POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# Nubes+: Fault-Tolerant Object-Oriented Programming for Stateful Serverless Functions

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Author: **Sanja Kosier**

Student ID: 930834
Advisor: Prof. Alessandro Margara
Academic Year: 2022-23

# Abstract

The development of cloud computing significantly changed the technological environment and business models of tech companies. Cloud computing and the serverless model are becoming more and more popular due to their availability, and convenient pay-as-you-go model. Until today, they have reached a large number of applications, allowing developers to spend more time focusing on their product development rather than infrastructure management. However, despite these advantages, managing the state in serverless environments is complex and requires careful design from developers to achieve effective execution.

This thesis tackles the issues of stateful serverless services (SSFs) that preserve their state on external storage. The main problems highlighted in the context of SSFs are state management and fault tolerance. State management introduces complexity and overhead and requires developers to design database interactions carefully. This thesis presents Nubes+, an improved version of the Nubes programming model that adds object-oriented abstractions to serverless programming with an exactly-once guarantee. Nubes+ improves the robustness, consistency, and reliability of Nubes by addressing the problem of unreliable executions.

**Keywords:** Fault tolerance, cloud computing, serverless, Function as a Service

# Abstract in lingua italiana

Lo sviluppo del cloud computing ha cambiato in modo significativo l'ambiente tecnologico e i modelli di business delle aziende tecnologiche. Il cloud computing e il modello serverless stanno diventando sempre più popolari grazie alla loro disponibilità e al conveniente modello pay-as-you-go. Fino ad oggi hanno raggiunto un gran numero di applicazioni. Gli sviluppatori trascorrono il tempo concentrandosi maggiormente sullo sviluppo del prodotto piuttosto che sulla gestione dell'infrastruttura. Tuttavia, la gestione dello stato in ambienti serverless è complessa e richiede un'attenta progettazione da parte degli sviluppatori per ottenere un'esecuzione efficace.

Questa tesi affronta i problemi causati dai servizi stateful serverless (SSF), che preservano il loro stato su storage esterno. I principali problemi evidenziati nel contesto di gli SSF sono gestione statale e tolleranza ai guasti. La gestione dello stato introduce complessità e sovraccarico e richiede agli sviluppatori di progettare attentamente le interazioni del database. Questa tesi presenta Nubes+, una versione migliorata del modello di programmazione Nubes che aggiunge astrazioni orientate agli oggetti alla programmazione serverless con una garanzia esattamente una volta. Nubes+ migliora la robustezza, la coerenza e l'affidabilità di Nubes risolvendo il problema delle esecuzioni inaffidabili.

**Parole chiave:** Tolleranza agli errori, cloud computing, serverless, Function as a Service

# Contents

**Abstract** **i**

**Abstract in lingua italiana** **iii**

**Contents** **v**

**Introduction** **1**

**1 Background** **3**

1.1 Cloud Computing . . . 3
1.2 Serverless vs Function-as-a-Service . . . 4
1.3 Faults, Failures and Fault tolerance . . . 6
1.4 Object-Oriented Programming . . . 6
1.4.1 Golang and OOP . . . 8
1.5 Used Technologies . . . 9
1.5.1 Nubes . . . 9
1.5.2 Beldi . . . 11
1.5.3 Amazon Web Service . . . 13

**2 Implementation Details** **17**

2.1 Architecure and General Overview . . . 17
2.2 Differences with Nubes . . . 19
2.2.1 Database - tables and queries . . . 20
2.2.2 SSFs, handlers and client library . . . 25
2.2.3 Generator . . . 27
2.3 Limitations . . . 28
2.4 Lifecycle . . . 29

**3 Evaluation** **31**

3.1 Testing setup . . . 31

3.2 Testing models . . . 33
3.3 Environment setup . . . 33
3.4 Performance review . . . 34
3.5 Discussion . . . 36

**4 Related Work** **39**
4.1 Serverless Workflows . . . 39
4.2 Fault tolerance for stateful serverless systems . . . 40

**5 Conclusions** **43**
5.1 Review and Critique of Beldi library . . . 43
5.2 Future Work . . . 46
5.3 Overall Conclusion . . . 47

**Bibliography** **49**

**List of Figures** **55**

**List of Tables** **57**

**Listings** **59**

**Acknowledgements** **61**

# Introduction

Cloud computing profoundly changed the technological world and how tech companies base their business model. The use of resources and resources of great power has never been more accessible and more affordable. It reached the point where it would be almost impossible to count and list all its use cases. Developers are more focused on building their products rather than managing infrastructures.

The serverless model that cloud computing offers opened doors to a new set of possibilities. Due to their huge possibilities and the benefits they offer and come with, their usage has been included in the flow that requires storing and managing state. State management in serverless environments brings a new layer of complexity, and its effective execution is subject to innovative solutions.

## Problem Statement

Although serverless functions offer many benefits, they have certain limitations that need to be considered if one proceeds with their usage for development. The state is one of them. Stateless serverless functions, as their name says, do not persist state between their invocations. They can use in-memory storage or may have access to temporary local file systems and write to disk. Still, this type of storage is only ephemeral and unsuitable for cross-function invocations. To explain this more, we must put it in the context of distributed systems and cloud platforms. Developers do not have control over creating or removing new containers; hence, one can not know if the function invocation will have access to the same local disk. Restart is also a valid use case that may happen. Thus, since they can not rely on local storage, serverless functions are viewed as stateless as they are not designed to persevere the state between invocations.

On the other hand, stateful serverless functions (SSFs) rely on external storage solutions to manage and preserve state across multiple function invocations. In the context of the stateful serverless function, we will point out two main problems: state management and fault tolerance.

Managing the state adds a layer of complexity and overhead. Developers must carefully design and model database interactions to perform them correctly. Managing a state often means using storage services, which come with additional costs and is often important factors that need to be considered. Due to state management, SSFs can be viewed as clients of storage services rather than stateful service itself when putting stateful serverless functions and storage services in the same context. Since the stateful function executions are dynamic and distributed, exactly-once execution presents itself as a main challenge. External factors like network issues, infrastructure failures like an individual machine and data center failures, scheduled downtimes or maintenance, hardware resources, electricity, power supply, and transient errors can affect the execution flow. All of them cause risks of duplication or unintended repetitions. Suppose the exactly-once execution is not guaranteed, and the risk of repetition is present. In that case, faults can lead to data corruption, inconsistent data states, unreliable executions, invalid data consumption, and make the whole application flow unreliable.

## Thesis Objectives and Outline

The aim of this thesis is to enrich an already existing programming model that uses stateful serverless invocations with exactly-once guarantee. We tackled the issue of unreliable executions that may occur in Nubes, an already existing programming model that introduced object-oriented abstractions into serverless. Nubes hides all the state management from the developers and, using object-oriented concepts, allows them to represent the state in terms of object types. The state can be further accessed and changed by calling the object methods defined by the developers.

By addressing the problem of unreliable executions and providing exactly-once semantics, we enriched Nubes, gave it an additional feature, and packed it under the name Nubes+. In Nubes+, we improved Nubes by making it more robust, reliable, and consistent.

This thesis is divided into five chapters. It starts with Chapter 1, which presents the background and technological context. Chapter 2 gives implementation details, starting from the architectural overview. Chapter 2 describes the differences between Nubes, the limitations we encountered, and how one can develop with Nubes+. Chapter 3 analyses the performances of the developed solution. Chapter 4 presents related work in fault tolerance for stateful serverless functions and serverless workflows. Finally, Chapter 5 concludes the findings and highlights potential avenues for future research.

# 1 | Background

This chapter presents the background and technological context to understand how Nubes+ operates. It starts by giving an overview of cloud computing, the serverless model and the service concept called Function-as-a-Service. Understanding such concepts is necessary to comprehend the thesis's goal. The chapter follows by giving a short introduction to faults, failures, and fault tolerance to explain the term from the title and its challenges. Lastly, the chapter provides a concise overview of object-oriented concepts and puts them in the context of the Golang programming language.

## 1.1. Cloud Computing

Cloud computing [30, 33, 35, 44] is the on-demand delivery of computing resources. Computing resources, including storage, software, servers, networking capabilities, computing power (and many others), are delivered as services over the Internet. It generally comes with a pay-as-you-go pricing model.

The pay-as-you-go pricing model means paying only for those resources used for as long as they are used. When the client stops with the service usage, the billing also stops. There is no need for long-term contracts or licensing. It is important to note that the pricing depends on the cloud provider and differs from one cloud provider to another. The prices for using different services are also mutually different and are determined by the provider. The pay-as-you-go pricing model is convenient, often helps lower operating costs and is one of the convincing reasons to migrate to the cloud.

Cloud computing offers its users many benefits. Firstly, it is accessible. To access their cloud services, users need only an internet connection. It is agile as it comes with a wide range of different services and technologies that can be provisioned and used within minutes, with just a few mouse clicks. It is elastic when it comes to resource provisioning. If one is unsure about the right amount of resources needed, there is no need for over-provisioning and purchasing great capacities that will not be used. Instead, resources can always be scaled up or down later depending, e.g., on the spikes or dips in traffic.

Cloud offers global deployment. As cloud providers have infrastructures all over the world, underlying physical resources provided in physical data centers are abstracted in so-called cloud regions. Users have the possibility to deploy their applications in regions that cover different parts of the globe and provide application users in these areas with lower latency and, consequently, better user experience. Also, in other cloud regions, users can provision different amounts of service resources.

There is no need to build and buy all the necessary hardware, software, data centers, and all the other equipment needed for their management. Users can provision resources instead and pay for their usage following the mentioned pay-as-you-go pricing model, which presents itself as another benefit.

Because of its benefits, there is a wide variety of cloud computing use cases, and it would be impossible to count them all. With cloud services from a cloud provider, one can create web and mobile applications that can be easily deployed and scaled. Many people keep their phone and laptop data in cloud storage. With the cloud, USBs and external hard disks are no longer needed to transfer data from one user's machine to another. Organizations often use cloud computing for application development, data storage, disaster recovery, or, in other words, to safely back up digital assets, for infrastructure scaling, etc.

Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) are three main types of cloud computing services. They offer different levels of flexibility, control, and management.

## 1.2. Serverless vs Function-as-a-Service

The term serverless, although it can give the impression there is no server running underneath, is a cloud computing application model where developers and users of serverless services do not have to think about and handle server management since cloud providers abstract all the underlying infrastructure [28, 46]. Server management and interaction are entirely hidden from the developer. Developers deploy their code to containers fully managed by cloud providers where the cloud provider takes responsibility for all infrastructure management and maintenance such as OS updates and patches, system monitoring, security management, capacity planning, etc. Using serverless functionalities allows developers to focus more on the code and application development and speeds up the development process.

Other benefits of serverless functionalities are built-in availability, automatic scaling de-

pending on the traffic, and a pay-as-you-go pricing model. When the code executes, the cloud provider provisions needed resources and scales the infrastructure if needed. Once the execution ends, resources are scaled down (so-called *scaling to zero*), and pricing stops. There are no idle resources and no unnecessary costs. The cost usually depends on execution time and the amount of resources used.

Function-as-a-Service (FaaS) [45] is the central technology in serverless architecture. Although it is often used as a synonym for serverless, FaaS is, in reality, a subset of it. Serverless can be any service category whose configuration and server management are handled by the cloud provider and hidden from its users [46]. Serverless database and storage, API gateways, event streaming, and messaging are some of them.

As mentioned, FaaS is the main serverless technology and comes with its benefits. It is event-driven, which means code execution runs as a response to events (e.g., file uploads, database changes), requests (HTTP requests), or other configurable triggers.

FaaS has many usages. Their true strength lies in the serverless benefits that transactions can be easily separated and scaled, which makes them a good fit for high-intensity workloads that can be completed in parallel. They are widely used in data processing pipelines as well. It is a good practice to design a function to perform only one action to respond to an event. Following this approach, developers design and chain several functions in a processing pipeline. Apart from those, FaaS finds its usage in web development, machine learning and microservice architectures, where each microservice is represented as its function.

