IMPLEMENTATION OF SERVERLESS COMPUTING WITH DSL FRAMEWORK IN SWIFT

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering, Master's thesis

2022

Aleksei Sapitskii

Examiner(s): Professor Kari Smolander

M. Sc. Muhhamad Hamza

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Aleksei Sapitskii

Implementation of Serverless Computing with DSL Framework in Swift

Master's thesis

2022

46 pages, 27 figures, 1 tables and 2 appendices

Examiner(s): Professor Kari Smolander and Muhhamad Hamza

Keywords: serverless, DSL, Swift

This work involves the study of existing solutions in the field of creating Serverless infrastructure and the analysis of existing solutions. Thesis proposes a new DSL tool that makes use of code generation techniques for serverless infrastructure. Tool aims to solve the problem of cloud provider lock-in and provide a more reliable way to create and manage infrastructure configurations. Tool implementation is based on code generation techniques for creating all the necessary infrastructure components and operates with virtualized AST of the DSL declaration. Additional flexibility of the solution achieved by emitting generated intermediate files and providing APIs for creating adapters and decorators on the framework client side. More detailed architecture and implementation of the tool described in the document, along with the evaluation of the solution.

Table Of Contents:

Abbreviations **4**

1. Introduction **5**

2. Thesis Background **6**

2.1 Cloud And Serverless Technologies 6
2.2 Infrastructure-As-Code Approach 9
2.3 Overview of an existing solutions 11
2.4 Thesis background summary 13

3. Architecture Of The Solution **14**

3.1 Implementation language 14
3.2 DSL instruments in Swift 15
3.3. Possible solutions for serverless DSL library 17
3.4 Intermediate representation instruments 21
3.5 Solution architecture summary 21

4. Solution Implementation **22**

4.1 Components overview 22
4.2 Establishing the single source of truth for infrastructure configurations 24
4.3 Avoiding of code duplication and ensuring additional checks 26
4.4 Solution implementation summary 29

5. Testing and evaluation of the solution **30**

5.1 Example of an API 30
5.2 AWS Lambda Runtime Implementation 31
5.3 Implementation with Swift DSL library 33
5.4 Quantitative evaluation of implementations 37
5.5 Solution evaluation summary 41

6. Conclusion **42**

References **44**

Appendix 1 **46**

Appendix 2 **47**

Abbreviations

- API - Application Programming Interface
- DevOps - Developer Operations
- DSL - Domain Specific Language
- GPL - General Purpose Language
- IaC - Infrastructure-as-Code
- UI - User Interface
- TDD - Test Driven Development
- CI - Continuous Integration
- CD - Continuous Deployment
- VM - Virtual Machine
- CLI - Command Line Instruments
- IDE - Interactive Development Environment
- CDK - Cloud Development Kit

1. Introduction

Recently, cloud computing became one of the most rapidly developing fields of IT changing the way large projects can be developed and scaled. Adoption of cloud technologies allows to scale and rebuild the system much easier and may eliminate the need in maintaining its own physical infrastructure. More and more companies are starting to make use of cloud services to build and maintain their infrastructure. The most innovative and fast growing area of cloud computing is serverless computing, having a 21% share of the cloud market and 75% percent annual growth in enterprise adoption (CBInsights, 2018). Main advantages of serverless are that it simplifies the DevOps processes, allows to scale and optimize the infrastructure based on the real-time demand and usually reduces costs of maintaining infrastructure because customers pay only for execution time.

However, despite all the benefits which are provided by serverless approach it requires certain skill and knowledge of specific platforms which provides the service. It means that companies must have special people who are able to work with a specific serverless provider stack and understand how to integrate it into the existing enterprise environment. This may slow down the process of adoption of serverless computing significantly (Eismann, S. 2022).

Current work is dedicated to attempting to solve that problem by providing an instrument for implementing serverless interactions using the DSL framework which will allow the use the same interfaces for different cloud services providers in a unified manner. This will ease the process of integrating new cloud services and will allow for using different providers without the need for significant changes in client implementation.

2. Thesis Background

2.1 Cloud And Serverless Technologies

Serverless is an execution model that provisions computing resources required to run applications on demand and in response to some specific event, scales the resources in the response to demand and stops using the resource when they are not needed (IBM Cloud, 2021). All those features do not require any management actions from the client and are fully encapsulated on the cloud provider side.

Serverless specifies the more broad term of the cloud computing which in turn are usually implemented using various virtualization techniques and implies orchestrating and managing of virtual machines (VM), virtual storages and message brokers to provide all necessary services. And all modern cloud providers offer complex cloud ecosystems with a broad set of available tools and components that are able to serve any client needs. Major examples of such ecosystems are Amazon Web Services (EC2, S3, Lambda), Microsoft Azure (Sphere, Data Lake), Google App Engine - all that services and their components are getting use of virtual infrastructure (Jain, 2016).

Such a wide range of tools serves to fill demand in storage, monitoring, logging, data analytics, deploying and execution, however it also increases the complexity of the system and entry threshold. Any application can make use of only a portion of such services or combine them in different manners without actually caring about the physical infrastructure behind them. But clients still need to care about managing the computation resources on its own and control the allocation and deallocation of executors and computing units (Jonas, 2019).

On the other hand serverless are usually part of cloud services and the key difference is that serverless execution model lifts the client responsibility of managing computation resources onto the cloud provider. Serverless abstracts the process of managing computation powers from client and makes it much easier to scale and maintain while ensuring the efficiency of computing (Eismann, 2022).

High efficiency of the computing in serverless is possible due to the optimization that and predictions that could be made when all physical infrastructure and virtual machines are fine-tuned to interact in a common consistent environment. Serverless model is highly inspired by event-driven architectures which are based on so- called events and triggers, and each event could be initiated by some set of triggers. As an event-driven approach serverless execution model implies the following properties (Maréchaux, 2006):

- Decoupling between components. Publishers know nothing about subscribers and their implementations.

- Many-to-many connections. Events can easily be propagated to any number of subscribers, as well as subscribers may subscribe to any number of publishers.

- Client based behavior. Control flow is determined by the subscriber and depends on how it handles the specific type of the event.

- Asynchronous nature. The process of event messaging and handling is well suited for asynchronous and multithreaded environments.

Consequently, all connections between client and cloud platform are established using the events. Several parts of serverless event model can be highlighted:

  - Event driven bus provided by cloud platform
  - Event handlers provided by client application
  - Event base API provided by cloud platform

Moreover, such an architectural approach allows to accept clients' requests while no resources are allocated, so serverless has zero cost of downtime and all billing for the resources is based on per-usage manner. Which means that clients may not only get resource-management benefits, but also transparent economical improvements. From an architectural perspective of cloud computing, serverless services introduce a new layer of abstraction which is grounded on the base cloud platform infrastructure. This layer is usually composed out of several different components that make it possible to perform necessary tasks in a flexible and unified way. The basic blocks are messaging mechanisms, storagaing, cloud functions and data transformations. Illustration of the serverless computing layer in context of different architectural layers represented on Figure 1.

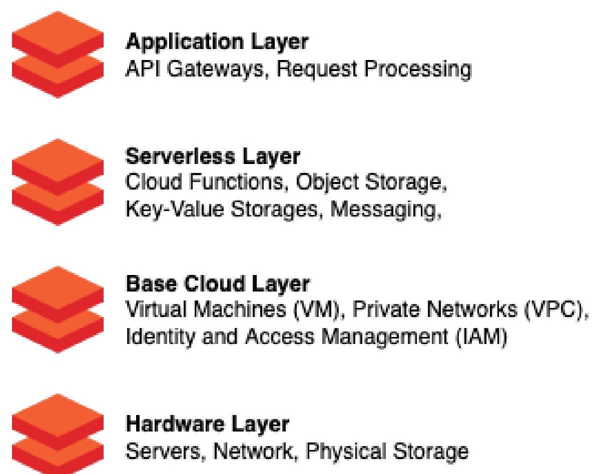


Figure 1. Architectural layers of serververless cloud

To sum up, high-level abstractions of serverless computing provide three significant improvements over more low-level “basic” cloud computing options. These major improvements are (Jonas, 2019):

- High decoupling of components, including decoupling of computing and storage, making it possible to scale and pay for them independently.

- Eliminating resource managing overhead during the computing

- Pay for usage billing model, which means paying only for the resources that are currently in usage and in proportion relative to the load.

For illustration purposes, the simple example of a serverless application which serves to handle requests through HTTP/HTTPS on Figure 2 could be described. The first element of the application is REST API which is some service provided by the cloud platform and which actы as an HTTP adapter which is accessible to the clients. This adapter's primary role is to transform user requests into the cloud platform events, here it’s worth mentioning that cloud platforms usually provide other interfaces based on WebSockets or Messaging Queues too. That events derived from the HTTP requests are then forwarded to event handlers. Event handlers are implemented by the client and perform custom logic which is appropriate for that event. These handlers are often called Lambdas or Cloud Functions. Those functions may call other functions or perform requests to other services based on the cloud platform, and get access to the storages. After all the calculations are completed the client receives a usual HTTP response from the REST API adapter. After that function is considered as finished by the cloud platform if there are no new requests during some interval and all necessary for that function resources become deallocated. It’s worth mentioning that while the function is finished there are no computational resources allocated for it. That fact has one significant drawback, calls to that function may experience significant delays - from ~10ms to several seconds. To get rid of such problems serverless clients may have a pool of “warm resources” by regularly calling functions with some interval to maintain the allocated resources.

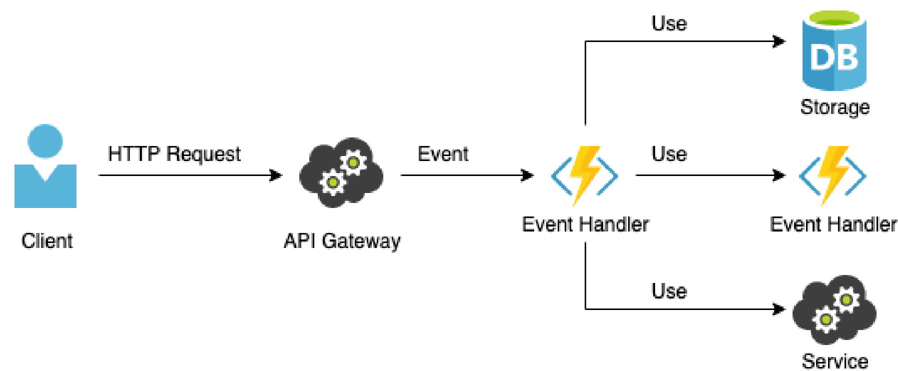


Figure 2. Simple serverless application structure

2.2 Infrastructure-As-Code Approach

While serverless computing provides rather high-level interfaces for performing a wide range of tasks, they still significantly rely on the virtual infrastructure maintained by the cloud platform. Serverless properties of high-isolation and security are directly inherited from the underlying virtual infrastructure. That virtual infrastructure is based on virtual machines (VM), containers, unikernels and similar virtualization tools. To ensure high performance and proper sandboxing of resources along with comprehensive utilization of multi-tenant resources, serverless cloud must include fine-grained orchestration mechanisms (Jonas, 2019).

