloud-demo.

[117] Justice Opara-Martins, Reza Sahandi, and Feng Tian. “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective”. In: *Journal of Cloud Computing* 5 (2016), pp. 1–18.

[118] *OpenFaaS*. url: https://www.openfaas.com/.

[119] *OpenTelemetry Lambda*. url: https://baselime.io/.

[120] *OpenTelemetry Lambda*. url: https://github.com/open-telemetry/opentelemetry-lambda.

[121] *OpenTelemetry Trace 1.0 is now available*. url: https://cloud.google.com/blog/products/operations/opentelemetry-specification-enables-standardized-tracing.

[122] *OpenWhisk*. url: https://openwhisk.apache.org/.

[123] *Parser*. url: https://docs.powertools.aws.dev/lambda/python/latest/utilities/parser.

[124] Dana Petcu. "Portability and interoperability between clouds: challenges and case study". In: *Towards a Service-Based Internet: 4th European Conference, ServiceWave 2011, Poznan, Poland, October 26-28, 2011. Proceedings 4*. Springer. 2011, pp. 62–74.

[125] *pkgcloud*. url: https://github.com/pkgcloud/pkgcloud.

[126] *Pub/Sub*. url: https://cloud.google.com/pubsub/?hl=en.

[127] *Pulumi*. url: https://www.pulumi.com/.

[128] *Pulumi & Continuous delivery*. url: https://www.pulumi.com/docs/using-pulumi/continuous-delivery/.

[129] *Pulumi Cloud*. url: https://github.com/pulumi/pulumi-cloud.

[130] *Pulumi ESC*. url: https://www.pulumi.com/product/esc/.

[131] *Pulumi vs. Nitric*. url: https://nitric.io/docs/faq/comparison/pulumi.

[132] *Pydantic*. url: https://docs.pydantic.dev/latest/.

[133] *Python Client for Google Cloud Storage*. url: https://cloud.google.com/python/docs/reference/storage/latest.

[134] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. "A systematic mapping study of infrastructure as code research". In: *Information and Software Technology* 108 (2019), pp. 65–77.

[135] *Resource*. url: https://docs.python.org/3/library/resource.html.

[136] *Retrieving parameters and secrets with Powertools for AWS Lambda (TypeScript)*. url: https://aws.amazon.com/blogs/compute/retrieving-parameters-and-secrets-with-powertools-for-aws-lambda-typescript/.

[137] Michael Roberts and John Chapin. *What is Serverless?* O'Reilly Media, Incorporated, 2017.

[138] Pedro Rodrigues, Filipe Freitas, and José Simão. "QuickFaaS: Providing Portability and Interoperability between FaaS Platforms". In: *Future Internet* 14.12 (2022), p. 360.

[139] *Runtime Support*. url: https://cloud.google.com/functions/docs/runtime-support.

[140] *S3 Client*. url: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html.

[141] *Serverless*. url: https://glossary.cncf.io/serverless/.

[142] *Serverless Application Model*. url: https://aws.amazon.com/serverless/sam/.

[143] *Serverless computing*. url: https://en.wikipedia.org/wiki/Serverless_computing/.

[144] *Serverless computing*. url: https://learn.microsoft.com/en-us/azure/architecture/aws-professional/services#serverless-computing.

[145] *Serverless Framework*. url: https://www.serverless.com/framework.

[146] *Serverless Reference Architecture: MapReduce*. url: https://github.com/awslabs/lambda-refarch-mapreduce.

[147] Mazen Shawosh and Nicholas Berente. “Software development outsourcing, asset specificity, and vendor lock-in”. In: (2019).

[148] *Simplifying serverless best practices with Lambda Powertools*. url: https://aws.amazon.com/blogs/opensource/simplifying-serverless-best-practices-with-lambda-powertools/.

[149] *Spanner*. url: https://cloud.google.com/spanner/.

[150] *Spanner DB Functions and Triggers*. url: https://www.googlecloudcommunity.com/gc/Databases/Spanner-DB-Functions-and-Triggers/m-p/174216.

[151] *Spark on Google Cloud*. url: https://cloud.google.com/solutions/spark.

[152] Josef Spillner. “Transformation of python applications into function-as-a-service deployments”. In: *arXiv preprint arXiv:1705.08169* (2017).

[153] *StackOverflow: HTTP middleware and google cloud functions*. url: https://stackoverflow.com/a/68353981/12555857.

[154] *Status of Python versions*. url: https://devguide.python.org/versions/.

[155] *Step Functions*. url: https://docs.aws.amazon.com/step-functions/.

[156] Davide Taibi, Josef Spillner, and Konrad Wawruch. “Serverless computing-where are we now, and where are we heading?” In: *IEEE software* 38.1 (2020), pp. 25–31.

[157] *Terraform*. url: https://www.terraform.io/.

[158] *Terraform vs. Nitric*. url: https://nitric.io/docs/faq/comparison/terraform.

[159] *The cloud aware application framework*. url: https://nitric.io/.

[160] *The Current State of Infrastructure From Code*. url: https://www.readysetcloud.io/blog/allen.helton/infrastructure-from-code-benchmark/.

[161] *The serverless platform for data teams*. url: https://modal.com/.

[162] *The stylish Node.js middleware engine for AWS Lambda*. url: https://middy.js.org/.

[163] *Tracer*. url: https://docs.powertools.aws.dev/lambda/python/latest/core/tracer/.

[164] *Unpacking Observability: Understanding Logs, Events, Traces, and Spans*. url: https://medium.com/dzerolabs/observability-journey-understanding-logs-events-traces-and-spans-836524d63172.

[165] *Using AWS Lambda with Amazon Lex*. url: https://docs.aws.amazon.com/lambda/latest/dg/services-lex.html.

[166] *Validation*. url: https://docs.powertools.aws.dev/lambda/python/latest/utilities/validation/.

[167] *Vercel is the Frontend Cloud. Build, scale, and secure a faster, personalized web.* url: https://vercel.com/.

[168] *What is OpenTelemetry?* url: https://opentelemetry.io/docs/what-is-opentelemetry/.

[169] *What is serverless?* 2022. url: https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless.

[170] *What is the Azure equivalent of AWS Lambda?* url: https://stackoverflow.com/a/36341710/12555857.

[171] *Why do we need monads?* url: https://stackoverflow.com/a/28139260/12555857.

[172] *Working with Lambda container images*. url: https://docs.aws.amazon.com/lambda/latest/dg/images-create.html.

[173] *Working with Lambda layers*. url: https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html.

[174] Vladimir Yussupov et al. “Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 2019, pp. 273–283.

[175] Vladimir Yussupov et al. “SEAPORT: Assessing the Portability of Serverless Applications.” In: *CLOSER*. 2020, pp. 456–467.

[176] Alex van der Zeeuw, Alexander JAM van Deursen, and Giedo Jansen. “The orchestrated digital inequalities of the IoT: How vendor lock-in hinders and playfulness creates IoT benefits in every life”. In: *new media & society* (2022), p. 14614448221138075.

[177] Haidong Zhao. “Managing Vendor Lock-in in Serverless Edge-to-Cloud Computing from the Client Side”. PhD thesis. Technical University of Berlin, 2022.

[178] *ZooKeeper design goals*. url: https://zookeeper.apache.org/doc/current/zookeeperOver.html.