All leading cloud providers come with FaaS platforms: Amazon Web Services [20] has AWS Lambda [25], Google Cloud [41] has Google Cloud Functions [42], IBM Cloud [43] has IBM Cloud Functions [36] and Microsoft Azure [47] has Azure Functions [32].

Once developers have their code ready, they need to upload it to the FaaS platform of their cloud provider, usually as a `.zip` file or container image. Invoking the function depends on the cloud provider and configured type of event or trigger.

Depending on external storage usage, functions can be stateless or stateful. We mentioned peculiarities and differences between stateless serverless functions and stateful serverless functions earlier in the problem statement. Stateful serverless functions most commonly keep state in the low latency database of their cloud provider [55]. AWS Lambda serverless functions store state in DynamoDB, BigTable serves as state storage in Google Cloud Functions and Azure Functions use CosmosDB.

## 1.3. Faults, Failures and Fault tolerance

Let's start by connecting the terms from the heading. A system fails when it is not able to provide the services it was designed for. Failure is the result of an error that is a part of a system's state and the cause of an error is called a fault [53]. A system that can provide its services adequately and continue to operate even in the presence of fault is said to be fault-tolerant.

For a system to be fault-tolerant, it needs to meet dependability requirements. Some dependability requirements are availability, reliability, safety, maintainability and security. An available system is a system that is able to, or better, that is available to provide its services as soon as it is needed. Reliability describes how reliable the system is during an interval of time. It is important to emphasize a different time-measuring scale here, as a system can be highly available but very unreliable. For example, if a system goes down every hour for one millisecond, it has an availability of $(60*60*1000-1)/(60*60*1000) = 99.9999\%$, but the fact it fails every hour makes it very unreliable.

Safety answers the question of how safe it will be if the system fails. For example, aviation systems must have a high level of safety guaranteed. Maintainable system is a system that can be easily fixed in case of failures. Security is often connected with the integrity of the data system provides. The main technique used to deal with failures is redundancy.

Let's say that the system is able to continue to operate in the presence of failures. Still, if the failure happens, the component where the failure occurred needs to recover to a correct state. Recovery from an error is fundamental to fault tolerance.

How to deal with failures and approach fault tolerance is a broad topic and depends on the types of failures that may occur in the system and its function in general.

## 1.4. Object-Oriented Programming

Object-oriented programming is a programming paradigm based on the concept of objects. The fundamental idea behind it is to combine both data and the functions that operate on that data in a single unit called an object [7]. We say that an object is an instance of a class, as classes are used to define objects. Let us briefly describe key concepts around object-oriented programming.

Unlike some other programming paradigms, where development starts from organizing a code into functions, in the object-oriented paradigm development starts from object modeling. As in real life, objects have their attributes and behavior. To emphasize, the

fundamental idea around objects is that they combine, or better, encapsulate the two together, as by themselves neither data nor functions would model real-world objects in an effective way. Encapsulation of data and methods that operate on that data is essential for several reasons. It hides internal implementation details and the object's state from external access and instead exposes an interface to interact with it (the object's public methods). Moreover, it provides control over data access and modification as methods can be declared as e.g. read-only.

The importance of the inheritance concept in the object-oriented paradigm can be seen in the fact that Bjarne Stroustrup, also known as the father of C++ programming language, in the conclusion of his paper called *What is "Object-Oriented Programming"?* states: *Object-oriented programming is programming using inheritance* [52].

Object classes, just like in real life, can share common characteristics. They can take the position of superclass (base class) or subclass (derived class). For example, an animal is a superclass of a lion, a tiger, and a giraffe. By deriving subclasses from the superclass, classes are put into a hierarchy and are inheriting the attributes and behavior of their superclass. This way, derived classes can use methods included in the base class and do not need to define them again. Inheritance promotes code reusability and is one of the core principles of good object-oriented programming.

The last core feature of object-oriented programming we will mention is polymorphism, which relates to Barbara Liskov's Substitution Principle [10]. The Liskov Substitution Principle is one of the five SOLID principles of object-oriented programming, it puts the requirement on the subclass and is formally defined like this:

***Subtype Requirement:*** *Let $\phi(x)$ be a property provable about objects $x$ of type $T$. Then $\phi(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.*

In other words, an instance of a subclass should be able to replace the superclass object, and the correctness of the program should not be affected. Supporting the Liskov Substitution Principle is usually done through polymorphism. As we mentioned for inheritance, the superclass contains methods common to all derived classes in the hierarchy. However, a single derived class may be able to override a method with its own implementation. That method then becomes polymorphic, and its action depends on the class of object it is applied to. Thus, through method overriding and interfaces, polymorphism allows objects of different classes to be treated as objects of a common superclass. Polymorphism allows us to write and design more flexible, more maintainable object-oriented systems with higher levels of abstraction.

Some of the most common programming languages that support object-oriented paradigms are Java, C++, C#, Python and others.

### 1.4.1. Golang and OOP

Nubes was written in the Golang [40] programming language; therefore, Nubes+ as well. One can not say that Golang is a purely object-oriented language, as it misses some of the core concepts we mentioned previously. There are no classes. Developers can define data types as structures and associate methods to define their behavior. However, this concept is different from having an object class with its methods. Inheritance is also not possible; instead, Go promotes object composition. Another peculiarity is that the type does not explicitly state it implements an interface but implicitly if it implements all the methods stated in the interface contract. Interfaces are used to provide a level of abstraction as polymorphism is possible in Go.

Go uses a combination of naming conventions and visibility rules to achieve encapsulation [39]. Methods and structure members that start with capital letters are *exported*, which means they are accessible from outside of the package. On the other hand, structure attributes that begin with lowercase letters are *unexported* and their scope is limited to package visibility. Listing 1.1 demonstrates the mentioned peculiarities.

```go
type Vehicle interface {
    GetMaxSpeed() int
}

type Car struct {
    name     string
    MaxSpeed int
}

func (c *Car) GetMaxSpeed() int {
    return c.MaxSpeed
}
```

**Listing 1.1:** Example of naming convention and encapsulation in Go

We defined a type *Car* with two member attributes and one method. Type implements interface *Vehicle* as it implements a method *GetMaxSpeed* defined in the *Vehicle* contract. Between two member attributes, *name* and *MaxSpeed*, *MaxSpeed* is exported and can be accessed from outside of the package. On the other hand, the member attribute *name* is

not accessible from outside the package as it is unexported.

We want to note that Go supports multiple return values and encourages developers to use explicit error handling. Usually, an error is returned as a last return value.

## 1.5. Used Technologies

This section covers the most important technologies and cloud services used for the purposes of this thesis.

### 1.5.1. Nubes

Nubes [11] is an object-oriented programming model for serverless computing. The model translates the defined types into serverless functions that may be automatically deployed into the client's AWS account. In addition, all the state management is hidden from the developer. It generates a client library that acts as an interface for interaction with the stateful serverless functions made using Nubes by exposing the previously defined object types. This thesis enhances Nubes with fault tolerance, therefore, we will provide a detailed overview of the model.

#### Object relationships

It is possible to define object relationships with Nubes. Objects can be in one-to-one, one-to-many and many-to-many relationships. Relationships can be unidirectional and bidirectional. The developer must use the appropriate reference type defined in the Nubes library to define object relationships. *Reference* and *ReferenceList* are used for unidirectional declarations. *ReferenceNavigationList* is used in bidirectional relations. Furthermore, to define the direction in bidirectional relationships, the developer must put a proper tag next to the type attribute, either *hasMany* or *hasOne*.

```
1 func (r ReferenceNavigationList[T]) GetIds() ([]string, error)
2 func (r ReferenceNavigationList[T]) Get() ([]*T, error)
3 func (r ReferenceNavigationList[T]) GetStubs() ([]T, error)
4 func (r ReferenceNavigationList[T]) AddToManyToMany(newId string) error
5 func (r ReferenceNavigationList[T]) DeleteBatchFromManyToMany(ids []
    string) error
```

**Listing 1.2:** Methods defined in *ReferenceNavigationList* object type.

## Methods

Certain rules must be followed when defining custom methods in Nubes. All methods must return an error. Methods can return up to one optional return value. If the method returns a value, the value must be returned as the first value, error as the second. Methods can be defined with up to one input argument. The method can both have an input argument and return a value. The method needs to be defined as an object method and needs to be exported.

```
1 func (s *<ObjectName>)<MethodName>() error
2 func (s *<ObjectName>)<MethodName>(<InputArgument>) error
3 func (s *<ObjectName>)<MethodName>() (<ReturnType>, error)
4 func (s *<ObjectName>)<MethodName>(<InputArgument>) error
5 func (s *<ObjectName>)<MethodName>(<InputArgument>)(<ReturnType>, error)
```

**Listing 1.3:** Allowed method signatures.

## Generator

The generator has an essential role in developing with Nubes. Once the developer finishes with the type definitions, the generator steps in. The generator is a *command line interface* (CLI) that parses Go files where the developer has defined types into *abstract syntax tree* (AST) structures. Once the files have been parsed, the generator puts the correct logic in place to determine, e.g., which handler function needs to be generated, where the additional logic needs to be put in place, which databases need to be created, etc. It understands mutual object relationships, which are necessary to provide the proper methods for object handling. Once the files have been parsed, generator prints translated types' definitions, serverless handlers, deployment file, or the client's library, depending on the input flags.

The generator expects one of the two modes to be provided as a first input parameter: *handlers* or *client*. Other possible parameters are:

- -*t* path to folder with types definitions
- -*o* path to output folder
- -*m* module name of the project
- -*g* boolean flag that indicates if the deployment files should be created
- -*i* boolean flag that indicates if the database tables should be created

### Additional features

Nubes offers additional features that can be annotated with the appropriate *tag*. The tag must be annotated next to an attribute value in the type definition. Attributes can be tagged as *read-only* or used as a custom ID. Both can be achieved via adding *'nubes: "id,readonly"'* next to the attribute declaration.

## 1.5.2. Beldi

Nubes+ relies on the use of the Beldi library to ensure fault tolerance guarantees. Beldi is a library and a runtime system for writing and composing fault-tolerant and transactional stateful serverless functions [55]. Stateful serverless functions, although they can preserve the state themselves, usually preserve the state in the database of the local cloud provider. From this perspective, SSFs are not so much stateful services as they might be seen as clients of scalable, fault-tolerant storage services. Beldi's purpose is to guarantee exactly one semantics for SSF processes that fail in execution in this type of client-service communication, where clients are SSF workflows.

Beldi's API is one of its four main components. Beldi offers read and write methods for database interaction purposes as well as a *condWrite* method that writes to the database only if the condition given as a parameter is true. Beldi offers synchronization mechanisms like transactions; thus, operations between beginning and end transaction calls enjoy ACID semantics. Beldi allows its users to chain SSFs in a workflow as it comes with synchronous and asynchronous lambda function invocations (*syncInvoke* and *asyncInvoke*). Let's emphasize that all the complexity of logging, replaying, and concurrency control protocols that enable exactly-once execution and transaction semantics are hidden from the developer. Once the SSF that is using Beldi's API is executed, it can automatically determine the context in which it was executed, e.g., is it part of the transaction, what is its step number, etc.

Another component of Beldi's is a set of database tables where logs of database interactions and function invocation are stored, as well as their state. Beldi keeps four logs for each SSF. The result of every read operation is kept in the *ReadLog*. *WriteLog* describes performed write operation and is stored in the same table as data being written to avoid cross-table transactions. The *InvokeLog* is a Beldi novelty and ensures at-most-once semantics for calls to other SSFs. The *IntentLog* describes the completion status of an instance ID, its return value, arguments, type of invocation and timestamp from the garbage collector.

The last two components of Beldi's are Intent Collector (IC) and Garbage Collector (GC). Both are stateful serverless functions periodically triggered by a timer. The purpose of IC is to scan SSF's intent table and check if there are instances that have not yet finished, i.e., ones missing the *done* flag in the completion status. If so, IC restarts those unfinished SSFs with the original arguments and instance ID. Even if the actual instance is still in the execution process, due to Beldi's *IntentLog* that ensures at-most-once semantics for each step of the SSF, SSF restart is safe to perform.