One of the most popular solutions that serve purposes of virtual infrastructure management is Kubernetes (Burns, 2016), which allows smoothless multi-tenant computing multiplexing and deploying short-lived isolated environments.. Major cloud platforms provide Kubernetes orchestration integrated with their infrastructure - Google Kubernetes Engine (GKE) and AWS Elastic Kubernetes Service (EKS), that gives developers flexibility in terms of configuring arbitrary containers, while having all operational overhead on their side.

Together with more high-level interfaces that were discussed in section 1.1, orchestration mechanisms compose the full-fledged serverless infrastructure. Naturally, any serverless based application requires proper deployment of the whole serverless infrastructure. That infrastructure must be reproducible and reliable to provide the application with expected behavior. All cloud providers have several options for creating and deploying infrastructure, those options are UI-based configurations, API-based configurations or Code-based configurations.

All of those methods could be used in production, however UI-based configurations are rapidly losing their popularity, due to serious flaws they have in comparison with other methods. First of all, infrastructure is hardly reproducible, because everything is made manually through some kind of specialized UI console. That makes it impossible to deploy the infrastructure in a fast and automated manner and limits reusability of such configuration for different modules or applications. Another big problem of such an approach is versioning - evolution of the application inevitably leads to changes in the infrastructure, but with UI based configurations it’s impossible to keep configuration in sync with the application code. On the other hand, infrastructure fully defined and configured using configuration files solves all the stated problems.

Infrastructure-as-Code (IaC) approach is a way of defining a deploying infrastructure using configuration files and scripts written in some language. Such methodology is highly inspired by common software development practices which evolved through decades of enterprise software engineering. IaC emphasizes repeatable, reusable and consistent configurations and opens an opportunity for applying best testing practices such as Test Driven Development (TDD) and Unit Testing (Morris, 2016). Moreover in the case of IaC approach all configurations would be stored along with the regular code of the application in the same repository which means that infrastructure would be versioned and naturally synchronized with application code. Therefore, having the code versioned in the same repository with the rest of the application, also allows for applying Continuous Integration (CI) and Continuous Deployment (CD) practices, the whole process of applying IaC configurations is illustrated on Figure 3.

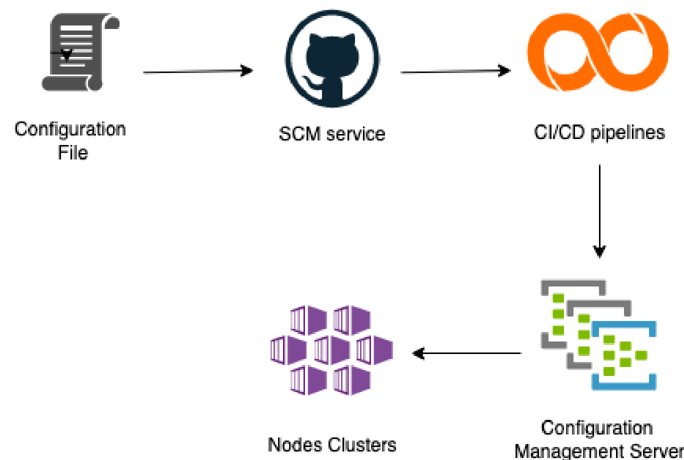


Figure 3. Infrastructure-as-Code (IaC) approach schematically

Figure 3 shows the whole process of deploying the configuration on the actual infrastructure, and the process looks mostly the same as with deploying application code while exploiting the same automation mechanisms. Usually companies and big projects use IaC to achieve several goals that are inherited from the main beneficial properties of IaC (Morris, 2016):

- IT infrastructure becomes “open to changes” and requires less time for configuration

- Teams can easily recover from failures and replay configurations

- Changes to the infrastructure could be made in continuous and iterative manner

- Infrastructure solutions could be proven and documented through tests

There are no limitations on how the configuration file must be structured; it could be written with imperative or declarative paradigm in mind. But the most sensible and natural way of creating configuration files is using declarative Domain Specific Languages (DSLs), which allows to specify all needed properties in a concise and brief form.

A Domain Specific Language (DSL) is a language primarily intended for use in the context of some specific domain. Compared to General Purpose Languages (GPLs) DSLs are usually Turing incomplete and have only syntax constructions closely related to the domain where they are used. Thus, DSLs configurations are much easier to read and implement, because they are not overwhelmed with unnecessary implementation details and emphasize use of the essential abstractions. There are two type of DSLs based on the way they are executed (Mernik, 2005):

- External DSL - language that is written in a distinct language with its own interpreter and processed or executed by another so-called host language.

- Internal/Embedded DSL - host language and DSL are the same and DSL written in that language semantics and processed by it.

While both approaches have their own benefits - in the case of internal DSL leads to much tighter integration between the domain described in DSL and other applications. That also opens up opportunities to work with configuration of the infrastructure without additional competencies. Any developer who works on the project would be able to understand and work on the infrastructure, without the necessity to learn new tools and languages. Also, using DSL reduces not only the entry threshold, but also the amount of code in terms of lines of code (LOC), due to the domain oriented syntax(Mernik, 2005).

2.3 Overview of an existing solutions

There are several solutions for IaC configuration of cloud infrastructure in general and for serverless in particular. Different solutions provide different ways of integration - some solutions are specific to the one cloud provider, others specific to only serverless clouds, several have support for multiple clouds. Solutions are implemented in different languages and use different paradigms and approaches.

- Google Cloud Deployment Manager - is an IaC tool dedicated to configuration and deployment of the Google Cloud Platform (GCP) infrastructure. Tool provides functionality to create and manage infrastructure resources using declarative YAML language in conjunction with Python scripts. It suggests a wide range of settings to give users as much control of infrastructure as possible. The main problem of the solution, that it works with GCP exclusively and is relatively verbose.

- AWS CloudFormation and AWS CDK - solution for working with Amazon cloud services. AWS CloudFormation is a solution based on the YAML/JSON configurations and it offers shared language for description and provisioning resources in the Amazon cloud environment. It offers concise and understandable DSL in terms of Amazon services domain. On the other hand AWS CDK is a set of frameworks for several popular languages, which provide type-safe way of interacting with AWS cloud.

- Terraform - a serverless tool that supports a wide range of cloud providers, and works not only with serverless infrastructure, but also with base cloud infrastructure, taking a lot of responsibility that is hidden by serverless on itself. Initially developed for more classical infrastructure Terraform syntax may be too verbose and contain a lot of unnecessary implementation details. Configurations are declared using YAML.

- Serverless.com framework - provides a powerful instrument of configuring and deploying infrastructure configuration using several popular cloud providers. In contrast to Terraform which also supports different providers and configured using YAML Serverless.com is focused only on serverless computation, thus it eliminates the verbosity which exists in Terraform declarations. However, it supports fewer cloud providers and in some aspects is less flexible than Terraform.

All the mentioned solutions provide production ready instruments for implementing serverless applications, however while some of the instruments like AWS CloudFormation provide frameworks for specific languages making the interactions with cloud type-safe and more tightly integrated with application code, they are specific only to some cloud providers. On the other hand tools like Terraform and Serverless.com are more universal and may be used with different providers, they don’t provide type-safety and actually represents a parallel codebase to the main application, which can be out-of-sync with application code in terms of interfaces, constants and API endpoints. Changes to application code are not reflected in the infrastructure configurations. Also managing large amounts of configurations still may include a lot of tedious and error-prone work and lack of single architectural structure. While the ability to call and create configurations straight from the application code will simplify and unify the process of creating serverless applications. So, at this point it seems reasonable to provide general purpose languages with DSLs which will support all language features, auto completion, type-safety and will fit well into the existing application architectures. That approach would also solve the problem of synchronization between infrastructure code and application code, making it possible to detect such problems at compile-time and also perform integration and unit tests in the same test infrastructure that application already has.

2.4 Thesis background summary

Serverless computing is one of the most prominent programming paradigms in modern software engineering. And in the advent of serverless programming it's important to build a solid and scalable foundation taking in account time-proven practices that ease the process of development and deploying the artifacts. As serverless computation resides in the bounding context between infrastructure management and software development, the best approaches from well-established development processes could be encompassed there too. Infrastructure-as-code is one of those techniques which allows to gain more control over the classical infrastructure management and could be successfully applied in the case of serverless computing too and existing solutions provide several ways to do that using custom configurations. However, using DSL and code generation in conjunction with infrastructure-as-code may facilitate and abstract infrastructure details even further, allowing the developer to keep the infrastructure declarations in sync with changes made to the application code and share the same objects between infrastructure declarations and application code, making room for additional type-safety and checks for correctness. The properties which can not be found in existing solutions.

3. Architecture Of The Solution

3.1 Implementation language

Swift language has been chosen as an instrument for the project implementation. Swift is an open-sourced general purpose programming language that was announced in 2014. It's maintained and supported by Apple and supports multiple platforms: all Apple platforms, Linux, macOS, Android. The language by itself is primarily used as an instrument for development on Apple devices. However, the language is open-sourced and is actively adopting new features for server-side development since 2016 by means of the special "Swift Server API Workgroup" which consists of Swift core team from Apple as well along with engineers from Amazon and MongoDB. These API will provide low-level "server" functions as the basic blocks for developing server-side capabilities, removing the reliance on interfacing with generally platform specific C libraries. The latest stable releases of Swift throughout the 2021 already included prominent changes to the standard libraries and memory model - now the language standard libraries provide high-level interface intended to work with WebSockets, TLS, UDP, TCP with introduction of Network library, also Swift adopted the async/await paradigm and structural concurrency (Swift Evolution Async/Await., 2021). All of the recent changes in the language have an explicit vector towards server side development and there is still a huge space for proposing features and incorporating them into the Swift runtime.

Characteristics that make Swift a good choice for server-side applications in general and serverless and cloud computing in specific are:

- Small memory footprint - Swift has a smaller memory footprint, compared to popular server-side languages that rely on automatic memory management and may achieve memory footprints close to C/C++ footprint results. Benchmarks related to other languages and measuring techniques may be found on the benchmarking project web-site. That property is valuable for cloud computing, since it allows maximizing resource utilization on cloud platforms.

