Nubes: Object-Oriented Programming for Stateful Serverless Functions

Kinga Anna Marek
kingaanna.marek@mail.polimi.it
Politecnico di Milano
Italy

Luca De Martini
luca.demartini@polimi.it
Politecnico di Milano
Italy

Alessandro Margara
alessandro.margara@polimi.it
Politecnico di Milano
Italy

Abstract

Serveless computing and the Function-as-a-Service (FaaS) model promise rapid development of cloud-based applications by abstracting away deployment and resource allocation. As the stateless nature of functions undermines the generality of the model, they are often paired with storage services to persist their state. However, this approach exposes state management to developers, who need to manually encode the interactions between functions and storage. The relations between functions and state are hidden within function implementations, negatively affecting modularity and reuse.

To overcome these problems, we propose a novel abstraction that brings the benefits of object-oriented programming to FaaS, and we implement this abstraction into the Nubes framework. In Nubes, developers define objects that encapsulate state in the form of attributes and expose methods to other objects. Applications are written using familiar object-oriented concepts, Nubes then transparently and automatically manages the state of objects using a cloud storage service and handles the execution of serverless functions. Nubes simplifies application development and deployment and promotes the reuse of objects as composable building blocks for cloud applications. Using a case study, we show that Nubes significantly reduces code complexity with limited overhead with respect to manually crafted solutions.

Keywords: serverless, function as a service, cloud computing, stateful functions, object-oriented programming

ACM Reference Format:
Kinga Anna Marek, Luca De Martini, and Alessandro Margara. 2023. Nubes: Object-Oriented Programming for Stateful Serverless Functions. In *9th International Workshop on Serverless Computing (WoSC '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3631295.3631398

WoSC '23, December 11–15, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0455-0/23/12.
https://doi.org/10.1145/3631295.3631398

1 Introduction

Serverless computing [3] is a programming and execution paradigm that aims to simplify the development and operation of cloud applications by automating the management of the infrastructure where the applications run. In the serverless paradigm, developers implement the application components, and the cloud platform takes care of instantiating and running them, while hiding resource allocation and deployment concerns. The cloud platform may automatically and dynamically scale up or down the resources provided to each component based on demand. As pricing is typically computed per resource usage, this may further reduce operational costs.

The most common way to utilize the serverless paradigm is with a combination of Function-as-a-Service (FaaS) and Database-as-a-Service (DBaaS). With the FaaS model, developers encode the application logic by programming functions that the cloud platform instantiates and executes on demand. As functions are dynamically instantiated, their state is not persisted between invocations, which limits their practical applicability. To address this problem, functions are often paired with the use of databases, also offered as services, to store the application state. This approach decouples the storage layer from the compute layer, allowing them to scale independently and based on their heterogeneous needs.

However, this separation presents significant shortcomings. Developers need to manually implement database access, state query, retrieval, and update. The relation between state and functions is implicit and developers must manually enforce the assumptions made by different software modules in terms of state access. This negatively limits modularity, composability, and reusability of functions. Yet, modularity is a fundamental feature to build large-scale distributed applications that are easy to operate and maintain: indeed, recent architectural patterns, such as the microservices paradigm [6], focus on decomposing an application into independent components that expose restricted APIs which are used by the other components.

We see a symmetry between the current state of serverless functions and the downside of procedural programming languages, where the separation between data and functions is frequently seen as a limiting factor for modularity and reuse. In general-purpose programming languages, the object-oriented paradigm overcomes this problem by

bringing together state and functions within composable units denoted as objects.

Moving from this observation, we propose an object-oriented programming model for serverless computing and implement it in the Nubes prototype system. In Nubes, developers define classes that encapsulate part of the application state and expose methods to access and modify such state. Developers use classes to instantiate objects and implement complete applications, relying on familiar concepts of object-oriented programming, including state encapsulation and relations between objects. In this programming model, classes represent base building blocks that can be reused across applications.

Nubes automatically deploys the resulting application into a cloud platform. It translates methods to serverless functions that persist the state of objects into an external storage service. This approach preserves the benefits of delegating compute and storage functionalities to distinct services that can scale independently, but abstracts this implementation detail away from the programming model, relieving developers from the burden of explicit state management. Nubes optimizes various state access patterns that are typical of object-oriented programming, thus raising the level of abstraction without introducing significant performance overheads.