Beldi guarantees exactly-once semantics by extending the log-based approach from Olive [51]. Olive introduced Distributed Atomic Affinity Logging (DAAL), a technique that puts an item's log entries in the same atomicity scope as the item's data. Its purpose was to make performing operations (e.g., writing to the database) and intent logging atomic. Once the client performs those two operations atomically, at-most-once execution semantics can be ensured. To bypass database limitations in the atomicity scope and to allow Beldi to be compatible with and support all standard databases, Beldi introduces a new structure called linked DAAL. Linked DAAL enables logs to exist on multiple atomicity scopes. They are types of non-blocking linked lists and can be traversed as such. A few optimizations are put in place here. To reduce the overhead of unnecessarily returning the full content of each log row, Beldi applies projection and filters out all unneeded columns. In order to keep all the logs and linked DAALs from growing indefinitely, GC prunes old rows and log entries, ensuring linked DAALs are kept shallow. This way, Beldi prevents overheads and unwanted storage costs.

As already mentioned, apart from execution guarantee, Beldi offers its users synchronization mechanisms such as transactions and locks to prevent unsafe handling of state in case of concurrency. To give a more detailed overview, Beldi borrowed another approach from Olive, called *Locks with intent*, where not a client, but intent is the one owning the data locks. In case of SSF crashes, the lock will not be instantly lost. Instead, IC will restart the instance as it will miss the *done* flag in its completion status. Upon reaching the lock (item) part, the restarted instance will detect it has already acquired the lock and will continue carrying out the remaining operations. Beldi goes with opacity as the isolation layer in the context of transactions.

Although Beldi allows developers to build fault-tolerant and transactional workflows of stateful serverless functions on existing serverless platforms, it should be noted that preventing deadlocks, infinite loops and other unwanted behaviors is the developer's responsibility.

### 1.5.3. Amazon Web Service

Amazon Web Services [20] is a cloud computing platform developed by Amazon. It is one of the leading and most influential cloud providers in the cloud computing industry. It offers a wide range of services (more than 200) and tools for computing power, data storage, analytics, data processing, machine learning, etc. Their data centers are distributed worldwide, making low latency and high availability essential characteristics of their services. Because of using a pay-as-you-go model, cost is optimized, and their users pay only for the resources they consume.

One of the advantages of using AWS services is the same as that of cloud computing: users are free from worrying about where their data is physically kept or how to interact with the servers that run their applications. AWS handles the entire hardware management, which is already in place and ready to be used.

Due to the broad range of service frameworks and programming languages supported, AWS enables a wide variety of developers to integrate third-party services with the services they offer, which also explains the reason behind being so widely used.

Nubes+ deeply relies on Amazon Web Services. Both components that make Nubes+, namely Nubes and the Beldi library, use the features of Amazon Web Services under the hood. The services that Nubes+ uses are listed below.

#### Amazon DynamoDB [3, 16, 37]

DynamoDB is a fault-tolerant, low-latency, NoSQL database. Since this thesis is greatly based on fault tolerance, it is worth discussing how DynamoDB generally manages fault tolerance.

Fault tolerance, but also high availability and durability of data, are achieved in several ways [48]. The first of them worth mentioning are replication and redundancy of data. Data in DynamoDB is automatically replicated across many Availability Zones (AZs), geographically distributed data centers. To maintain data redundancy and high availability even in the case of AZ failures, each write is concurrently written to many copies inside an AZ. In case of failure of one AZ, the other can continue to serve requests.

Besides data replication within multiple AZs, DynamoDB provides an option for global tables. As the name says, global tables allow the user's DynamoDB tables to be replicated across multiple AWS regions, thus allowing fault tolerance during (unlikely) regional outages. This is not the only benefit, however. In environments of large-scale applications with massively scaled applications with globally distributed users, global tables enable

applications to stay highly available and deliver low-latency data.

DynamoDB uses a quorum-based strategy to keep data consistency and fault tolerance against replica or node failures. In a quorum-based strategy, an operation writing or reading the data is considered successful when a certain number of replicas (quorum) acknowledge the operation. However, DynamoDB uses a so-called *sloppy quorum* where all read and write operations are not necessarily performed on the first $N$ encountered nodes but on the first $N$ healthy nodes, where $N$ is the decided quorum. This approach does not reduce durability during server failures or other failure conditions.

In case of data corruption or clumsiness of the user, DynamoDB provides point-in-time recovery [18]. Users can restore their table data to the previous state (based on time and date) within a 35-day retention period. In the case of transient errors caused by network issues or similar, DynamoDB puts a retrying mechanism with exponential backoff and jitter in place. Once the system is up again, the next retry attempt is expected to execute successfully.

There are a few more ways DynamoDB assures fault tolerance. Automatic partitioning distributes data across several servers to maintain high performance, and load balancing ensures that requests are spread equally throughout the partitions and prevents overload on specific nodes [17]. Users can configure automatic monitoring tools on CloudWatch and get notified when something is wrong, as well as manually check DynamoDB dashboards and track service health.

In Nubes+, we are using DynamoDB to store our data. Therefore, its core components are worth mentioning. Those are tables, items, and attributes. A table is a collection of items with a concept similar to rows in other database systems. Similarly, an item is a collection of attributes, and those can be compared to columns or fields in other database systems. For the purpose of this thesis, we will use the official nomenclature: tables, items and attributes.

### AWS Lambda [25, 26]

AWS Lambda is a serverless computing service and is Amazon's implementation of the FaaS (Function-as-a-Service) model. In Nubes+, we are using AWS Lambda to deploy all of our functions required for handling an object's state; in other words, we are using it to deploy our stateful serverless functions. When an end client of Nubes+ calls object functions, they invoke the deployed lambda function in their AWS account. Since we are saving an object's state in DynamoDB, we are integrating two AWS services, making our architecture more complex.

## Amazon S3 [19]

Amazon Simple Storage Service, or simply Amazon S3, is, as the name says, an object storage service. It allows users to store and retrieve any amount of data. Users can upload data manually, use S3 to store internal cloud trail logs of their AWS account, or build applications and store data automatically. Prior to inserting data, users need to create a bucket, a place where objects are stored. We use Amazon S3 during the function deployment phase. Once we build function executables and compress them, we store them in an S3 bucket together with a compiled cloud formation template file in JSON format.

## Amazon CloudWatch [13, 14]

We mentioned Amazon CloudWatch when talking about configuring monitoring tools for DynamoDB. Needless to say, it is not limited only to DynamoDB. CloudWatch is a service used for monitoring and observability. It collects real-time metrics, logs, and event data from various AWS resources. Users can set alarms and receive notifications if a certain metric threshold is reached. With Nubes+, we use CloudWatch Logs to monitor and store log files from our AWS resources. We integrated it with deployed lambda functions and during the deployment phase, we create a log group [15] per each deployed function. We used CloudWatch Logs mostly during the development phase for troubleshooting and optimization.

## AWS CloudFormation [21, 22]

CloudFormation is a service that helps users model, provision, and configure all of the AWS resources they need. Users create a CloudFormation template that describes the resources they want, and CloudFormation then provisions and configures them, freeing users from the manual setup. In order not to provision all resources and services needed for Nubes+ individually and manually, we used the AWS CloudFormation template. We also grouped our resources into a stack [23] to manage them as a single unit. An approach where one describes needed infrastructure is called Infrastructure-as-Code (IaC) and comes with many benefits. Some of those are automation, repeatability, consistency, and version control.

## AWS IAM [24]

AWS Identity and Access Management is a service for securely managing identities and access to services and resources. Core components are users, groups, roles, policies and permissions. Since we are building Nubes+, an abstraction layer that will use and create

resources on other's AWS accounts, it is essential to configure security permissions properly and accordingly. Considering we are integrating a few AWS services with Nubes+, we are provisioning roles that grant permissions to services to communicate with each other as needed.

# 2 | Implementation Details

To gain a deeper understanding of the differences between Nubes+ and Nubes, specifically, the changes made and the reasons behind them to ensure exactly-once execution guarantees, the chapter begins by providing a general overview of the architecture. The subsequent section offers a detailed overview of all introduced changes, followed by a section that discusses encountered limitations. Finally, the last section explains the lifecycle process of the development with Nubes+.

## 2.1. Architecure and General Overview

A high-level architectural overview of Nubes+ and its related components is presented in Figure 2.1. The full execution flow includes several components. The client user has the role of the main actor. The Nubes+ abstraction layer serves as a library interface for the client user. AWS serves as a cloud provider. Stateful serverless functions are deployed on AWS Lambda and since stateful, storing and retrieving the state is obtained from DynamoDB, that serves as the main database.

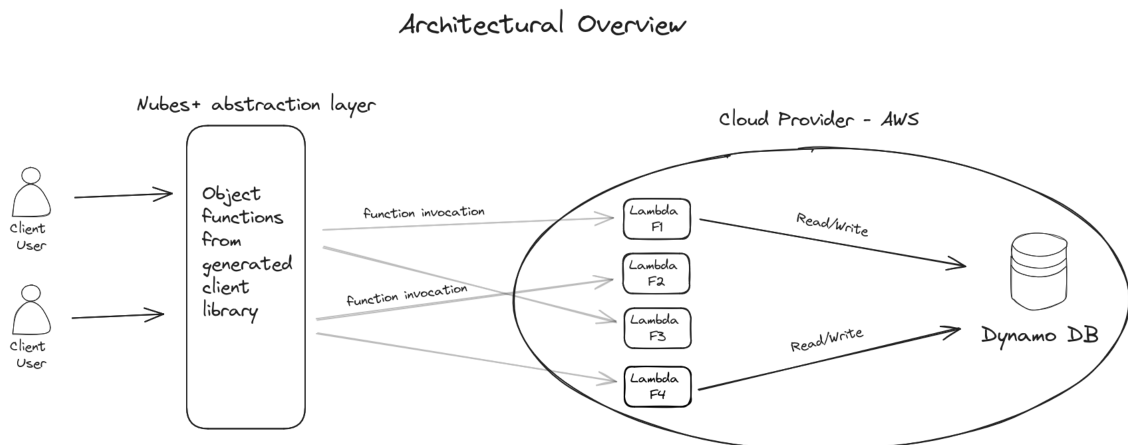


Figure 2.1: Architectural overview of an execution flow.

The flow begins with the client user calling the object functions from the client library

generated with Nubes+. Under the hood, the function performs an invocation to the Lambda method deployed on the client's AWS account. Deployed serverless functions are stateful; thus, when needed, Lambda performs read and write operations to the DynamoDB. It is worth noting that the types and methods in the client library adhere to the object-oriented programming paradigm.

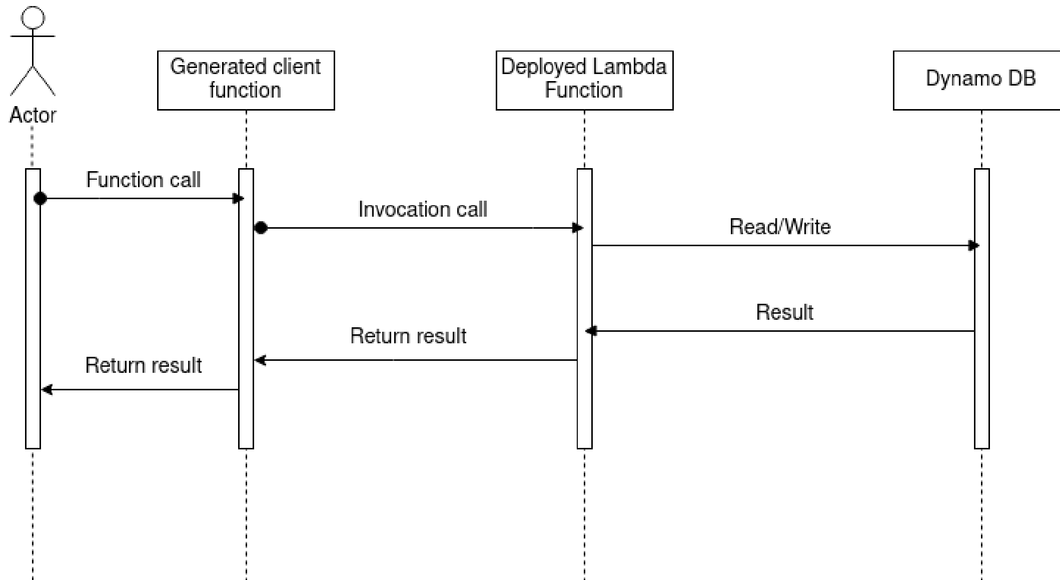


Figure 2.2: Full invocation flow when Nubes is used as an abstraction layer. Communication with the DynamoDB is put in place in the deployed Lambda function.