- Quick startup time - Swift applications have small boot time, since there are almost no warm up operations, that implies less wait time for streamlining CI/CD processes and deploying on VMs or containers. That property is especially precious for serverless computing and cloud functions, since it helps to reduce the "cold-start" problem of serverless that was mentioned in paragraph 1.1.

- Deterministic performance - Swift doesn't use garbage collection techniques and lacks just-in-time (JIT) compilation, which means light runtime without additional operations and memory allocations tracing. That also makes Swift runtime more deterministic compared to JIT based languages, since even modern garbage techniques still may produce non deterministic behavior. JIT also imposes delays in the initial execution of applications, since it is necessary to load and compile bytecode in the runtime.

- High level APIs and expressiveness - all the previous properties make Swift stand out from the background of high-level languages such as Python or Ruby. But it stills benefits on the background of more low-level language like C/C++ or Rust, in

terms of rich type-system and expressive syntax, while providing direct interoperability with C and unsafe memory operations.

There are already developments regarding the native serverless computing in the "swift-server" project. One of them is the "swift-aws-lambda-runtime" module, which is actually the Swift implementation of AWS Lambda Runtime, it's designed to make building AWS Lambda functions safe and simple. That library uses an embedded asynchronous HTTP Client based that is fine-tuned for performance in the AWS Runtime context. The library provides a multi-tier API that allows building a range of Lambda functions: From quick and simple closures to complex, performance-sensitive event handlers. As most Lambda functions are triggered by events from AWS platform - library module AWSLambdaEvents provides implementation for common AWS events to simplify writing Lambda functions. The library is based on three main protocols which provide interfaces of different abstraction levels - ByteBufferLambdaHandler, EventLoopHandler, LambdaHandler. All of them provide strongly typed asynchronous processing protocols for Lambda functions. The library provides safe and high performant native instruments for AWS serverless infrastructure. However, while the core API is considered stable the library is still on its way to 1.0 version and doesn't provide convenient ways to deploy serverless infrastructure. Now, to deploy Lambda function to AWS the developer has to manually compile the code for Amazon Linux using additional instruments like AWS Serverless Application Model (SAM), AWS Command Line Instruments (AWS CLI) or Amazon Linux XCode toolchain which is a plugin that integrates with the XCode build system and allows to build the executable using XCode, but binds the developer to using the XCode development environment.

Summing up, it can be said that the Swift language was chosen as the language for the implementation of the project due to its qualities perfectly suitable for cloud development and the ability to integrate new solutions into open source projects supported by the language team, since the language roadmap is targeting serverless and cloud computing realm.

3.2 DSL instruments in Swift

In 2019 Apple has announced its brand new declarative SwiftUI framework for developing on iOS devices. With the introduction of SwiftUI, Swift 5.1 received new hidden functionality that was called "result builders". After stabilizing the result builders API, it was officially unveiled in Swift 5.4 release in 2021, the motivation and architectural decisions behind that feature described in detail in the corresponding "swift evolution proposal" (Swift Evolution, 2021). The main effect of this proposal is that it allows implementation of a new type of embedded DSLs in Swift by calling transformations to the statements of the "building function".

Result builders is a feature that allows certain specially-annotated functions to collect and build up the resulting values from a sequence of components or statements. The idea can be illustrated with a simple Figure 4 from the feature proposal (Swift Evolution, 2021). Such syntax allows the design of type-safe and expressive DSL libraries inside Swift to describe special domain specific problems. That approach is especially useful in conjunction with code generation techniques, including generating structured data formats

like JSON, YAML, etc. The feature is based on compile-time code generation and does not affect runtime performance of the applications.

```
@TupleBuilder
func build() -> (Int, Int, Int) {
   1
   2
   3
}

// This code is interpreted exactly as if it were this code:
func build() -> (Int, Int, Int) {
   let _a = TupleBuilder.buildExpression(1)
   let _b = TupleBuilder.buildExpression(2)
   let _c = TupleBuilder.buildExpression(3)

   return TupleBuilder.buildBlock(_a, _b, _c)
}
```

Figure 4. Result builders example

Any single result builder must implement just two basic requirements:

- Result builder must be annotated with `@resultBuilder` attribute, which allows that type to be used as result builder.

- Result builder must have at least one invariant of the static `buildBlock` method.

However, to provide flexible syntax and rich functionality, the result builder should provide an extended set of static result-building methods. The full set of result building methods contains the following methods:

- `func buildBlock(_ components: Component...) -> Component]`
- `func buildOptional(_ component: Component?) -> Component`
- `func buildEither(first: Component) -> Component`
- `func buildEither(second: Component) -> Component`
- `func buildArray(_ components: [Component]) -> Component`
- `func buildExpression(_ expression: Expression) -> Component`
- `func buildFinalResult(_ component: Component) -> FinalResult`
- `func buildLimitedAvailability(_ component: Component) -> Component`

Architecture incorporated in ResultBuilders basically implements the closure of operations design pattern mentioned by E.Evans in “Domain Driven Design”. This design pattern is inspired by rigorous mathematical formalisms, where the operations are usually closed under the set of its operands. Such refinement of interfaces significantly simplifies the

interpretation operations and allows for combining and chaining operations in the declarative form, resembling the way how mathematical statements are expressed. In conjunction with immutable value objects such approach enforces side-effects free functions and safe-multithreading.

Based on the set of implemented methods, several imperative and control flow constructions could be used in result building context. Exception-handling statements, if-else statements, switch statements and for-in loops can be used inside of the building context, while the builder implements necessary methods. More detailed documentation and future plans to improve the Result Builders API could be found on its evolution proposal.

```
NavigationView {
   HStack {
      CircleImage(image: viewModel.avatar)
      VStack {
         Text(viewModel.name)
            .padding(.leading, 15)
            .font(.title)
         Text(viewModel.personastate.description)
            .font(.headline)
      }
      Spacer()
   }

   VStack {
      FriendListView(friendList: viewModel.friends)
   }
}
```

Figure 5. Production-ready snippet of SwiftUI DSL

3.3. Possible solutions for serverless DSL library

There are several frameworks and standalone solutions exist as the open-source solutions and as proprietary solutions too. They have different integration processes and work with different sets of cloud providers. All those properties influence possible architectural decisions and reusability issues related to the library. Some of the approaches for the library that were considered during the research are using third-party cloud provider specific SDKs as external dependencies to the library or using DSL to generate code for solutions that support multiple cloud provider platforms.

Summing up all the problems in existing solutions the requirements for the DSL framework could be formulated as following. The implementation must be provider agnostic allowing to ease the problem of vendor lock-in that exists in serverless computing. However, there should be a way to integrate some specific features that are part of the providers ecosystem, using modifiers or decorators. This will provide the user with all possible capabilities, while keeping the separation of concerns in place. The solution must seamlessly support integration of different cloud providers, without forcing the user

to write its own integration code for each vendor. The implementation dependency and version management should be kept as simple as possible, eliminating the situation where the new versions must be published when any of the dependencies were updated. There must be a way to define all the necessary configuration parameters from a single source of truth, even in cases when providers require separate configuration files for different purposes like authentication constants, database configurations or environment parameters. To support different cloud providers and consequently various computational environments functions deployed using the DSL declarations must support different forms of publishing like plain binaries and Docker images.

1. Library based on the third party SDKs approach.

   That approach implies the use of SDKs provided by specific cloud platforms, for example AWS Swift SDK, Azure Swift SDK or Google Cloud Swift SDK. In that case several SDK specific wrappers must be implemented in the library and each wrapper would have to implement custom logic to provide the common behavior and common abstractions across all the platforms. All the internal quirks and differences between platforms would have to be resolved during the development of the library.

   Solution advantages :

   - Library uses official SDKs provided by cloud providers
   - Type-safety ensured on the 3rd party SDKs level
   - Easy and predictable integration
   - Unnecessary functionality provided by more general cloud SDKs

   Solution disadvantages:

   - Wrapper for each specific SDK should be implemented manually
   - Set of cloud providers is limited by number of available SDKs
   - Manual SDKs version management

That solution requires a large amount of boilerplate logic to implement wrappers for each provider, while the opportunities to use different cloud platforms are highly limited, since only large-scale popular platforms provide SDKs for different languages and inappropriate for applications that are going to use less popular platforms.

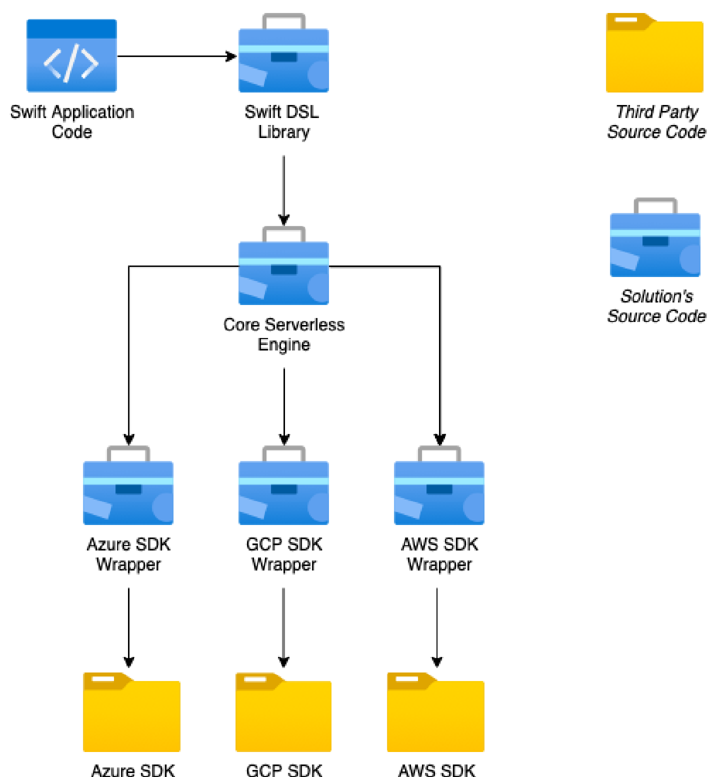


Figure 6. High-level architecture of the solution based on SDKs

Another significant problem which exists in that solution is that each specific SDK API update may trigger changes in the source code of the library and introduce overhead related to monitoring and adopting new SDKs versions. SDKs provided by cloud platforms are not optimized only for serverless interactions, but also contain additional overhead to interact with all services included in cloud platform infrastructure.

2. Library based on the code generation

That approach includes working with instruments that support several cloud platforms and store configuration files in separate structured data formats, making it possible to generate such files from Swift runtime using DSL interface. That approach fully utilizes the DSL opportunities and puts all the heavy lifting of providers integration to the underlying frameworks such as Terraform or Serverless.com.