The presented architectural overview has stayed the same with respect to previous Nubes implementation. To better explain the main difference between Nubes and Nubes+, we must look closely at the sequential diagrams in Figures 2.2 and 2.3. In both sequential diagrams, actors start the execution flow by calling a function from the generated client library obtained with the abstraction layer. It is followed by the invocation call to the Lambda function deployed on the client AWS account. The line of distinction between two diagrams, or better, two approaches, lies in the final stage, communication with the DynamoDB. In the scenario where Nubes is used as an abstraction layer, this step is orchestrated by the Lambda function. All the state management used within the function came from and was developed by Nubes.

On the other hand, looking at the second diagram, the responsibility of communicating with the database shifts to the Beldi library. This choice allows us to implement fault tolerance effectively as the library is now in charge of handing the invocations to DynamoDB

and carries out the role of tracking and performing database requests.

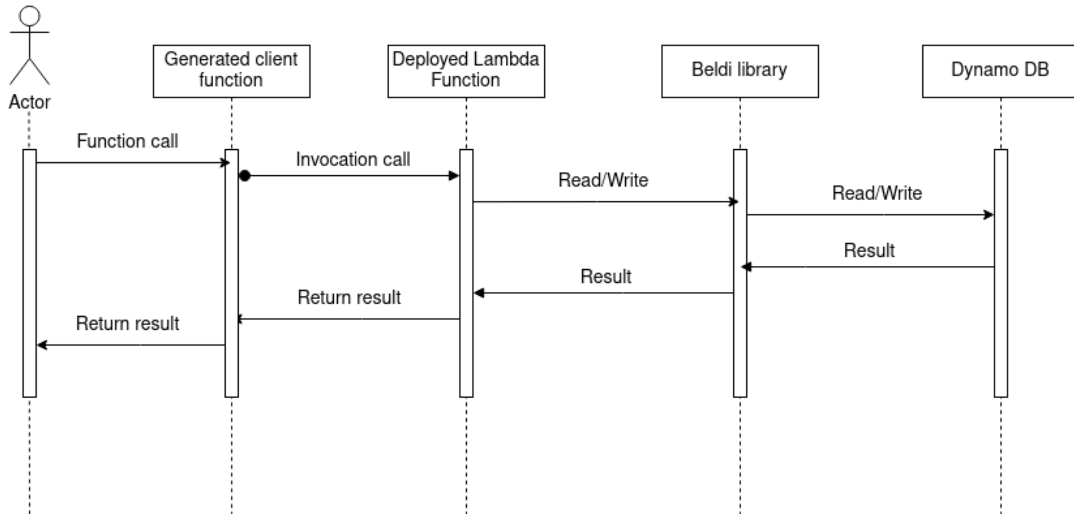


Figure 2.3: Full invocation flow when Nubes+ is used as an abstraction layer. The DynamoDB communication responsibility lies within the Beldi library.

## 2.2. Differences with Nubes

Nubes+ enriches Nubes with exactly-once execution guarantees and to achieve it, it leverages Beldi's library we already described. Introducing Beldi's library came with the required changes we had to follow to stay compatible. Since Nubes is an abstraction layer that provides state management, or in other words, frees developers from state management, we had to migrate and adjust all the database interactions to use Beldi's API instead. Beldi's API provided within the library determines from the SSF's input whether SSF is a part of the transaction, its step number, which lambda is called and other parameters needed to achieve its functionalities. Therefore, apart from database call functions, a few additional changes had to be implemented for us to stay compatible. We needed to change how serverless functions are invoked from the client side and how they are handled on the backend side. While the invocation logic and logic on the backend side needed to be adjusted, it is essential to note that those are hidden from the end client. On the client's library side, changes are kept to a minimum concerning the client's existing interfaces. This chapter provides a detailed overview of the changes we needed to introduce with Nubes+, the limitations we faced, and the approaches we needed to take to overcome them.

### 2.2.1. Database - tables and queries

A few factors come into play when it comes to database initialization and the creation of needed tables. As before, we create a table per each detected *Nobject type*. Furthermore, as detailed in Section 1.5.2, to provide exactly-once execution guarantees, Beldi stores different types of logs depending on the behavior of a stateful serverless function. Those logs are function-specific. Therefore, each stateful serverless function needs its log tables in DynamoDB. In other words, additional tables need to be created for each deployed stateful serverless function that leverages Beldi. To be precise and give an example, for the deployed function *Export*, we need to create an *Export-collector* table that will keep logs from the intent collector and an *Export-log* table that will keep read, write and invoke logs. Moreover, if one wants to use transactions inside of a serverless function or, in other words, to put SSF in a transactional context, an additional shadow table with the sufix *-local* needs to be created. In our example, it would be called an *Export-local* table and would act as a local copy of the state for the transaction. To ensure ACID semantics and fault tolerance, we put a transaction in each stateful serverless function we deploy, as some perform multiple read and write database invocations.

Apart from Beldi's tables and tables for storing client's data based on *Nobject* types, a few more tables potentially need to be created depending on the client's project.

#### Many-To-Many object relationship

Let's examine a scenario like in Listing 2.1. *Shop* and *User* objects are in a mutual many-to-many relationship. In Nubes, to keep track of many-to-many object relationships, an additional join table is created where a table name is a string concatenation of the two types' names. It contains two columns; one stores the IDs of the first *Nobject* type (e.g., *User*) and the other stores the IDs of the other *Nobject* type (*Shop*). The order of columns with IDs is the same as the order of types in the table name. Moreover, the first type name is set to be a primary key, while the second type (of which IDs are stored in the second column) is used as a sort key. A query with a partition key is used to obtain the elements associated with the object defined as the first type in this relation. To do so the other way around, to get all the IDs of the first type associated with the second type's ID, it was necessary to introduce indexes and use them in this scenario. The index's key is the second type in the join table relation, one used as a sort key. This way, Nubes provides a way to obtain all the elements associated with the ID from either side of a many-to-many relation.

```
type Shop struct {
    Id     string
    Name   string
    Owners lib.ReferenceNavigationList[User] `nubes:"hasMany-Shops"`
}

type User struct {
    FirstName string
    LastName  string
    Shops     lib.ReferenceNavigationList[Shop] `nubes:"hasMany-Owners"`
    Orders    lib.ReferenceList[Order]
}
```

**Listing 2.1:** Example of objects in many-to-many relationship.

The same approach was not possible with Nubes+ as we are limited to Beldi's APIs. Beldi's read API does not provide a way to query with indexes or parameters besides a key and a table name.

```
func Read(env *Env, tablename string, key string) interface{}

func Write(env *Env, tablename string,
            key string,
            update map[exp.NameBuilder]exp.OperandBuilder)

func CondWrite(env *Env, tablename string,
                key string,
                update map[exp.NameBuilder]expression.OperandBuilder,
                cond exp.ConditionBuilder) bool
```

**Listing 2.2:** Beldi's APIs for database reads and writes.

Another limitation that can be noticed in Listing 2.2 is that Beldi returns a single element from the table name associated with a given key. Therefore, returning multiple objects with a single query is not possible. To overcome these limitations, we needed to find a way to get all the objects from many-to-many relationships associated with their owner using a single read call and without the help of partition keys or indices. Our way around it is to use not one but two join tables. Their names are in the format: *FirsttypeSecondtype* and *SecondtypeFirsttype*. For the example above, the tables would be called *UserShop* and *ShopUser*. To achieve our goal and tackle Beldi's limitations, these tables serve us as a set collection structure where each key is unique. Keys are IDs of the first object

type in the table name. The value we store per each key is a list of the second type's IDs. Following this approach, we solved the problem of returning multiple elements associated with one key in one table read call, as Beldi presented itself with limitations here.

To ensure both tables are compatible with one another and to keep data integrity, tables are simultaneously updated. For example, adding a new shop ID to the list of shop IDs of a user is followed by appending a user ID to the list of user IDs of that particular shop. Needless to say, these actions are done in a transaction, as updating multiple tables is not possible with Beldi.

## One-To-Many object relationship

Even designing bidirectional one-to-many relationships presented itself as a problem. Nubes uses queries to get multiple objects associated with a specific key from one-to-many relationships and relies on partition keys and indices. Beldi does not provide this flexibility; thus, once again, we encountered Beldi's limitation, and finding another approach to achieve the same functionality was needed. We applied similar logic as we did in many-to-many relationship.

```
type Shop struct {
    Id       string
    Name     string
    Products lib.ReferenceNavigationList[Product]`nubes:"hasOne-SoldBy"`
}

type Product struct {
    Id                string
    Name              string
    QuantityAvailable int
    SoldBy            lib.Reference[Shop] `dynamodbav:",omitempty"`
    Discount          lib.ReferenceList[Discount]
    Price             float64
}
```

**Listing 2.3:** Example of objects in one-to-many relationship.

To declare one-to-many object relationships, one object has to have a reference to another object, while the other one has a collection of the previous one. To define this relationship with Nubes and Nubes+, one object has a *Reference* type, and the other has a *ReferenceNavigationList*; see Listing 2.3. From the point of view of an object that has a reference to another one, nothing has changed. On the other hand, retrieving many

associated with a parent object needed to be adjusted. In Nubes, retrieving many is done by a query that filters the rows based on the value of the *Reference* field. The *Reference* field has to have the same ID as the object type with the *ReferenceNavigationList* field. In Nubes+, we use an additional table. The table name is the concatenation of the types in the format *ObjectmanyObjectone*. For the example shown in Listing 2.3, the table would be called *ShopProduct*. As for many-to-many, the table serves as a set, where the key is the collection owning object type's ID, and the value is the list of IDs. For example, in *ShopProduct*, we store a list of product IDs for a shop ID.

Properly updating this table presented itself as a challenge. Namely, many factors and different use cases exist when the reference type can be changed. Export or setter functions are two main examples. Covering those use cases in export, setter functions (and others), and adding checks if the field has changed presented itself as an error-prone task. For this reason we decided to unify the behavior of the bidirectional relationships. In many-to-many object relationships, the client has at its disposal an *AddToManyToMany* method from the *referenceNavigationList* type to add an item to a collection; see Listing 1.2.

The novelty of Nubes+ in one-to-many relationships is if clients want to add an item into a collection, they must do it explicitly. Essentially, in the same way as for many-to-many relationships. Since all the collections are of type *referenceNavigationList*, the method list is still as in Listing 1.2. However, calling an *AddToManyToMany* function on an object in a one-to-many relationship would be misleading. To avoid misleading function names, we renamed the *AddToManyToMany* into *AddToMany*. When called in the context of a one-to-many relationship, it will append an ID in a list of associated IDs with that particular object.

## Storing data

There is also a difference in how Nubes and Nubes+ store client data. Essentially, the content of the object tables is different, and the reason behind it lies in Beldi's necessity to keep write logs in the same atomicity scope as client data; thus, write logs and client data are kept in the same table. This is not true in Nubes, where tables contain only the client's data. Examples of table contents are given in the Figures 2.4 and 2.5.

Database initialization is performed with the generator during handler definition generation if specified by the client with the parameter flag. It happens as a final stage of code-to-code translation, and all the information needed for table creation is gathered during the parsing process. Tables needed for storing client's data are created based on defined *Nobject* types. For each many-to-many object relationship, two additional tables

**Attributes** — Add new attribute ▼

| Attribute name | Value | Type | |
|---|---|---|---|
| Id - *Partition key* | 561cda11-6b21-45e5-9ac0-fe7168757281 | String | |
| Discount | Insert a field ▼ | List | Remove |
| 0 | 8652ed8e-2c72-4708-8848-a3f32cea321d | String | Remove |
| Name | USB | String | Remove |
| Price | 7.99 | Number | Remove |
| QuantityAvailable | 30 | Number | Remove |
| SoldBy | 2f945e5c-79bf-4fb9-a60b-292e699558bf | String | Remove |

Figure 2.4: Example of data stored on DynamoDB with Nubes as an abstraction layer. An item contains only the attributes of the *Project* object, defined in Listing 2.3.