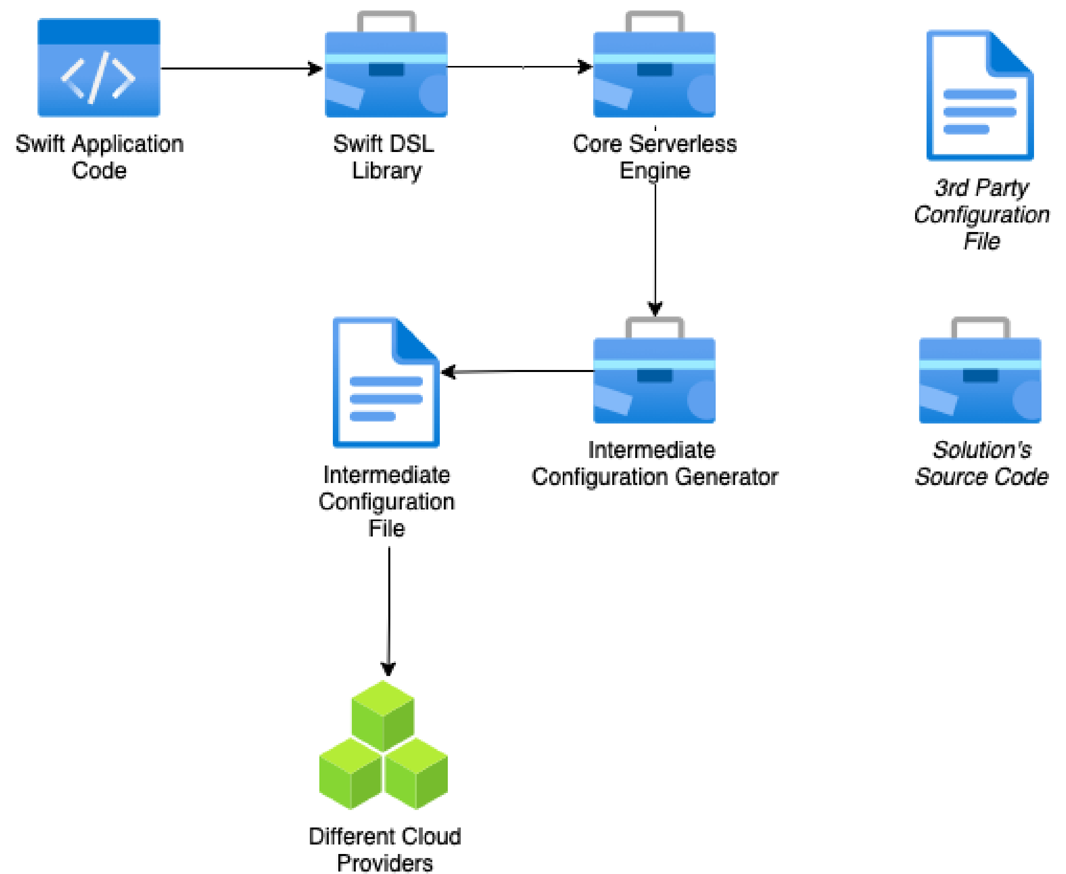


Figure 7. High-level architecture of code-generation based solution

Solution advantages:

- Single code generator for different platforms
- Number of platforms limited only by 3rd party supported providers
- Intermediate artifact in form of configuration file may be reused
- All SDKs version management overhead lies on 3rd party
- Less boilerplate code in the library

Solution disadvantage:

- Type-safety must be ensured on the library side
- Code generation requires more complex code

That solution provides a unified way to declare and generate serverless infrastructure without coupling to specific cloud providers on library level. In that case the library produces intermediate artifacts that further can be shared between projects to smoothly recreate the same environment. Moreover it has a perspective to be extended on different languages in the form of lightweight language-specific wrappers, very similar to how Protobuf protocol was implemented by Google, saving type-system power of each individual language, while having language and platform independent intermediate

representation. The only serious disadvantage of such an approach compared to the integrating SDKs per provider is that code generation usually requires more complex code structures and algorithms.

3.4 Intermediate representation instruments

Both of the solutions presented above in paragraphs 2.1 and 2.2 provide several benefits and several drawbacks each, however the possible architecture based on the code-generation techniques seems to be more viable and platform/language independent. Those properties are crucial for providing a flexible and universal solution. Thus, it was decided to use the code generation approach, since it provides more valuable benefits. Two existing solutions mentioned in paragraph 1.3 support multiple cloud providers and their infrastructure could be used as an intermediate representation of the proposed solution - Terraform and Serverless.com.

On the one side Terraform is a classical open-source solution which is now used in wide-range of applications and has a large and active community as well as extensive documentation and code examples. Terraform makes use of its special HashiCorp Configuration Language (HCL), which allows to describe and deploy any kind of cloud configuration, Terraform is able to manage a broad range of services including PaaS and IaaS. HCL by itself is a full-fledged complex language with its own syntax and special constructions. While it makes Terraform applicable for any type of task in the context of cloud infrastructure, it also makes it redundantly verbose for solely serverless infrastructure purposes and adds complexity to the task of translating Swift DSL to the HCL representation.

On the other side Serverless.com is the solution initially positioned as serverless oriented and also has an open-sourced nature. But in contrast to Terraform its limited only to serverless realm, at first glance it only imposes limitations on the usage, but in fact that also brings simplicity and brevity to the Serverless.com configurations. Syntax of Serverless.com configurations are more concise and can be used as intermediate representation more easily than HCL, without imposing limitations on the current project, since the project is targeted solely for serverless environments.

3.5 Solution architecture summary

Swift language is actively developed in open source with Apple support and aims on natively adopting new programming paradigms. Concise and powerful concurrency model inspired by structural concurrency ideas and implemented in form of actors and async/await interactions, native support of distributed computing by usage of distributed actors make Swift very promising in terms of modern server-side development. Thus ideas of unifying and abstracting the serverless computation using the language foundation in the area of DSL building may have a way to the language standard libs. The idea of intermediate representations for infrastructure configurations is inspired from the way modern compilers work and may allow for a flexible and structured way to deploy infrastructure from the code without the need to learn completely new concepts and languages.

4. Solution Implementation

4.1 Components overview

The implementation of the solution is written using the latest Swift 5.6 version (at the moment of writing), however serverless functions themselves could be written with any compatible Swift versions, as well as in other languages, but it will mean loss of unification and compile time checks. The library would be distributed using SwiftPM packages and have the structure illustrated on Figure 6. The reasoning behind such division is that the SwiftyServerless package will only contain logic and abstractions strictly related to the serverless domain overall, so it would be unaware of specific cloud providers and their configuration details.

Meanwhile, that package will expose the API that is required for building custom provider-specific adapters if needed. The SwiftyServerless package makes use of SPI (system programming interfaces) experimental feature to maintain the proper level of interface segregation and encapsulation. System programming interfaces allow to make the methods and initializers public only for the client code that imports the host package with specific @_spi attribute. Modules exposing SPI and using library evolution generate an additional .private.swiftinterface file (with -emit-private-module-interface-path) in addition to the usual .swiftinterface file. This private interface exposes both API and SPI. Clients can access SPI by marking the import as @_spi(spiName) import Module. This also makes it easy to find out which clients are using certain SPIs by doing a textual search. @_spi is a private attribute that is mostly used for Swift compiler development, but it is undergoing the Swift evolution process, which means that most probably it won’t be changed in the nearest future. For example ServerlessServiceRepresentation class is marked with @_spi, so the adapter packages may transform and enrich initial representation with new information in a functional manner - without changing the initial instance.

Such a functional approach to the data transformation allows to enforce multithreading safety on solution design level, because it eliminates the possibility of data-races, due to the absence of shared state between threads. While the creation of new representation may take additional computation efforts the representations itselves are simple lightweight DTO structures, thus the overhead would be unnoticeable.

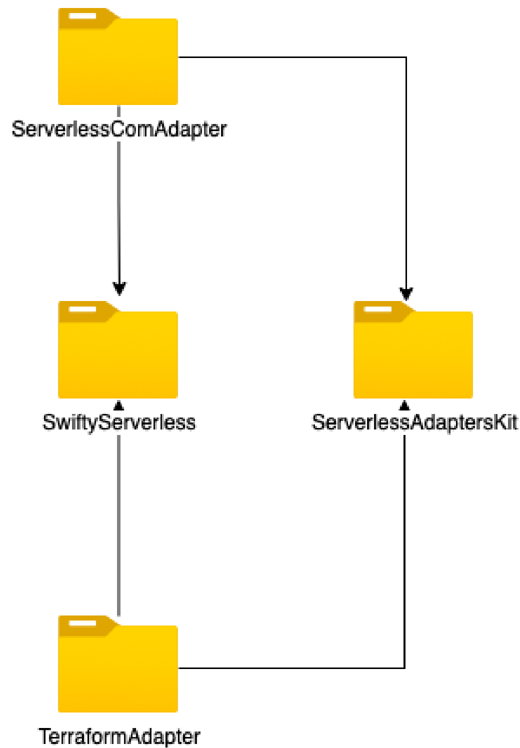


Figure 8. Swift packages structure

The DSL implementation is based on the hierarchy of ResultBuilders in the form of tree-like structure. Each result builder collects the heterogeneous set of elements and forms the representation of the described component. The opportunity to work with heterogeneous collections, without losing the actual types metadata is achieved by using enumerations with associated values, as illustrated in Figure 9.

```
public enum Types {
        indirect case list([ServerlessServiceParameters.Types])
        case provider(ServerlessProviderRepresentation)
        case function(ServerlessFunctionRepresentation)
}
```

Figure 9. Enum with associated types

Such structure allows to wrap up the underlying type information like ServerlessProviderRepresentation or ServerlessFunctionRepresentation into the single data type, which effectively makes the collection homogeneous for compiler, but allows to retrieve wrapped data in order to perform additional checks or to manipulate and enrich specific data. The simple tree of serverless representations is depicted on Figure 7.

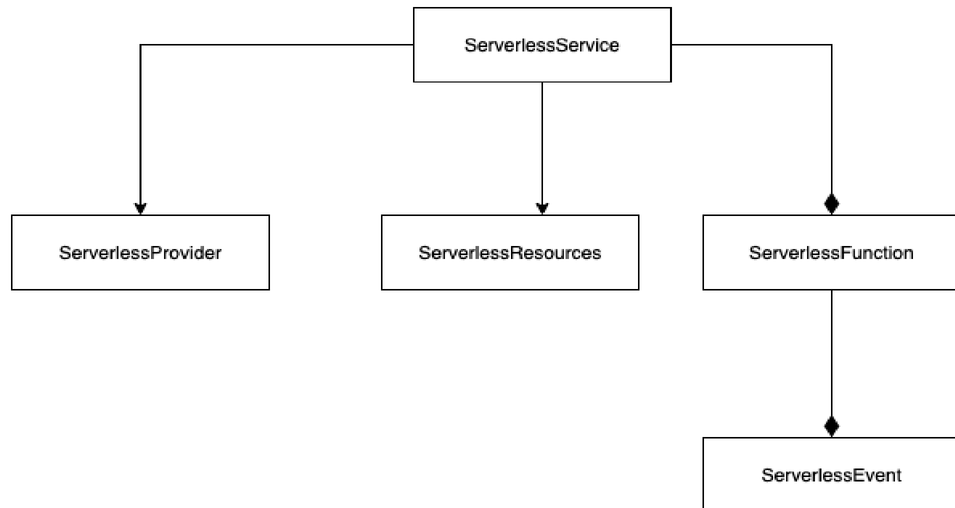


Figure 9. Tree of serverless components representations

While the actual structure of the would remain more or less the same, the actual content of any node must be open to modifications. In order to give room for implementing various plugins or modifiers custom decorators which conforms to the Decorator protocol could be used. That approach makes it possible to add new parameters to the declarative representations dynamically and then implement custom options for encoding new properties into desired intermediate representation on adapters level. That also brings another crucial advantage of such a framework - it is possible to combine and generate additional configurations from the single DSL representation.

4.2 Establishing the single source of truth for infrastructure configurations

Often, setting up the serverless environment and deploying functions require not only one configuration file, but a set of files each dedicated to the specific subdomain like authentication, databases etc. In such a situation, it becomes even harder to maintain all the necessary configurations in the appropriate state, because there is no single source of truth, which contains all configuration properties in one place. Moreover, those configuration files are also usually located in different directories. But, code generation techniques allow us to keep all configurations in sync by generating all the necessary files from one declaration even if they have different formats or must be located in specific directories. For example, OpenWhisk cloud provider requires to declare a separate .wskprops configuration file which contains key/value parameters, usually for authorization purposes. So, in case of deploying the serverless function the developers have to create such a file.

```
APIHOST=openwhisk.ng.bluemix.net
AUTH=xxxxxx:yyyyy
APIGW_ACCESS_TOKEN=<some token>
```

Figure 10. Example of .wskprops required by OpenWhisk

The configuration itself is very simple and it's desirable to have it in the same place, along with other parameters. That could be achieved using decorators as they allow additional parameters and encode them in appropriate format. For example, the declarative Swift code from Figure 11 makes use of .configure decorator which adds additional configurations and writes them into the format associated with the given provider. Basically, output of that declaration would consist of two intermediate files: .wskprops with specific provider settings and serverless.yml with serverless.com representation.

```
service(name: "example", version: "3.12.0") {
   provider(name: "openwhisk", ignoreCerts: true)
      .configure(using: OpenWhisk.config)
}
.plugins("serverless-openwhisk")
.serverlessComRepresentation()
```

Figure 11. Simple service declaration using Swift DSL

```
service: example
frameworkVersion: 3.12.0
configValidationMode: error
useDotenv: true
provider:
  name: openwhisk
  runtime: swift
  ignoreCerts: true
plugins:
- serverless-openwhisk
```

Figure 12. Generated serverless.yml

```
APIHOST=http://localhost:3233
AUTH=23bc46b1-71f6-4ed5-8c54-816aa
```

Figure 13. Generated .wskprops file

Demonstrated code-generation approach emphasizes operating on the higher level of abstractions, more tightly related with the domain, instead of digging into implementation details of the authorization process associated with the provider, because such necessity to manipulate with the implementation details undermines the whole purpose of abstractioning other parts of the code. Versatility and suppliness of that approach is also empowered by the ability to share, interpret and modify the fully platform agnostic intermediate artifacts. All the intermediate representations could be easily collected and transferred, meanwhile the collection of such files may independently describe the whole serverless infrastructure, including the provider specific details. That allows us to reproduce required infrastructure in different environments, while keeping the generated configuration files without changes. Moreover, richer abstractions could be built over for different provider adapters - some providers adapters may provide a way be configured

using permissions components, auth components or resource components which will further be interpreted into separate configuration files.

```
service(name: "example", version: "3.12.0") {
   provider(name: "openwhisk", ignoreCerts: true)

   auth("23bc46b1-71f6-4ed5-8c54-816aa")
.readOnly
}
.hostedOn("http://localhost:3233")
.plugins("serverless-openwhisk")
.serverlessComRepresentation()
```

Figure 14. DSL using higher level abstractions for authorization and permissions management

However, a more general option for setting up the config still must be preserved and also extended to make it possible to specify the configuration file instead of generating one, because by some means it could be interpreted as a more secure way to handle permission or authorization configurations, without explicitly exposing constants to the application code. Such vulnerability is classified as “CWE-798: Use of Hard-coded Credentials” according to Common Weakness Enumeration (CWE). Hard-coded credentials usually create a highly dangerous hole that allows an external attacker to overcome the established authentication process. There are two variants of the vulnerability:

- software contains an authentication mechanism that checks the input credentials against a hard-coded set of credentials

- the software connects to another system or component, and it contains hard-coded credentials for connecting to that component

An example of the case when the user would prefer to specify a standalone config file instead of autogenerated one is continuous integration implementation with Jenkins. To ensure proper level of security the DevOps developer most likely will store the file using Credentials store, which allows to store the file ones, but does not allow to fetch its contents afterwards. Such credentials configured in Jenkins are stored in an encrypted form on the controller instance and are only handled in Pipeline projects via their credential IDs. This minimizes the chances of exposing the actual credentials themselves to Jenkins users and hinders the ability to copy functional credentials from one Jenkins instance to another.

4.3 Avoiding of code duplication and ensuring additional checks

Nowadays, server application developers have to declare the serverless infrastructure configuration and actual application code separately. Such division has several negative implications which could be neglected by declaring the infrastructure along with the application code. The total separation of infrastructure from the application code not only

leads to the losing of a single source of truth as was stated in paragraph 3.2, but also impoverishes the opportunities for checking the correctness of integrations between the infrastructure and the application. The code snippet on the Figure 15 demonstrates the declaration of a simple application endpoint that is intended to return the list of users to the client.

```
let app = Application()
let router = try app.make(Router.self)
router.get("/users") {
   req in return try User.all()
}
try app.run()
```

Figure 15. Declaration of a simple endpoint "/users"

There is nothing wrong in declaring such a simple endpoint, but if the developer will need to tie that endpoint with the serverless function, it would have to copy-paste the route to the endpoint. While it does not seem to be a big problem in the small example, such a solution is not scalable and highly error-prone when the number of endpoints increases. Traditionally developers have to manually track changes to the endpoint and update according configuration for the serverless infrastructure, eventually the endpoints stated in the application code and in the serverless configuration files could easily go out of sync. In the best case scenario such problems would be identified on the stage of automated integration/acceptance testing or during the manual testing stage. In the worst case scenario such a problem may occur after production deployment, if there were not enough automated tests and the manual testing cases have missed the case where the function is used. In both cases, the software development process would suffer negative consequences of the technical issue. At best, some efforts would be spent on finding the bug, fixing it and retesting which may take a noticeable amount of time depending on the established software processes. At worst, client experience would be affected by such mistakes and even more resources would be spent on fixing the problem. But, it is possible not to increase the development and testing cycles length and not to put the risk of exposing erroneous code to the clients, with the help of build time checks.

If the solution makes it possible to generate the code for serverless infrastructure then it's also possible to build more precise and type safe abstractions in a way that is usual for any developer who works in OOP paradigm. To create an intention revealing interfaces that operate with the ubiquitous terminology that is clear to any experienced developer the DI Container pattern could be applied for typed integration of serverless functions between different components of the app, while still preserving the low-coupling between the concepts of infrastructure and business logic. The purpose of such a container would be to eliminate the possibility of desynchronization by introducing a new layer of abstraction. The declarative declaration of service using the functions container is presented on Figure 16.

```
service(name: "example", version: "3.12.0") {
        provider(name: "openwhisk", ignoreCerts: true)
                .configure(using: OpenWhisk.config)
```

```
function(of: UsersHandler.self)
        .stage(.production)

function(of: PhotosHandler.self)
        .stage(.dev)

}
.plugins("serverless-openwhisk")
.serverlessComRepresentation()
```

Figure 16. Declaration of the typed functions registered in the container

With that approach the handler object is the single source of truth for both infrastructure part of the code as well as for the application code, so the unnecessary and error prone duplication of code was eliminated. Moreover, all the necessary meta information about the handler, like file path could be inferred automatically. The implementation of the underlying dependency container, which is depicted on Figure 17 is relatively simple, but it allows access and register entities based on their type.

```
actor DefaultHandlerContainer: HandlerContainer {
  private var services = [String: AnyHandler]()

  private func typeName(_ anyHandler: AnyHandler) ->
String {
    return "\(type(of: some))"
  }

  func register<T>(service: T) {
    let key = typeName(T.self)
    services[key] = service
  }

  func resolve<T>() -> T? {
    let key = typeName(T.self)
    return services[key] as? T
  }
}
```

Figure 17. Implementation of handlers container

That implementation allows to resolve specific handlers based on their type, and returns not the abstract protocol, but actual class which allows access to the specific properties and metadata of the class. It lifts the responsibility of providing the correct file path to the handler from the developer and that is needed for developer to use the handler is to create the class object, then make it conform to the Handler protocol and that's all. The implementation is simple and has room for evolution - one obvious drawback is that it still provides only runtime or build time checks, because if the handler has not been registered before using in declaration it will propagate an error. Inside there is a dictionary, where the

key is a string containing the type name, and the value is an object that we register in the locator. To get some kind of dependency, you need to register it first. Handlers should be registered in one place at the start of the application.

The handlers container has a mutable state inside it and while it is highly likely that container could be accessed from different threads, the access to the state must be synchronized to allow safe usage in a multithreading environment. That behavior could easily be achieved using the new Swift actors model, which enforces isolated context when the properties of the actors are being accessed from the outside, but the client code would have to access methods of the actors an asynchronous ones, because accessing the properties may assume waiting for other threads to finish their operations.

Opportunity to move such checks to the compile time stage is possible using source code analysis and currently is being investigated. Such code analysis could be performed using the SourceKit library that is distributed within Swift toolchain and provides rich source code indexing. Similar approach is used in Dagger for JVM and there is also an adaption for Swift by Uber in their Needle framework which is still in the early stage of development, having the last version of 0.18.1, while claimed as production ready and used across Uber development environment.