**Attributes** — Add new attribute ▼

| Attribute name | Value | Type | |
|---|---|---|---|
| K - *Partition key* | 303f8421-9107-4e4f-a626-9e2210251b22 | String | |
| ROWHASH - *Sort key* | HEAD | String | |
| GCSIZE | 0 | Number | Remove |
| LOGS | Insert a field ▼ | Map | Remove |
| LOGSIZE | 2 | Number | Remove |
| V | Insert a field ▼ | Map | Remove |
| Discount | Insert a field ▼ | List | Remove |
| 0 | 5f42ae45-14b9-4b6a-94b3-d6613e2a2e6e | String | Remove |
| Id | 303f8421-9107-4e4f-a626-9e2210251b22 | String | Remove |
| Name | USB | String | Remove |
| Price | 7.99 | Number | Remove |
| QuantityAvailable | 30 | Number | Remove |
| SoldBy | 9eb1c573-8a5c-4689-b50d-0c91b237e583 | String | Remove |

Figure 2.5: Example of data stored on DynamoDB with Nubes+ as an abstraction layer. An item contains attributes of Beldi's *WriteLog* together with the attributes of the *Project* object, defined in Listing 2.3.

are created. For each bidirectional one-to-many object relationship, an additional table is created. Three tables are created for each stateful function deployed on AWS Lambda to store Beldi's logs.

### 2.2.2. SSFs, handlers and client library

In this section, we will go through other changes that need to be introduced. First, we will give an overview of backend changes worth mentioning, and then we will proceed with changes concerning those who develop with Nubes+.

Before discussing why those changes were needed and how we implemented them, we need to provide more details about the core piece of the library - *Env* (Environment). *Env* is a structure defined in Beldi's library that holds all the necessary information and values needed for Beldi to provide its guarantees; see Listing 2.7. *LambdaId* is the name of the lambda function deployed on AWS Lambda that is currently being invoked. *InstanceId* is a unique identifier of a particular lambda invocation call. *LogTable*, *IntentTable* and *LocalTable* contain table names where Beldi's log data is stored. *StepNumber* increasingly counts every external operation where the external operation is, e.g., reading a value from the database. *Input* is the client's actual input, usually data to be stored in the database. *TnxId* is the transaction identifier used in a transactional context, and *Instruction* stands for either *COMMIT*, *ABORT*, or *EXECUTE*. The *Env* structure is initialized before the main body of the SSF is executed. However, to achieve this initialization, the handler function needs to be wrapped with Beldi's *Wrapper* method; see the example in Listing 2.5.

Notice that there is a difference with Nubes where the handler function is passed as an argument to the Start method immediately. The start method is one of the functions in AWS's lambda package, which offers the lambda programming model. See Listing 2.4 and Listing 2.5.

```
1 func HandlerName(input aws.JSONValue) (<optional>, error) {
2     // function body
3 }
4
5 func main() {
6     lambda.Start(HandlerName)
7 }
```

**Listing 2.4:** Passing a handler function to the *Start* method.

```
func HandlerName(env *beldilib.Env) (interface{}, error) {
    // function body
}

func main() {
    lambda.Start(beldilib.Wrapper(HandlerName))
}
```

**Listing 2.5:** Passing a handler function to the *Start* method.

The *Env* type in the context of Beldi has a similar role as the *Context* type in Golang projects [38]. Good practice when writing projects in Go that deal with networks and concurrencies is to pass a variable of type *Context* as a first variable. *Context* provides and holds information about the environment where the function is executed. The same applies to the variable of type *Env*. It gives the library all the context about the environment where lambda invocation is executed. Beldi requires clients of its APIs to provide that context (variable of type *Env*) as a first function parameter to understand the current state of the invocation and how to proceed. That is the crucial reason for the change we are introducing, and it concerns those who develop with Nubes+. Namely, context (a variable of type *Env*) must be propagated to every read or write method. Due to a chain of propagations, exposed functions of the Nubes library come in Nubes+ with an adjusted signature, presented in the Listing 2.6

```
func Load [T Nobject](env *beldilib.Env, id string) (*T, error)
func Export [T Nobject](env *beldilib.Env, objToInsert Nobject) (*T,
    error)
func Delete [T Nobject](env *beldilib.Env, id string) error
```

**Listing 2.6:** Nubes+ library functions.

For the same reason, due to a chain of propagations, the same applies to all custom methods defined by the client. If a client wants to create its own methods and call, e.g. *Export*, in the method body, there will be a need to put a variable of type *Env* as a first function parameter. Even if the custom method does not call methods for explicit database-related operations, it is highly likely to perform operations on the object. Let's remind our reader that in the handler-generating phase performed by the generator, all custom-made methods are extended with additional checks if the object state should be retrieved or saved to the database. Additional checks are based on statuses of *invocationDepth* and *isInitialized* flags. Therefore, even if the client does not explicitly use database

interaction methods, due to the operations on the object, there will be a need to perform calls to the database to retrieve the object state or to save performed changes. Again, database interaction methods in the Beldi library require a variable of type *Env* to be passed as a first argument. Consequently, if the client does not explicitly use database interaction methods in its custom method, the generator will insert a variable of type *Env* as the first method's parameter during the code-to-code handler translation phase.

```
1  type Env struct {
2      LambdaId    string
3      InstanceId  string
4      LogTable    string
5      IntentTable string
6      LocalTable  string
7      StepNumber  int32
8      Input       interface{}
9      TxnId       string
10     Instruction string
11     Baseline    bool
12
13     // Added with Nubes+
14     TypeName string
15 }
```

**Listing 2.7:** *Env* structure holds information needed for Beldi to provide its guarantees. *TypeName* is added by us to know on which object type the function was called.

### 2.2.3. Generator

Changes and adjustments needed to be made on the generator side as well. The generator has two modes: handler and client. Handler mode creates handler functions and enriches methods and defined object types. Client mode creates a client library, or in other words, exposes methods ready to be used by the client. When called, they invoke the appropriate lambda function on AWS and return the result to the client. Files and code generated for the client library changed minimally. In the previous section, we mentioned that we renamed a method from the client library from *AddManyToMany* to *AddToMany*. The reasons behind it were already explained. Apart from that, only the error handling part of the invocation calls to the AWS was improved as we added more checks of the returned result.

Changes in the handler part mainly include adjustments in the template files for handler

functions. Files and code generated in a handler mode are semantically more different from the previous implementation; however, the logical flow has stayed the same. We introduced transactions, wrapped the SSF function with Beldi's Wrapper method, and ensured all the underlying methods use Beldi's API when database interaction is needed.

Enhanced logging was also helpful to have during the development phase and was intentionally left in the code. We believe logging is a crucial aspect of code development and comes with benefits. The idea is to provide valuable information about the object's state and SSF at different points during execution. If something goes wrong, clients can go to their CloudWatch, check logs of deployed lambda and see a sequence of steps performed that led up to an error or unexpected behavior. It will make it easier to diagnose and fix potential issues.

## 2.3. Limitations

Introducing Beldi's library as a way to provide fault-tolerant object-oriented programming for stateful serverless functions brought about certain limitations that need careful consideration. In contrast to Nubes, migration to Beldi usage posed challenges as previously, each database interaction was custom-made and optimized. After the migration, not all optimal approaches could be preserved as database tables have predefined structures and provide minimal query flexibility.

Let's start with the mentioned *Read* function and its definition. The function definition, shown in Listing 2.2, is designed to return a single object based on a given key. The abstraction layer Nubes+ provides the possibility of defining object relationships; thus, returning multiple objects from the database is a valid use case. Beldi lacks this possibility, thus forcing us to adopt a less-than-optimal $\mathcal{O}(n)$ approach. Instead of fetching multiple objects simultaneously, objects must be fetched one by one, introducing potential performance inefficiencies. Apart from retrieving multiple objects at once, the simple nature of the queries prevents us from doing value-based filtering; instead, we must retrieve all values from the database and perform the filtering ourselves on the backend side, introducing potential performance inefficiencies here as well.

Furthermore, to remain compatible with Nubes and continue offering the possibility of defining one-to-many and many-to-many object relationships, we had to come up with a solution that involved data redundancy due to the constraints we faced. In both scenarios, we had to create an additional table containing data already present in the database but conforming to the format compatible with Beldi queries. This approach was not previously necessary under Nube, as custom indexes were strategically implemented and

utilized instead.

We would like to acknowledge that these limitations are not inherent flaws in Beldi but rather challenges that arose from integrating Beldi into an existing architecture. The changes we introduced resulted from finding a solution that balances the unique features of Beldi with the desired functionalities of Nubes+.

## 2.4. Lifecycle

Project lifecycle starts with the type definition. Developers define all the object types they need and encapsulate their functions. Let's emphasize that it is the task of a developer to follow programming principles and standard practices and design object methods in a way that reflects logical coherence with the objects they operate on.

Nubes+ provides a wide range of possibilities in type definition. We have already provided details on defining object methods, including the requirements that must be followed. Additionally, we have discussed various approaches to defining object relationships. For a more in-depth look at the complete set of possibilities in the type definition, we encourage readers to consult the Nubes referenced paper [11]. However, the changes introduced in Nubes+ must also be taken into account.

After defining all type definitions, the generator generates the handler functions and redefines type definitions for Nubes+ needs. At this point, handler functions must be built and deployed on AWS Lambda to enable the end client to use them in their project.

```
./build_handlers.sh generated/
sls deploy
```

**Listing 2.8:** Build and deploy.

To build and deploy, in the main directory of types definitions, project developers can call two commands presented in Listing 2.8. The first command invokes a Bash script that builds each handler function from the folder specified as an argument, placing the newly obtained binary files in the created */bin* folder. The second command searches for the *serverless.yml* file in the directory where it was executed. The *serverless.yml* file is created by the generator in the handler generation phase. Let us note that the requirements to run the final command are properly configured AWS credentials and installed serverless framework [49].

After the object types have been defined, the function handlers have been generated and deployed on AWS; it remains to generate the client library where function calls perform lambda invocations and return the result to the client.

# 3 | Evaluation

The focus of this thesis was to add exactly-once execution guarantees into Nubes. To achieve fault tolerance, we needed to bring another layer of complexity that would perform all the necessary logic behind it. The main focus of this chapter is to analyze the performances of Nubes+ and understand the performance trade-off exactly-once execution brings.

Since Nubes+ is enhanced Nubes, we will present comparisons between Nubes+, Nubes, and baseline implementation. Baseline implementation in our evaluation later in the text denoted as SSF, represents implementation where all the state management needed to be designed and implemented by the developer, as one would do without Nubes. We will provide more details later in the text.

## 3.1. Testing setup

We performed the same case study as it was done in Nubes [11]. We built a simple application that covers many functionalities and features Nubes+ provides. To be precise, we built a simple hotel booking application where *User* can make a *Reservation* for a *Room* in a *Hotel*. The *Hotel* is located in a *City*. This kind of hotel booking application is a common practice when putting under evaluation architectural design or microservices, and it derives from DeathStarBench [4]. Other applications derived from DeathStarBench include social network, media service, e.g., movie review applications, banking system, and others. Topics are different, but the core use cases are similar.

Table 3.1 presents the list of all functionalities we tested together with the high-level access mode to the database. Let us note here that even methods we described as read-only, like *get-user-reservations* that fetches a list of reservations made by a user, are not purely read-only anymore. We mentioned how Beldi logs each external call to the database; thus, each read call is written to the log table. However, we put a high-level overview of the access mode as the table value to make it intuitive to the reader what the idea behind the method is and what it does.

| Task | Access mode | Features under test |
|---|---|---|
| **user-registration** | read-write | export to DB |
| **delete-user** | read-write | delete from DB |
| **set-hotel-rate** | read-write | update a field in the DB |
| **reserve** | read-write | export to DB,<br>update a field in the DB,<br>update many-to-many relation |
| **recommend-location** | read-only | get from one-to-many relation |
| **recommend-price** | read-only | get from one-to-many relation |
| **login** | read-only | invoke a method on an object |
| **get-hotels** | read-only | get copies of multiple objects<br>from one-to-many relation |
| **get-user-reservations** | read-only | get copies of multiple objects<br>from one-to-many relation |

Table 3.1: Overview of all tasks performed and their purposes for the evaluation.