4.4 Solution implementation summary

Proposed solution introduces several important properties to how the serverless infrastructure code is written. It makes it possible to use the syntax constructions that are common for developers of the application and get rid of provider specific SDKs which also does not provide a way to create infrastructure configurations. In turn, the DSL framework, also automates generation of helper files and becomes a single source of truth. Duplication of code and out-of-sync problems mitigated by such a single source of truth and that lead to more maintainable and detangled structure of code. Creating the declaration inside the host language of the application, also contains an advantage which is natural to cloud provider SDKs - type safety and build-time/compile-time checks that can reduce time spent on the software testing stage.

5. Testing and evaluation of the solution

5.1 Example of an API

To demonstrate and evaluate the solution, we need to conduct a proof-of-concept evaluation based on the several APIs implementations one of which would be implemented using AWS Runtime CDK, other one would be implemented and deployed using the presented solution in the form of Swift DSL. Current paragraph is dedicated to the process of implementation and deploying an API using those two solutions. Given the fact that AWS Runtime CDK has been maintained and developed for several years, it presents more instruments to interact with AWS runtime in different ways. But, it also contains much of the business logic required for integrations and development, whereas the proposed solution dispatches most of the work to the underlying implementation that is responsible for establishing the integration and deployment to the specific provider. So, the framework could be quickly extended to support a wider range of AWS specific abstractions through lightweight adapters, that do not require years of implementation and do not actually contain much business logic. But, now the goal of the work was to implement the core cloud terminology in code in an extendable and flexible manner, and ensure the basic support of at least several cloud providers.
So, keeping in mind that fact the following API would operate using the basic building blocks of the AWS infrastructure, to present a fair comparison between the implementations.

The simple API would be intended for a cloud based TO-DO list application and provide several endpoints to fetch all to-do tasks, fetch the specific one or to create a new one. Endpoints specifications are shown below.

- **POST: /task -** Creates a new to-do task with a unique id.

  Parameters:

```
"title": "ToDoTask"
"type": "object"
"properties": {
  "id": {
    "type": "integer"
  },
  "description": {
    "type": "string"
  }
},
"required": ["id", "description"]
```

- **GET: /tasks -** Fetches all available tasks.

  Result:

```
"title": "ToDoTasksList"
"type": "object"
"properties": {
  "tasks": ["ToDoTask"]
 },
 "required": ["tasks"]
```

- **GET: /task/<id> -** Fetches task with the provided identificator.

  Result:

```
"title": "ToDoTask"
"type": "object"
"properties": {
  "id": {
    "type": "integer"
   },
   "description": {
     "type": "string"
   }
 },
 "required": ["id", "description"]
```

5.2 AWS Lambda Runtime Implementation

To start building the API AWS Runtime dependency must be included to the Package.swift file, to build the TO-DO API application target should import AWSLambdaRuntime and AWSLambdaEvents components along with resources config file. Then the lambda function itself must be implemented using AWSLambdaRuntime. For the returned data several mock tasks have been added as a static array. The implementation of the lambda function using AWSLambda runtime is depicted in Figure 22 (code related to JSON encoding/decoding is irrelevant to the subject and was omitted for the sake of simplicity).

```
import Foundation
import AWSLambdaRuntime
import AWSLambdaEvents

Lambda.run { (context,
```

```
            request: APIGateway.V2.Request,
            callback: @escaping (Result<APIGateway.V2.Response, Error>)
-> Void) in
    switch request.routeKey {

    case "GET /tasks":
      // JSON Encoding logic

    case "GET /task/{id}":
       // JSON Encoding logic

    case "POST /task":
       // JSON Encoding logic

    default:
      callback(.success(APIGateway.V2.Response(statusCode:
.notFound)))
    }
}
```

Figure 18. Implementation of lambda function

The implementation of the serverless function is straightforward and intuitive, because of the one-to-one match of serverless function and Swift closure. Such design seems very intuitive to framework clients and fits naturally in the process of development Swift code, in such a way that is familiar for any developer. The closure accepts several parameters which are context, request and callback. The context can be used to retrieve environment variables and constants from the AWS environment.

One of the drawbacks of the listed code that could be outlined is tight integration with underlying implementation of cloud provider, because details of deploying environment get sunk into the function implementation and interface. Not only through imports of AWS CDKs, but also function interfaces are tied up with the APIGateway component of AWS infrastructure in such a way that even version of the APIGateway has to be specified in the interface. However, in fact those interfaces do not introduce any useful insights to the function implementation and it seems that the function should ideally be agnostic of its deployment environment. After creating the implementation of the function and providing mock data, the function must be deployed to AWS manually. Deploying a function to AWS firstly implies creating a Docker image for Amazon Linux 2 OS. To compile the functions using Docker it is reasonable to create a separate script, where the specific compiler and build parameters would be specified.

```
docker run \
  --rm \
  --volume "$(pwd)/:/src" \
  --workdir "/src/" \
  swift:5.3.1-amazonlinux2 \
  swift build --product ToDoList-API -c release -Xswiftc -static-stdlib
```

Figure 19. Creating of a docker image

Script builds source files which reside in the working directory and then docker image is created using the compiled artifacts. After building the Docker image for Amazon OS all created images must be zipped in the archive with special structure and after that it finally would be ready for deployment. The next step is uploading the artifacts to AWS Lambda using the AWS account console. Firstly, users have to create a new Lambda function either from scratch or using some existing blueprint. Creation of the functions is made through the web application UI and requires function name, runtime and optionally specifying permissions. After that the artifact generated by the user must be uploaded to the function. However, there are several problems with this step. First of all it is performed through the UI and which is an error-prone and not reproducible approach, due to the lack of any version management and inability to properly review such changes (discussed in more details in Paragraph 1). For example, when a user has several functions prepared for uploading it may upload the wrong binary to the function. The second, is duplication of data in the UI configuration and in the build parameters for artifacts - the user has to specify name and runtime while building the artifacts and while creating the function in the console. This not only leads to duplication, but also a possible point for desynchronization between build parameters and created functions.

Next step is connecting the function to API Gateway which allows the user to trigger a function using regular REST API. This step involves creating an integration with previously instantiated Lambda, it means that users need to select name, region and stage for the endpoint. But, also users have to manually specify routes to the resources, basically copy-pasting the endpoints which were setted up in the code. Again, such technique is likely to result in subtle errors and typos, not saying that all this setup is straightforward manual job that has to be automated. Such, approach may be suited for small projects where it is easy to control all the changes and problems would be identified quickly, but it's not appropriate for large projects with long testing cycles and many new changes arising. To sum up, the whole process of setting up the Lambda function could be divided into several consequent steps:

1. Writing the function implementation
2. Building a Docker image and packaging it
3. Creating a function through web console and uploading the package
4. Creating an integration with gateway

5.3 Implementation with Swift DSL library

On the other hand, the proposed solution does not require most of the steps involved in the setup with AWS Lambda Runtime and eliminates duplication of information between steps by reusing the code and parameters. The first step stays the same - creating the function implementation. This time function implementation would be almost the same, the only difference is that it will lack AWS entities like Lambda or APIGateway.Response, which makes the function agnostic of deployment environment and hence more reusable for

different providers if needed. And also the handler which is able to send back the API response must be implemented by a client.

```
struct LambdaFunction {
    public let endpoints: [String] = [
      .get("/tasks"),
      .get("/task/{id}"),
      .post("/task")
    ]

    private let requestPerformer: RequestPerformer

    init(requestPerformer: RequestPerformer) {
        self.requestPerformer = requestPerformer
    }

    func run(args: [String: Any]) {
        guard let route = args["route"] else {
            requestPerformer(.success(.notFound))
        }

        switch route {
        case "GET /tasks":
            // JSON Encoding logic

        case "GET /task/{id}":
            // JSON Encoding logic

        case "POST /task":
            // JSON Encoding logic

        default:
            requestPerformer(.success(.notFound))
        }
    }
}


func main(args: [String: Any]) {
    LambdaFunction(requestPerformer: .default).run(args: args)
}
```

Figure 20. Implementation of lambda function with plain Swift

Lambda function presented on Figure 20 simply parses the endpoint route and based on the passed parameters and route performs required logic and sends response back to the client using the RequestPerformer provided in the initializer. RequestPerformer is a generic composite wrapper around standard library URLSession with async/await interface. It provides the mechanism to communicate over HTTP protocol.

The next step would be creating the infrastructure configuration using Swift DSL. That configuration will represent the whole configuration and data from that declaration would be used in AWS to create an appropriate Lambda function. The declaration exemplified on Figure 21 contains all information that is necessary for deploying the serverless function and can be inferred automatically on stage of creating the intermediate representation.

```
import ServerlessComAdapter
import SwiftyServerless

service(name: "example", version: "3.12.0") {
   provider(name: "aws", ignoreCerts: true)

   serverlessFunction(
      name: "ToDoFunction",
      handler: LambdaFunction.self
   ) {
      event(.httpAPI(method: .get, route: "/tasks"))
      event(.httpAPI(method: .get, route: "/tasks/{id}"))
      event(.httpAPI(method: .post, route: "/task"))
   }
}
.serverlessComRepresentation()
```

Figure 21. Infrastructure declaration with DSL

To create the infrastructure declaration, client code must import several packages that are implied to be a part of distribution units from the solution. SwiftyServerless represents the set of objects that build the common foundation of serverless domains such as ServerlessService, ServerlessFunction, etc. That package does not contain any information about the possible intermediate representation and interpretation details. On the contrary ServerlessComAdapter is imported to provide mechanisms for interpreting the DSL components into ServerlessCom representation and provides additional decorators to Serverless DSL. One of such decorators .serverlessComRepresentation() is used in the Figure 21 and provides mechanisms to translate Swift structure into the expected representation. Further, the service component accepts parameters that describe the service properties through the initializer and for brevity's sake some of the parameters have sensible default values. The last required parameter for service initializer is lazy-computed anonymous closure closed over ServerlessServiceComponent type with the signature (ServerlessServiceComponent…) → ServerlessServiceComponent that effectively creates a namespace which restricts the set of available components that could be created inside it. ServerlessServiceComponent is an existential type that represents a box that could contain different implementations that conform to the same requirements. So, ServerlessProvider and ServerlessFunction should conform to ServerlessComponent protocol, but the builder still should be able to retrieve the actual nominal underlying type to provide compile-time

code checks for available methods of the type and code completion. Thus, the ServerlessServiceComponent is represented in the form of protocol with associated type capturing and requires the components implementation to capture the underlying type and have several common functionality - represented on Figure 22. Basically, all other components also create their own scopes built upon generic protocols. Moreover such structure allows for external packages to extend either the protocols or concrete types, using extensions. Example of such extension is adding .serverlessComRepresentation() decorator to ServerlessServiceComponent protocol in ServerlessComAdapter package, that allows to separate logic of interpreting the component to the special package, properly distributing areas of responsibility between code parts.

```
protocol ServerlessComponent {
        associatedtype T: Decoratable
        init(_ type: T)
}

protocol Decoratable {
        associatedtype T

        func decorate(using: Decorator<T>) → T
}

protocol ServerlessServiceComponent:
   ServerlessComponent,
   Decoratable
        where Decoratable.T = ServerlessServiceComponent,
        ServerlessComponent.T = ServerlessServiceComponent {
}
```