Let's comment on what is shown in the table 3.1. *user-registration* creates a user and exports it to the database. *delete-user* removes the user from the database. *login* checks if the given user's secrets match those in the database, e.g., if the password entered is the same as the one in the database. *get-user-reservations* returns a list of reservations performed by the user. *set-hotel-rate* updates the total hotel rating. *get-hotels* returns a list of hotels in a city. *reserve* is used to make a room reservation, and finally, *recommend-location* and *recommend-price* mimic a behavior when a user is recommended a few hotels based on the location and hotel rate.
Having the same testing setup as in Nubes gives us a few benefits:

- Building a hotel booking application is a common practice when working with microservice applications; thus, the results are relevant and meaningful to the broader community (Academia or others who develop with microservices) as we put Nubes+ in the same testing context.
- Following this approach allows us to understand the results better and understand exactly what has changed in performance with the introduction of fault tolerance.
- Use cases cover a good portion of what is possible with Nubes+. We would like to note here that the very fact that this type of application can be built using Nubes+ is worthy of praise as it demonstrates it has an appliance in day-to-day scenarios.

This application comes with different cardinality of object relationships. Namely, two one-to-many relationships exist: between hotel and rooms and city and hotels. Users are in many-to-many object relationships with reservations.

## 3.2. Testing models

We implemented three versions of the presented hotel booking application as testing models. We put them under the test to better understand their performance comparisons.

Further in the text, we will address them as:

- **SSF** for the basic implementation
- **Nubes** for the implementation that used the Nubes methodology
- **Nubes+** for the implementation that used the Nubes+ methodology

The main differences between Nubes+ and Nubes are explained in this work. They are mostly implementation-related and involve handling the many-to-many and one-to-many object relationships. Apart from that, anywhere where the return of multiple objects was needed, Nubes+ had to do it using the $\mathcal{O}(n)$ approach.

SSF is the primary implementation where the developer does all the state management in stateful serverless functions. Instead, in Nubes and Nubes+, all database interactions are hidden from the developer and the client.

## 3.3. Environment setup

All the experimental setup was done by using different Amazon Web Services. To be sure our computer can handle the testing load, we provisioned an EC2 instance of type t3a.2xlarge on AWS with Ubuntu as Amazon Machine Image (AMI). t3a.2xlarge instance type comes with 32 GigaBytes of RAM and 8 vCPU.

All our provisioned services (EC2 instance, DynamoDB, and deployed lambda functions) were placed in the same region to avoid cross-region latencies. We used the US East, North Virginia region for our tests, commonly known as us-east-1.

For invocation calls, we used wrk2, an open-source HTTP benchmarking tool [12]. After the invocation call have been performed, we based our results on retrieved CloudWatch data.

## 3.4. Performance review

Before we started the performance comparison, we had initialized DynamoDB tables and populated them with *dummy* data. Data was auto-generated as the content per se was unimportant for the evaluation.

| Element | Count | Total in DB |
|---|---|---|
| Cities | 5 | 5 |
| Number of hotels per city | 100 | 500 |
| Number of rooms per hotel | 25 | 12500 |
| Number of reservations per room | 20 | 250000 |
| Number of users | 50000 | 50000 |
| 1. Rate of invocations | 100 inv/s | - |
| 2. Rate of invocations | 1000 inv/s | - |

Table 3.2: Default parameters used to evaluate performance.

Table 3.2 shows the state of the database after initial initialization. The total number of elements in the *Reservation* table upon initial seeding was 250000. After the initial population, the total size of *User* table was 50000.

We were focused on measuring the time duration of each task. As mentioned, we used the wrk2 tool to perform invocations towards our *Gateway* Lambda method that further randomly dispatched invocations to other methods. Listing 3.1 shows the full configuration of the parameters we used, where parameters are defined as:

- -$c$ number of open connections
- -$t$ number of threads
- -$R$ rate of invocations per second
- -$d$ duration

To get more precise results, we took into consideration the cold start. The cold start usually occurs when a resource is in an idle state. When an event occurs, the resource starts from scratch and goes through initialization. Cold starts can take time and negatively influence our results; thus, we discarded results obtained from the first 30 seconds of invo-

cations. Taking that into account, the invocation duration we measured took 120 seconds.

```
#!/bin/bash
wrk2 -c 100 -t 8 --latency -R 100 -d 150s -s hotel-full.lua
    <URL_To_Gateway> 15 /path/to/result/folder
```

**Listing 3.1:** Bash script used for performance testing.

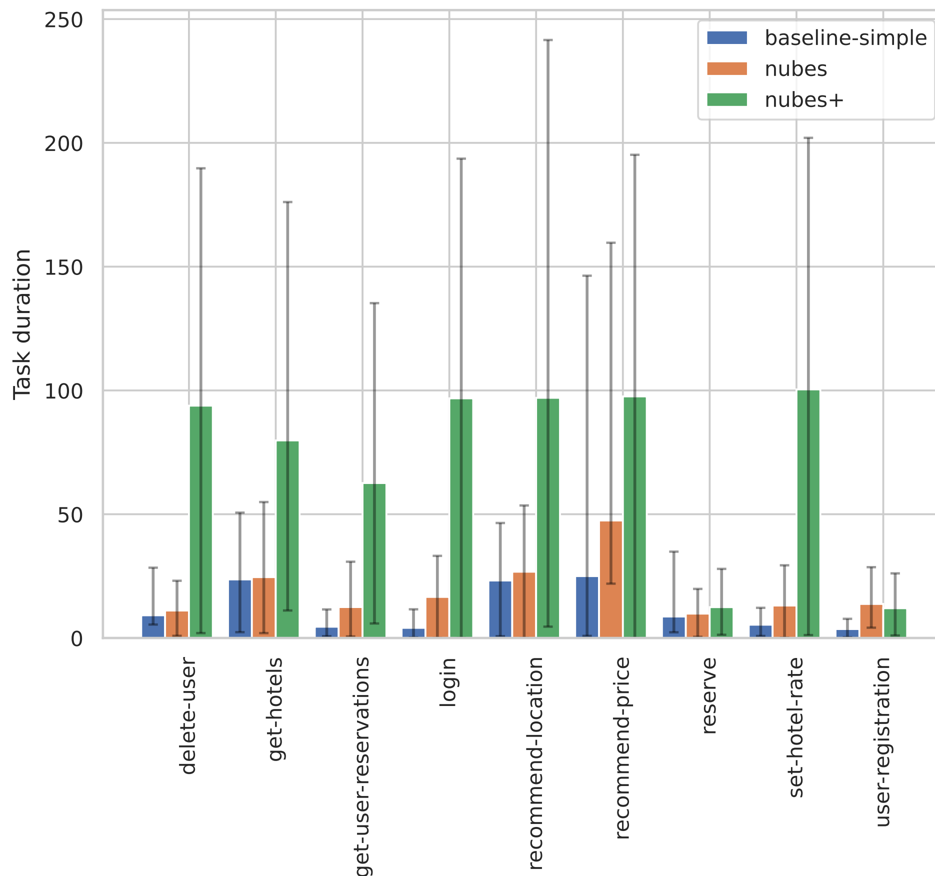


Figure 3.1: Task duration percentiles (10th, 50th and 90th) at 100 inv/s.

Figure 3.1 visually represents the 10th, 50th, and 90th percentile for duration measurements across all testing models, divided by task for a rate of 100 invocations per second. At first glance, it is evident that Nubes+ demonstrates a significantly higher duration for each task when compared to both Nubes and SSF. Results for Nubes and SSF show that our findings align with those concluded in [11]. Nubes introduces a slight overhead compared to the baseline SSF implementation; however, their performances are mutually comparable in most use cases. An interesting finding becomes apparent when one looks at the duration distribution of Nubes+ itself. Methods marked as read-only in the table

3.1 perform significantly slower than ones marked as read-write. This outcome aligns with our expectations. To ensure fault tolerance, Beldi introduces an additional level of complexity. For each external database call performed in the serverless function, Beldi performs additional database calls to log those operations. Furthermore, perhaps the most crucial reason for such an increase in duration of calls lies in Beldi's limitation of returning only one object in the read call. In a high-level overview, imagine a hotel with 200 rooms. To retrieve a list of Room objects associated with a hotel, the process involves initially querying the *HotelRoom* one-to-many join table, which provides a list of room IDs linked to the hotel. Subsequently, a separate query to the Room table is executed for each room ID obtained to fetch the corresponding Room object. Therefore, each time a method in Nubes+ returns a collection of objects to the end-user or during function processing, $n$ invocations to the database are required, where $n$ is the number of objects in the collection.

We performed another test, changing the number of invocations per second from 100 to 1000. The results are shown in the Figure 3.2. Regarding the duration of the Nubes+ task, nothing changed. We can conclude from the results that increasing the invocation rate does not influence task duration. System scale up to infinity if needed; therefore, Nubes+ preserves the serverless solutions' scalability.

## 3.5. Discussion

Performed evaluation and performance testing allows us to answer how does Nubes+ perform when compared to Nubes and baseline SSF model. While Nubes and SSF demonstrated comparable results, the slower performance of our model, Nubes+, is directly related to integrating a specialized fault-tolerant library into the system architecture.

The integration of the fault-tolerant library came with a set of challenges, involving architectural incompatibilities and integration mismatches with the existing model Nubes. While Beldi's integration contributed to resilience of our system, it is accounted for the observed slower performance, as data fetching one-by-one requires additional processing time. However, this trade-off was expected for enhancing fault tolerance, and ensuring the reliability of our model.

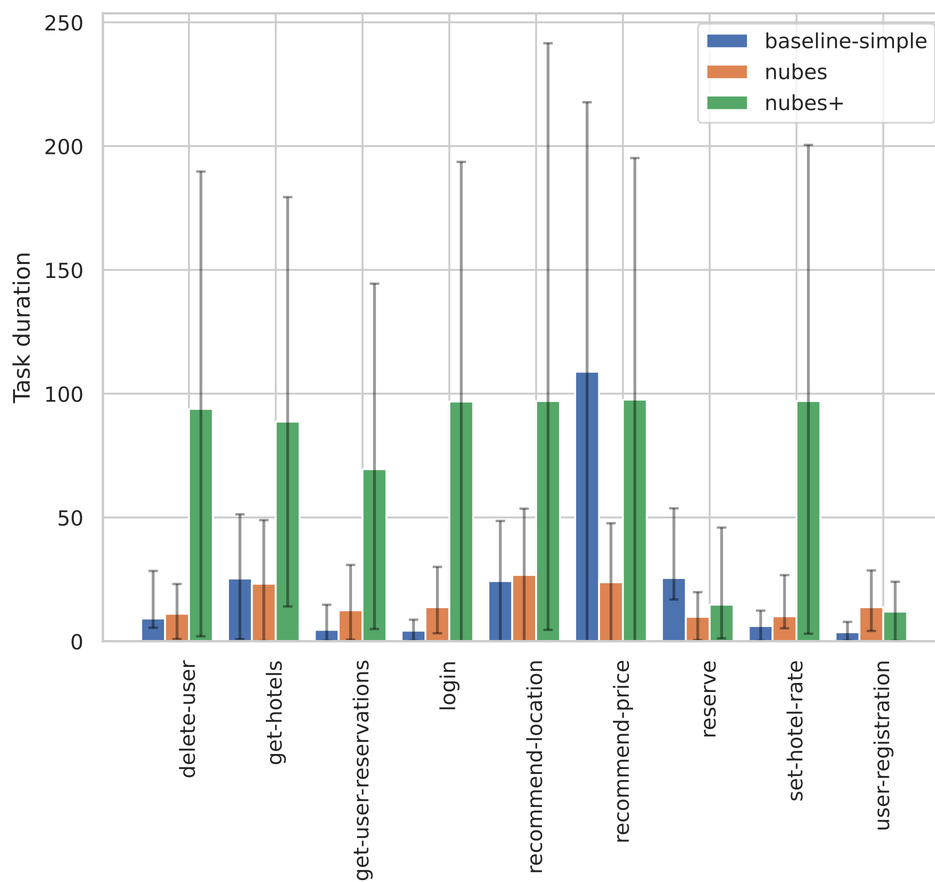


Figure 3.2: Task duration at 10th, 50th, and 90th percentiles, with a request rate of 1000 inv/s. The results mirror those depicted in Figure 3.1, with the distinction of baseline SSF exhibiting increased duration in the *recommend-price* task.

# 4 | Related Work

## 4.1. Serverless Workflows

Function workflows coordinate serverless functions together. As serverless functions are event-driven, chaining them into a data processing pipeline is convenient. Cloud providers come with a framework for defining and managing function workflows. AWS has AWS Step Functions [29] for orchestrating workflows. Microsoft Azure's framework for orchestrating Azure Functions is called Azure Durable Functions [31]. Google Cloud Platform offers Google Cloud Composer [34], a fully managed workflow orchestration.

To go a step further, the line of research is focused on optimizing the transfer of intermediate states and reducing latencies between functions chained into a workflow with the so-called control-flow paradigm. *Faastlane* and *FaaSFlow* are two such approaches. *Faastlane* [6] reduces latency and accelerates function workflows by executing functions as threads within a single process of a container instance. *FaaSFlow* [8] proposes a pattern for scheduling worker-side workflows in the serverless workflow execution context. A more recent study, *DataFlower* [9], proposes a data-flow paradigm for serverless workflows. In *DataFlower*, a container encapsulates a function and data logic units. The function logic unit is responsible for function execution, while the data logic unit manages asynchronous data transmission. Furthermore, there is a collaborative communication mechanism in place between the host and container to enhance the efficiency of data transfer.

*CRUCIAL* [2], is a system created to program highly parallel stateful serverless applications. The idea is to provide a solution for applications that require fine-grained support for mutable state and synchronization, such as machine learning.

## 4.2. Fault tolerance for stateful serverless systems

### Halfmoon

Halfmoon [50] is the newest serverless system for fault-tolerant stateful serverless computing. It uses logging approach, as Boki and Beldi do. As stated in the paper, Halfmoon defines Boki's and Belds's logging protocol as *symmetric* as they log *both* reads and writes to the external state. On the other hand, Halfmoon evolved around the idea that it suffices to log reads *or* writes; thus *asymmetrically*. It provides two logging protocols that come with exactly-once guarantees. Each of them provides either log-free reads or writes. Halfmoon prototype is implemented on top of Boki.

### Boki

Another way to achieve fault tolerance in stateful serverless applications is using Boki. Boki is a FaaS runtime that exports a shared log API to stateful serverless applications to manage their state while ensuring durability, consistency, and fault tolerance [5]. These benefits are achieved through LogBook, an abstraction layer used to access shared logs. Boki is the first system that uses distributed shared logs to manage state for stateful serverless operations.

One of the benefits LogBook provides is read consistency. In other words, LogBook ensures read-your-writes and monotonous readings when reading records. Another characteristic of LogBook is sequence numbers, or *seqnum*. For every newly added log in the record, API generates a unique *seqnum*, which helps to understand the relative order of logs in LogBook. When appending a log record in a LogBook, logs are tagged and put together with their set of tags. Tags help perform selective reads and reduce overheads as records with the same tag form a sort of abstract stream within the LogBook. Another characteristic of a LogBook is auxiliary data, designed to be per-log-record cache storage and used for optimization purposes. LogBooks are multiplexed into physical logs, each with associated metalogs to keep track of its internal state transitions. In Boki, metalogs play an essential role since they provide mechanisms for log ordering, ensuring read consistency, and tolerating machine failures.

Boki comes with three support libraries that provide solutions for stateful FaaS paradigms that benefit from using shared logs with LogBook abstraction: BokiFlow, BokiStore and BokiQueue.

BokiFlow is a library that puts Boki in the context of fault tolerance. Beldi greatly inspired it, guarantees exactly-one execution, comes with transactional semantics, and derives from

Beldi's codebase. However, BokiFlow differentiates itself from Beldi in three ways. Unlike Beldi, which can test-and-append atomically if the current step is logged and append it if it is not logged, this approach is not possible in BokiFlow as the LogBookAPI does not offer the possibility of conditional log appends. In BokiFlow, user data and LogBook are not in the same atomic scope. Moreover, BokiFlow database updates are idempotent. There is no requirement, like in Beldi, that database update and logging of that step have to be an atomic operation. The last thing that differentiates these two libraries is how they implemented locks. Beldi uses conditional updates provided by the database that meet the requirements of the atomic *test-and-set* operation. In BokiFlows, locks are implemented as registers backed by replicated state machines using the LogBook API.

BokiStore is designed as a solution for durable object storage for stateful functions. Studies [1, 54] showed that shared logs can support consistent, durable, scalable high-level data structures. The library provides transaction semantics and stores objects in JSON format.

BokiQueue is designed as a queue library where shared logs are used to build serverless message queues. Serverless functions can indirectly communicate with each other by using BokiQueue's push-and-pop API to send and receive messages.

# 5 | Conclusions

## 5.1. Review and Critique of Beldi library

One thing is sure: Beldi library provides fault tolerance to stateful serverless functions by extending the log-based approach from Olive. The previous chapter already described a more detailed overview of how it was accomplished. Putting Beldi in the context of Nubes allowed us to provide an enriched abstraction layer, Nubes+, built for stateful serverless function, with an execution guarantee. In this section, we would like to provide an overview of the library usage, limitations we encountered, and how we bypassed them to stay consistent with the existing client-facing interface Nubes was already providing.

When working with third-party libraries, the developer first turns to code documentation. Speaking strictly about the Beldi codebase, one must say Beldi does not come with user-friendly documentation since one does not exist. Although Beldi's paper introduces general client APIs, some uncertainties exist when one starts using them.

When developing a public library, developers should be guided by good practice where only functions offered to the end client user are publicly exposed. To put this into the Go programming language context, since both Beldi's library and Nubes+ were developed in it, clients should be able to call only exported functions in the package, where exported functions are those that start with capital letters. This was not the case with Beldi, where all the functions were publicly exposed. Not all of them are meant to be used by the client since their function is specific to the development logic, and generally, that should be hidden from the client. On the contrary, if the functions were purposely meant to be publicly exposed and the client was given the possibility and freedom to use them, the client should be given documentation to fully comprehend their functionalities. Then, they can use them as they fit.

Two main functions Beldi exposes are *Read* and *Write*. Both have their internal *LibRead* and *LibWrite* representatives for end database interactions. We noticed the *Delete* function was not present; only its under-the-hood version, *LibDelete* was. *Read* and *Write* functions take input parameters and do additional logic before calling their representatives

*LibRead* and *LibWrite*. We compensated for the lack of a *Delete* function by following the same logic as in *Read* and *Write*.

We noticed some hidden logic and inconsistency between the *Read* and *Write* functions in the library. We already presented *Read* and *Write* functions in the Listing 2.2. Let's imagine a scenario where the client writes a value to the table called *test* and then reads it like in the Listing 5.1:

```
value := map[expression.NameBuilder]expression.OperandBuilder{
    expression.Name("Value"): expression.Value(i),
}
beldilib.Write(env, "table", "K", value)

databaseValue := beldilib.Read(env, "table", "K")
```

**Listing 5.1:** Example of incorrect library behavior.

The current setup will return *nil* to the *databaseValue*. After digging deeper into the library codebase, we noticed the *Read* function has a projection expression hardcoded to *V*. In the context of DynamoDB, a projection expression is a string that identifies attributes the client wants returned. We compensated for this bug in the *Write* function. Namely, when inserting objects into the database, we saved the entire object under the attribute *V*. The *Read* function correctly returned the object afterward, which we also tested.

Something can always go wrong; thus, error handling and logging are good coding practices. Beldi does check for errors, but in a way we disagree with. To be more specific, if an error occurs during library function calls, the library halts program execution, or to put it in the context of Go language, the library panics. This is a bad practice since we do not want to halt the client's application. Instead, errors should be propagated to the caller and handled properly. That being said, our proposal is to refactor all functions that should be exposed to the client to return an error as an (additional) return parameter.

Good coding practice is to log essential and critical steps to the standard output during execution. Logging makes it easier to pinpoint the root cause of unwanted behaviors later in the debugging and troubleshooting phase. Beldi does not log its steps, and this is something we will propose later on in future work.

As mentioned earlier, to configure Beldi's behavior correctly, the developer needs to create three main tables for each lambda function, and those are *LambdaId*, *LambdaId-collector*, *LambdaId-log*, where *LambdaId* is the name of the deployed lambda function. Beldi comes

with the utility function *CreateLambdaTables* that, given *LambdaId*, creates required tables. However, if developers want to use transactional workflows, they must create an additional table in the *LambdaId-local* format; otherwise, an error will occur. This should have been mentioned in the library documentation if one existed. The library does come with the util method called *CreateTxnTables*. However, the table in the format *LambdaId-local* is not created there. Thus, the function name is misleading.

There was another hard-coded behavior we noticed in the Beldi library we needed to adjust. Beldi offers a *SyncInvoke* function that calls the callee function and waits for an answer; see Listing 5.2. However, also shown in the Listing 5.2, in the invocation call towards AWS, the function name is set to be in the format: *beldi-dev-%s*. If the developer deployed a function called: *ReserveHotel* and passed to *SyncInvoke* string *ReserveHotel* as a collee parameter, the library would, in reality, go to the client AWS and try and search for a function with the name *beldi-dev-ReserveHotel* to be executed, but one does not exist.

```
func SyncInvoke(env *Env, callee string, input interface{}) (interface{}
    , string) {

    // function body

    res, err := LambdaClient.Invoke(&lambdaSdk.InvokeInput{
        FunctionName: aws.String(fmt.Sprintf("beldi-dev-%s", callee)),
        Payload:      payload,
    })

    // ...
}
```

**Listing 5.2:** Example of incorrect library behavior in *SyncInvoke* function.

Let us emphasize that this overview was directed to Beldi's codebase. Those are minor issues that the developers probably overlooked. It would be beneficial if this feedback could be applied as it will save time for future library users.

On the other hand, Beldi's paper nicely explains all of Beldi's internal details and logic and was introduced in the Proceeding of the 14th USENIX Symposium on Operating Systems Design and Implementation.

## 5.2. Future Work

The presented abstraction layer Nubes+ opens the door to different possibilities; thus, various improvements can be implemented. To start from its components, one of the possible improvements regarding the Beldi library is to put in place adjustments regarding the issues we discussed in the section 5.1. Mainly, it includes adding documentation to the codebase, enhancing logging, reformatting the code to follow the recommended guidelines, and correcting a few irregularities we mentioned. Furthermore, Beldi currently uses the first version of the AWS SDK for Go package, which is slowly becoming deprecated. Thus, the additional improvement could be refactoring the codebase to use the AWS SDK version 2 package. More about both package versions can be found in the documentation [27].

If possible, broadening the spectrum of possibilities and covering a broader range of use cases, making Beldi's queries more flexible, or expanding the list of API methods for database interaction would be of great benefit. Beldi would become more adaptable as a solution and would have a positive effect on overall Nubes+ performances. For example, addressing the issue of not having the possibility to use database indices would result in minimizing the number of database calls. Fewer calls to the database would make Nubes+ faster, more robust, reliable, and resilient.

Nubes+ itself has room for improvement. In the related work, we mentioned Boki 4.2 and Halfmoon 4.2, the other technologies that tackle the issue of ensuring exactly-once execution in stateful serverless functions. It would be interesting to compare performances with the solution that relies on Boki or Halfmoon. A step further would be designing a solution that ensures fault tolerance customized for Nubes. That would come as a benefit of not having any trade-offs a library may come with.