Figure 22. Basic generic constraints that build up the components hierarchy

ServerlessFunctions are created from the standalone structures that incorporate some metainformation required to deploy functions. Before referencing in DSL functions have to be registered in the handler container - its implementation is depicted on Figure 17.

```
extension DefaultHandlerContainer {
        public static let shared = DefaultHandlerContainer()

        private init() {}
}

let lambda = LambdaFunction(handler: .default)
DefaulthandlerContainer.shared.register(lambda)
```

Figure 23. Registering function using singleton container

After registering it could be used in the ServerlessFunction component simply just by specifying the type of the handler as demonstrated in Figure 21. The function component DSL builder will then resolve the function by using the singleton (Gamma, E., 1995) instance of the container. Singleton is sometimes recognized as an anti-pattern, but in the

case with a container it seems to be rational to use it, because while it still contains global state, the state access is isolated through actors and the code is forced to use only one single storage of handlers. For some of the underlying intermediate representations it's important to know the absolute path to the handler to invoke it on the cloud platform. This information is also encapsulated by the ServerlessFunction structure by accessing #filePath preprocessor macros and this hides the details of path manipulating from the end user.

Building the module that contains the infrastructure declaration produces the intermediate representation which reflects the infrastructure declared in the code. The example from Figure 21 will produce a .yml file in the form of Serverless.com specification shown in Appendix 2.

Such approach is an effective replacement for steps 3 and 4 of deployment using AWS Lambda Runtime. It allows to get rid of duplication and will be versioned along with the serverless function implementation code. After making declarative infrastructure configurations, the code can be built and will produce intermediate representation files that are used for simplified deployment using Serverless.com in the case represented on Figure 21. After obtaining the intermediate files deployment could be made with a single command 'serverless deploy' and all necessary information would appear in the AWS console. So, in order to sum up the process of deployment a serverless function with Swift DSL library will follow the next steps:

1. Writing the function implementation
2. Writing the infrastructure declaration
3. Deploy the configuration

Although, there is a reduction only in one step compared to the deployment with AWS framework, the steps required by the proposed solution are much simpler and introduce code generation, which lifts some of the extra work required by the other solution. Compared to the implementation with AWS Runtime there is no need to manually build the source files to create Docker images and no requirements for manually packaging such artifacts according to some special structure. So, the overall process of deployment of the serverless function becomes more unified and automated.

5.4 Quantitative evaluation of implementations

Quantitative evaluation is an approach which is used to make evaluations and assumptions based on statistics or other quantitative data. Usually such data is obtained using monitoring or analytics techniques. In case of evaluating the proposed solution - there are several parameters which could be used to perform a quantitative evaluation. Those parameters are gained using built-in monitoring tools from Xcode and VSCode interactive development environments (IDE).

- Number of LOC (lines of code) required to deploy and implement the same serverless function using the proposed framework comparing to the implementation without it
- Function start times for cold and warm start using different implementations

- Number of required dependencies for client code

LOC parameters usually reflect the expressiveness of the software code, because having the same functionality implemented in fewer lines could mean that syntax of the code is more clear and important domain ideas could be identified easier. But, the main notion of that metric is to reflect the effort required to maintain and develop the software. However there are several drawbacks in this metric, more naive code may involve code duplication, but still will be easier to understand than brief and complex generic code, developer experience and language may affect the metric significantly too. In the case of DSL metric may eventually include lines that have no actual meaning, because statements in DSL can be separated using empty line or figure brackets. However, in the case of evaluating small examples of the same functionality used through the paper, such an approach seems to be reasonable and will show the difference between implementations. The LOC results depicted on Table 1 were obtained from the implementation of TODO API discussed in section 4.1.

	Traditional SDK	Swift DSL
OpenWhisk Provider	121	101
AWS Provider	113	87

Table 1. LOC for different implementations

Figure 25. LOC charts for different implementations

According to the charts represented on Figure 8 Swift DSL demonstrated a decrease in the number of lines of code for both examples. In the case of the OpenWhisk provider it showed a decrease of 7% and for AWS the decrease was 14%. Reduction in number of lines does not seem to be significant and with the increase of the project size the

percentage will become even lower, because the serverless function implementation contributes to the metric mostly, while the DSL code required for deploying the code usually will be much smaller and also grow slower than function logic. In some cases there would be even an increase in the number of lines, because like it was with an AWS Lambda Runtime deployment requires mostly actions in UI instead of making any code declarations, but the possibility to track changes and version management for such configurations outweigh the need of additional code, especially keeping in mind that the code is written in declarative manner and tightly related to the domain.

Next parameter that is possible to evaluate is the number of external dependencies that is required by compared implementations. Despite the fact that dependencies are good in terms of reducing the amount of work required by project maintainers and usually provide well designated and tested constructions, they still introduce a point of failure into the software structure. That means the user firstly has to deal with declaring those dependencies and establishing a way of resolving such dependencies using some kind of package manager supported by all dependencies and in complex cases maintaining them could be difficult. Although, the presence of dependencies imposes requirements on execution environments and complicates the process of setting up development tool sets.

In case of AWS Lambda Runtime implementation, developers should explicitly add several frameworks as a package dependencies, those are: AWSLambdaRuntime, AWSLambdaEvents, swift-nio, swift-backtrace and swift-log. Also as far as the source files need to be built as Docker image the implicit dependency on Docker gets introduced. On the other hand, the implementation of the TODO API example using Swift DSL solution introduces several other dependencies too. It requires the client code to explicitly add dependencies for SwiftyServerless and ServerlessComAdapter along with Serverless.com framework as implicit dependency. Dependency graphs for both cases depicted on Figure 9. As could be seen from the graphs, the approach with AWS Runtime framework requires more dependencies to be explicitly stated and managed by the developer in the case with TODO API 6 dependencies against 3.

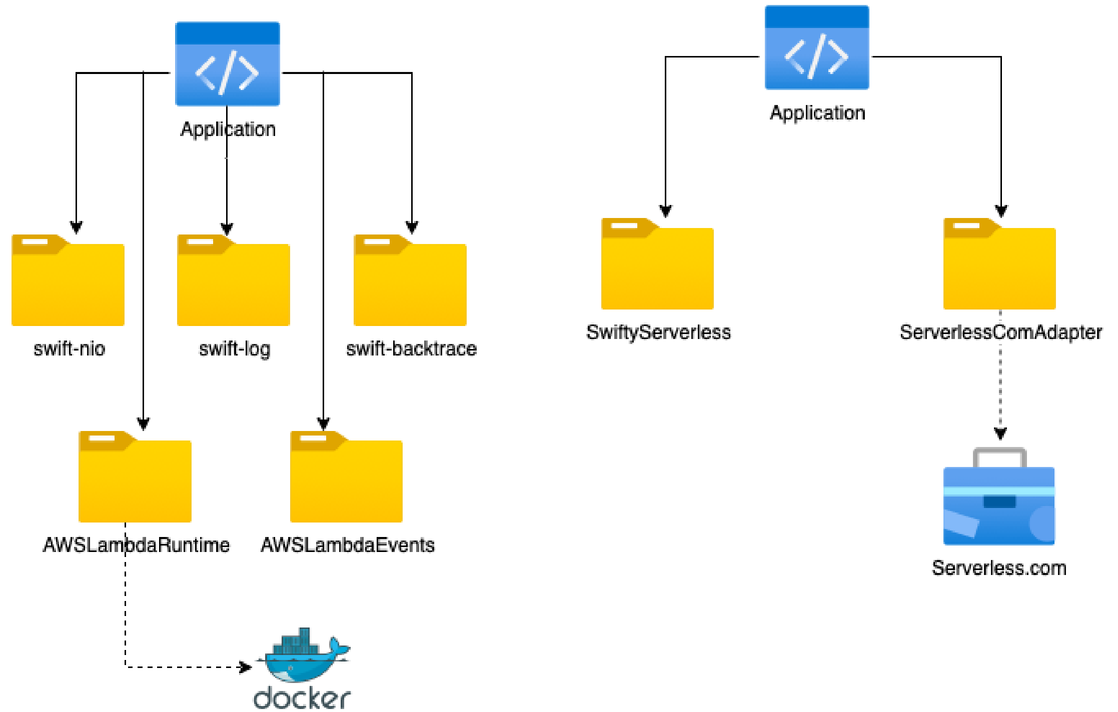
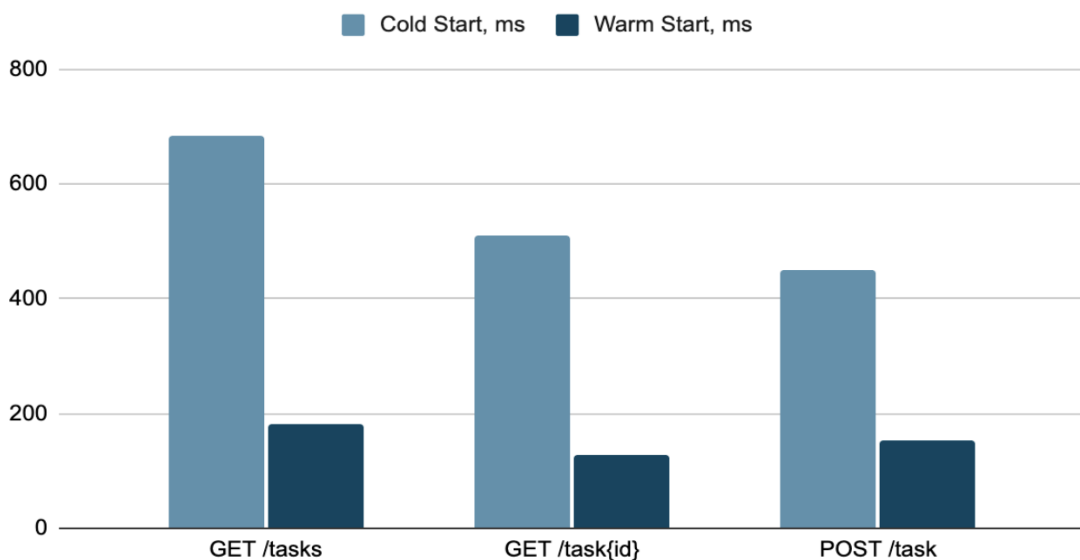
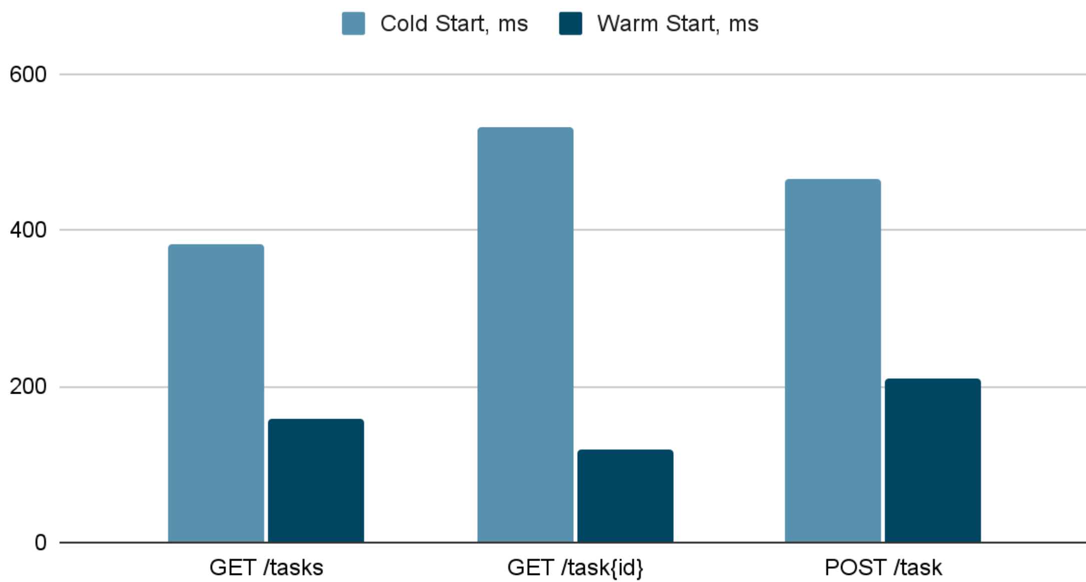


Figure 26. Dependencies graphs

In addition to LOC metrics and dependencies one possible option to compare and evaluate solutions is performance metrics. Performance is a crucial characteristic of a high-load cloud system, but the concerns regarding the most part of performance issues are lifted on to the cloud provider and this one the biggest advantages of serverless approach. However, in terms of deploying serverless function usage of provider-specific SDKs may introduce additional optimizations, which allow for a faster execution of the functions. Charts demonstrated on Figure 10 demonstrate this hypothesis in action.



(a)



(b)

Figure 27. Start times of the function deployed on AWS cloud

The data have been collected using the automated script - to simulate cold starts of the function script invoked the function 40 times with the interval of 1 hour for each case. Obtained data showed that the function deployed with the Amazon Runtime showed 5-10% faster response time compared to the same function deployed with Swift DSL. The reason for that is possibly the usage of pre-compiled binary Docker images, which are built for Amazon OS and optimized to run on top of that OS. However, DSL representation may also automate generation of Docker images and deploying it to the AWS, if that will have a notable effect on the performance. Similar measurements were conducted with functions deployed using OpenWhisk provider Appendix 1, but the measurements did not contain any significant variations.

5.5 Solution evaluation summary

DSL slightly reduces the number of code required to achieve the same functionality, compared to SDK from cloud providers, but makes the deployment process significantly more explicit, because it does not require any separate configuration or custom automation scripts which distracts the understanding of the process and provides unnecessary details. From the performance point of view, functions deployed with the DSL show the similar response times on cold and warm start compared to deployed manually and using SDKs, but in some cases they may lack optimizations achieved using the cloud providers solutions. However, such optimizations also could be added to the DSL by introducing additional layers for optimizations on build time.

6. Conclusion

The solution proposed in the current work currently resides in the development stage and has plenty of features to be implemented. However, based on the observation of existing solutions and conducted evaluations it could be stated that the proposed solution provides a more reliable and convenient way for deploying serverless infrastructure. It already allows to significantly simplify the process of deployment of the serverless function, which was demonstrated in section 4.3 by automating part of the manual job required to deploy functions, also it introduces additional build-time checks and type-safety to the infrastructure configurations. Such checks help to maintain the application code and its infrastructure by keeping them in the same project and written in the same language. Declarative configuration provides a way to abstract clients from underlying details and instruments used to deploy function, is such a way that additional optimizations and instruments could be injected additively without breaking the existing codebase. The fact that the solution may lack some features that are critical for specific projects is mitigated by the presence of intermediate files as build artifacts from the declaration, in critical cases it could be used to manually add necessary fields or options that are still not introduced in DSL. While this introduces desynchronization and will cut off several possible automations, it's a viable way for critical cases and makes the solution more flexible during the development phase.

Despite the solution's advantages there are plenty of problems too, part of them related to the DSL concept as whole, part of them specific to the concrete solution here. The first is tightly related with the nature of DSL. DSL solutions can operate only on a fixed set of instruments provided by developers of the framework, however that set may not be sufficient for all the corner cases emerged during production development and such cases are hardly be envisioned by the framework developers. Eventually, it may be too restrictive for developers to use the library and it would be dropped. The answers to this problem are generated intermediate representations that could be modified independently and modification-encompassing design of the library, which should allow writing custom modifiers for the DSL entities. The second large problem is lack of unified protocol for serverless interactions, that basically means that whereas all the providers operate on the same set of objects and definitions, the actual implementation may differ and that may require additional configuration options per provider. That either could be resolved in a client code using custom modifiers or should constitute an additional abstraction layer which provides the ready instruments for most provider-specific settings. However, as most of the logic is performed by intermediate files consumers, such adapters still would be thin and lightweight.

As could be seen from the problems section there is a lot of room for improving and extending the solution. At the time of writing the framework still cannot provide Swift interface to create and manage resources located on the cloud provider side. But the work is going on implementing the possibility to create AWS tables using its adapter package. Possible extensions and improvements may include more advanced use of type system and eliminating type erasure as much as possible, this may introduce more code duplication for different components, but will move the build-time checks to compile time and will result in better auto-completion options, because the compiler will be able to check all possible invariants. Also more build-time checks for the declaration correctness maye added, for example ensuring that all intermediate files required by declaration were created and reside in expected directories. The most advanced possibilities of improving such a solution are

opened with usage of the common serverless interactions model, which may allow to generate the serverless representation, along with Swift intermediate representations, based on the language AST and encompassing guaranteed language optimizations. That could be a large step towards native support of the serverless computing paradigm.

References

1. CBInsights. (September 2, 2018). Why Serverless Computing Is The Fastest-Growing Cloud Services Segment, from https://www.cbinsights.com/research/serverless-cloud-computing/

2. IBM Cloud. (September 28, 2021). Serverless, from https://www.ibm.com/cloud/learn/serverless#toc-serverless-BzwMo43s

3. Jain, N., & Choudhary, S. (2016, March). Overview of virtualization in cloud computing. In 2016 Symposium on Colossal Data Analysis and Networking (CDAN) (pp. 1-4). IEEE.

4. AWS Amazon. (March 1, 2021). What is AWS, from https://aws.amazon.com/what-is-aws/

5. Azure Products. (n.d.). Azure Cloud Services, from https://azure.microsoft.com/en-us/services/

6. Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., … Iosup, A. (2020). *Serverless Applications: Why, When, and How? IEEE Software, 0–0.* doi:10.1109/ms.2020.3023302

7. Maréchaux, J. L. (2006). Combining service-oriented architecture and event-driven architecture using an enterprise service bus. *IBM developer works*, *12691275*.

8. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., ... & Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.

9. AWS Amazon API References. (n. d.). API references, from https://docs.aws.amazon.com/apigateway/latest/developerguide/api-ref.html

10. Hennessy, J. L., & Patterson, D. A. (2019). A new golden age for computer architecture. Communications of the ACM, 62(2), 48-60.

11. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. Queue, 14(1), 70–93. doi:10.1145/2898442.2898444

12. Morris, K. (2016). Infrastructure as code: managing servers in the cloud. "O'Reilly Media, Inc.".

13. About Swift. (n. d.). Swift, from https://www.swift.org/about/

14. Swift Evolution Async/Await. (June 30, 2021). Swift Evolution GitHub, from https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md

15. Swift Evolution Result Builders. (November 12, 2020). Swift Evolution GitHub, from https://github.com/apple/swift-evolution/blob/main/proposals/0289-result-builders.md

16. Sadowski, M., & Frantzell, L. (2020). The Complete iOS App Using Serverless Swift. In *Serverless Swift* (pp. 121-146). Apress, Berkeley, CA.

17. Evans, E., & Evans, E. J. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

18. CWE-798: Use of Hard-coded Credentials. (June 15, 2010). Common Weakness Enumeration, from https://cwe.mitre.org/data/definitions/798.html

19. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2016). Serverless Computation with {OpenLambda}. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

20. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems. In *Research advances in cloud computing* (pp. 1-20). Springer, Singapore.

21. McGrath, G., & Brenner, P. R. (2017, June). Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (pp. 405-410). IEEE.

22. Gamma, E., Helm, R., Johnson, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

23. Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, *37*(4), 316-344.

24. Morris, K. (2016). *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.".

Appendix 1

```
import ServerlessComAdapter
import SwiftyServerless

service(name: "example", version: "3.12.0") {
   provider(name: "aws", ignoreCerts: true)

   serverlessFunction(
      name: "handleGetTasks",
      handler: GetTasksHandler.self
   ) {
      event(.httpAPI(method: .get, route: "/tasks"))
   }

   serverlessFunction(
      name: "handleGetTaskWithID",
      handler: GetTaskWithIDHandler.self
   ) {
      event(.httpAPI(method: .get, route: "/tasks/{id}"))
   }

   serverlessFunction(
      name: "handlePostTask",
      handler: PostTaskHandler.self
   ) {
      event(.httpAPI(method: .post, route: "/task"))
   }
}
.serverlessComRepresentation()
```

Appendix 2


```yaml
service: example
frameworkVersion: "3.12.0"
configValidationMode: error
useDotenv: true
deprecationNotificationMode: warn:summary


provider:
 name: aws
 stage: dev
 region: us-east-1
 profile: dev
 runtime: swift5
 memorySize: 1024
 timeout: 10
 ignoreCerts: true

functions:
 HandleGetTasks:
   events:
     - httpApi:
       method: GET
       path: /tasks

 HandleGetTaskWithID:
   events:
     - httpApi:
       method: GET
       path: /tasks/{id}

 HandlePostTask:
   events:
     - httpApi:
       method: POST
       path: /task
```