The long-term goal would be to make Nubes+ more flexible, versatile, and adaptive as a solution. The next steps towards achieving the mentioned goals would be improving the client-facing interface, or in other words, introducing more features to the client library and offering clients more possibilities. We bring several ideas on how the improvement on that part can be achieved. One is to expand the list of methods currently available to the client. It is a first step towards making Nubes+ more flexible and more adaptive, as it means covering more use cases on the client side. Another improvement could be optimizing already existing methods and reducing latency. Both greatly contribute to improving the overall user experience. An alternative approach to covering a wider range of client use cases is introducing the possibility of generating the client library in other programming languages. The proposed feature would enable the usage of Nubes+ in, e.g., web application models for backend service development, with the frontend serving as the

client for the generated library. The Nubes paper [11] discusses mentioned approach and the advantages of generating the client library in JavaScript, as JavaScript is primarily used for frontend web development. Apart from making Nubes+ available to a broader audience by extending the generating model with additional programming languages, the improvement of Nubes+ can go even further and make Nubes+ compatible with other database models or cloud providers.

## 5.3. Overall Conclusion

One of the most significant benefits of the cloud is that it is flexible with its wide range of services. Individuals and businesses are turning to the cloud as it allows them to be more flexible, cost-effective, and innovative in their use of technology. A frequently used cloud model is Function-as-a-Service. Clients upload their developed functions to the cloud, which are triggered as a response to upcoming events or requests. Because of their great benefits, the need to manage the state with serverless functions appeared very quickly. Managing a state with stateful serverless functions is a challenging task that requires careful design.

This thesis introduces Nubes+, an abstraction layer that provides fault tolerance for object-oriented programming for stateful serverless functions. Nubes+ is enhancing Nubes, an existing state management model for stateful serverless functions, with exactly-once execution guarantees.

To provide exactly-once execution guarantees, Nubes+ uses Beldi for database interactions. Although Beldi excels in providing fault tolerance for stateful serverless functions, its trade-offs become apparent compared to the custom-designed and optimized nature of Nubes. As reflected in the results, the need for effective workarounds emerged in various scenarios, demonstrating the complexities of integrating fault tolerance into our system.

Although introducing changes in Nubes to make it compatible with the Beldi library was challenging, the changes needed on the client side were reduced to a minimum, therefore, the client interface remained practically the same.

# Bibliography

[1] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522732. URL `https://doi.org/10.1145/2517349.2522732`.

[2] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López. Stateful serverless computing with crucial. *ACM Trans. Softw. Eng. Methodol.*, 31 (3), mar 2022. ISSN 1049-331X. doi: 10.1145/3490386. URL `https://doi.org/10.1145/3490386`.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles*, 2007. URL `https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store`.

[4] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304013. URL `https://doi.org/10.1145/3297858.3304013`.

[5] Z. Jia and E. Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, 2021.

[6] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu. Faastlane: Accelerating Function-

as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL `https://www.usenix.org/conference/atc21/presentation/kotni`.

[7] R. Lafore. *Object-Oriented Programming in C++, Fourth Edition.* SAMS, 2005.

[8] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 782–796, New York, NY, USA, 2022. doi: 10.1145/3503222.3507717.

[9] Z. Li, C. Xu, Q. Chen, J. Zhao, C. Chen, and M. Guo. Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration, 2023.

[10] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, nov 1994. ISSN 0164-0925. doi: 10.1145/197320.197383. URL `https://doi.org/10.1145/197320.197383`.

[11] K. Marek. Nubes: Object-oriented programming for stateful serverless functions, 2023.

[12] Online: a HTTP benchmarking tool based mostly on wrk. wrk2. `https://github.com/giltene/wrk2` [Accessed Nov, 2023].

[13] Online: Amazon CloudWatch. Amazon cloudwatch. `https://aws.amazon.com/cloudwatch` [Accessed Nov, 2023].

[14] Online: Amazon CloudWatch Documentation. What is amazon cloudwatch? `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html` [Accessed Nov, 2023].

[15] Online: Amazon CloudWatch Working with log groups and log streams. Working with log groups and log streams. `https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/Working-with-log-groups-and-streams.html` [Accessed Nov, 2023].

[16] Online: Amazon DynamoDB. Amazon dynamodb. `https://aws.amazon.com/dynamodb` [Accessed Nov, 2023].

[17] Online: Amazon DynamoDB Partitioning. Partitioning. `https://docs.aws.amazon.com/whitepapers/latest/comparing-dynamodb-and-hbase-for-nosql/partitioning.html` [Accessed Nov, 2023].

[18] Online: Amazon DynamoDB Point-in-time recovery: How it works. Point-in-time recovery: How it works. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/PointInTimeRecovery_Howitworks.html [Accessed Nov, 2023].

[19] Online: Amazon Simple Storage Service Documentation. Amazon simple storage service. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html [Accessed Nov, 2023].

[20] Online: Amazon Web Services. Amazon web services. https://aws.amazon.com [Accessed Nov, 2023].

[21] Online: AWS CloudFormation. Aws cloudformation. https://aws.amazon.com/cloudformation [Accessed Nov, 2023].

[22] Online: AWS CloudFormation Documentation. What is aws cloudformation? https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html [Accessed Nov, 2023].

[23] Online: AWS CloudFormation Working with stacks. Working with stacks. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stacks.html [Accessed Nov, 2023].

[24] Online: AWS Identity and Access Management Documentation. Aws identity and access management. https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html [Accessed Nov, 2023].

[25] Online: AWS Lambda Documentation. What is aws lambda? https://docs.aws.amazon.com/lambda/latest/dg/welcome.html [Accessed Nov, 2023].

[26] Online: AWS Lambda Features. Aws lambda features. https://aws.amazon.com/lambda/features [Accessed Nov, 2023].

[27] Online: AWS SDK for Go Documentation. Aws sdk for go documentation. https://docs.aws.amazon.com/sdk-for-go [Accessed Nov, 2023].

[28] Online: AWS Serverless. Serverless on aws. https://aws.amazon.com/serverless [Accessed Nov, 2023].

[29] Online: AWS Step Functions. Aws step functions. https://aws.amazon.com/step-functions [Accessed Nov, 2023].

[30] Online: AWS What is cloud computing? What is cloud computing? https://aws.amazon.com/what-is-cloud-computing [Accessed Nov, 2023].

[31] Online: Azure Entity Functions Documentation. Entity functions. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=function-based%2Cin-process%2Cpython-v2&pivots=csharp [Accessed Nov, 2023].

[32] Online: Azure Functions Documentation. Azure functions documentation. https://learn.microsoft.com/en-us/azure/azure-functions [Accessed Nov, 2023].

[33] Online: Azure What is cloud computing? What is cloud computing? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing [Accessed Nov, 2023].

[34] Online: Cloud Composer. Cloud composer. https://cloud.google.com/composer [Accessed Nov, 2023].

[35] Online: GCP What is cloud computing? What is cloud computing? https://cloud.google.com/learn/what-is-cloud-computing [Accessed Nov, 2023].

[36] Online: Getting started with IBM Cloud Functions. Getting started with ibm cloud functions. https://www.ibm.com/products/functions [Accessed Nov, 2023].

[37] Online: Global tables - multi-Region replication for DynamoDB. Global tables - multi-region replication for dynamodb. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GlobalTables.html [Accessed Nov, 2023].

[38] Online: Go Context Package Documentation. https://pkg.go.dev/context [Accessed Nov, 2023].

[39] Online: Go Documentation. Effective go. https://go.dev/doc/effective_go [Accessed Nov, 2023].

[40] Online: golang. Golang, language's official website. https://go.dev/ [Accessed Nov, 2023].

[41] Online: Google Cloud. Goodle cloud. https://cloud.google.com/?hl=en [Accessed Nov, 2023].

[42] Online: Google Cloud Functions. Cloud functions. https://cloud.google.com/functions/?hl=en [Accessed Nov, 2023].

[43] Online: IBM Cloud. Ibm cloud. https://www.ibm.com/cloud [Accessed Nov, 2023].

[44] Online: IBM What is cloud computing? What is cloud computing? https://www.ibm.com/topics/cloud-computing [Accessed Nov, 2023].

[45] Online: IBM What is FaaS (Function-as-a-Service)? What is faas (function-as-a-service)? https://www.ibm.com/topics/faas [Accessed Nov, 2023].

[46] Online: IBM What is serverless? What is serverless? https://www.ibm.com/topics/serverless [Accessed Nov, 2023].

[47] Online: Microsoft Azure. Microsoft azure. https://azure.microsoft.com/en-us [Accessed Nov, 2023].

[48] Online: Resilience and disaster recovery in Amazon DynamoDB. Resilience and disaster recovery in amazon dynamodb. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/disaster-recovery-resiliency.html [Accessed Nov, 2023].

[49] Online: Serverless Framework Documentation; Setting Up Serverless Framework With AWS. Setting up serverless framework with aws. https://www.serverless.com/framework/docs/getting-started [Accessed Nov, 2023].

[50] S. Qi, X. Liu, and X. Jin. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 314–330, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613154. URL https://doi.org/10.1145/3600006.3613154.

[51] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the Fault-Tolerance promise of cloud storage using locks with intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 501–516, Savannah, GA, Nov. 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty.

[52] B. Stroustrup. What is "object-oriented programming"? (1991 revised version). In *1st European Software Festival*, 1991.

[53] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., USA, 2006. ISBN 0132392275.

[54] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi. vCorfu: A Cloud-Scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 35–49, Boston, MA, Mar.

2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael.

[55] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, Nov. 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

# List of Figures

2.1 Architectural overview of an execution flow. . . . . . . . . . . . . . . . . . . . 17
2.2 Invocation flow with Nubes as an abstraction layer. . . . . . . . . . . . . . . 18
2.3 Invocation flow with Nubes+ as an abstraction layer. . . . . . . . . . . . . . 19
2.4 Example of data stored on DynamoDB with Nubes as an abstraction layer. 24
2.5 Example of data stored on DynamoDB with Nubes+ as an abstraction layer. 24

3.1 Task duration percentiles (10th, 50th and 90th) at 100 inv/s. . . . . . . . . . 35
3.2 Task Duration Percentiles at 1000 inv/s. . . . . . . . . . . . . . . . . . . . . 37

# List of Tables

3.1 Task, access mode and feature under test . . . . . . . . . . . . . . . . . . . . 32
3.2 Default parameters used to evaluate performance. . . . . . . . . . . . . . . . 34

# Listings

1.1 Example of naming convention and encapsulation in Go . . . . . . . . . . . 8
1.2 Methods defined in *ReferenceNavigationList* object type. . . . . . . . . . . 9
1.3 Allowed method signatures. . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.1 Example of objects in many-to-many relationship. . . . . . . . . . . . . . . . 21
2.2 Beldi's APIs for database reads and writes. . . . . . . . . . . . . . . . . . . . 21
2.3 Example of objects in one-to-many relationship. . . . . . . . . . . . . . . . . 22
2.4 Passing a handler function to the *Start* method. . . . . . . . . . . . . . . . 25
2.5 Passing a handler function to the *Start* method. . . . . . . . . . . . . . . . 26
2.6 Nubes+ library functions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.7 *Env* structure holds information needed for Beldi to provide its guarantees. *TypeName* is added by us to know on which object type the function was called. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.8 Build and deploy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.1 Bash script used for performance testing. . . . . . . . . . . . . . . . . . . . 35
5.1 Example of incorrect library behavior. . . . . . . . . . . . . . . . . . . . . . 44
5.2 Example of incorrect library behavior in *SyncInvoke* function. . . . . . . . 45

# Acknowledgements

This thesis brings to an end my studies and my Politecnico di Milano story, which greatly contributed to my personal growth. Many things marked my studies in Milan. Milan itself, travels, and conversations. I will forever be grateful for all the memories I made here, the people I met, those who passed through, and those who remained in my life during this time. I am happy to say there were many of them.

I would like to thank my mentor, Professor Alessandro Margara, for his guidance in developing this thesis.

Many thanks to all my friends who enriched my studies and made it fun, exciting, and unforgettable. Without you, this story would not be the same.

The biggest thank you goes to my family, who supported me and were with me in all the easy and difficult moments. Without you, this story would not be possible.