The paper is organized as follows. Section 2 presents the programming model offered by Nubes, and Section 3 details the design and implementation of the system. Section 4 evaluates the the proposed programming model and the performance of Nubes. Section 5 surveys related work, and Section 6 draws conclusions and indicates possible areas of future work.

2 Programming Interface

This section presents the programming interface of Nubes, which allows developers to write object-oriented applications that are automatically and transparently deployed and executed in a serverless environment. Figure 1 shows the high-level view of a Nubes application: one or more *client programs* (or simply *clients*) create or obtain (load) references to objects that live in a serverless environment. Their methods are executed within serverless functions and their state (attributes) is persisted in a cloud storage service. Nubes is written in Go: writing client programs in Go offers a seamless integration between the code of the client and the code executed in the serverless environment. Nonetheless, developers may also invoke objects methods as standard serverless functions from other programming languages, for instance, from a Web-based client written in Javascript.

The development workflow of a Nubes application involves four phases (see Figure 2): (1) *Classes definition*: developers define the types of objects (classes) they use in their applications. Classes are the blueprints for objects, which encapsulate state (attributes) and behaviors (methods). (2) *Compilation*: a `compiler` module automatically translates the classes into

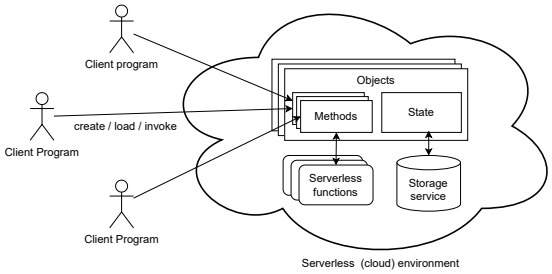

Figure 1. Overview of the Nubes architecture.

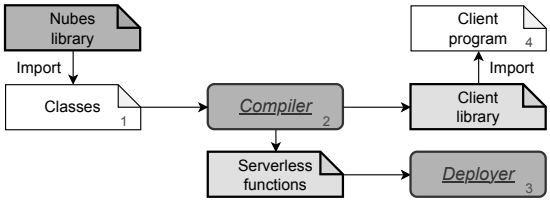

Figure 2. Components for developing of a Nubes application.

a form that can be deployed and executed in a serverless environment. The compilation process produces server-side and client-side components. Server-side, it produces the *serverless functions* to be deployed on the serverless environment. Client-side, it produces a *client library*, containing a modified version of the original classes that automatically converts invocations of local methods to invocations of serverless functions. (3) *Deployment*: serverless functions are deployed onto the serverless environment and the storage service is initialized using the provided `deployer` module. (4) *Client development*: developers import and use the classes defined in the client library to instantiate concrete objects and define the specific behavior of the application at hand.

2.1 Class definitions

Nubes needs to uniquely identify each class and each object instantiated from a class. To do this, developers need to provide a unique class name by implementing the `GetTypeName` method. Likewise, developers indicate a set of *key* attributes for each class using an annotation – a *tag* in Go. Each object of the class will then be uniquely identified by the values of the key attributes (see the example in Listing 1). Alternatively, a class can include an `Id` attribute of type string that will serve as the key and it will be automatically generated by Nubes at runtime.

```
type User struct {
  Email    string `nubes:"key"`
  Name     string
}
```

Listing 1. Example of class definition.

2.2 Object lifecycle

When seen from a client, objects will be in one of three states: *unexported*, *exported*, *deleted*.

Unexported objects are objects that have been created locally, using normal constructors, but have not been persisted to the storage. Invocation to methods of an unexported object will modify it within the client program. An unexported object can then be *exported* by invoking the `Export` function. Its state will be persisted in the storage and it will then be accessible from other client applications.

Developers should interact with exported objects only by invoking their public methods as they will automatically load and save the state of the object when needed. For instance, the following Listing 2 initializes an object of type `User`, sets the value of its fields, and exports it. The `Export` function returns a reference to the exported object, and an error code that signals the outcome of the invocation. The last line of Listing 2 shows how developers can seamlessly invoke methods on exported objects.

```
user := User {
  Email:   "user1@nubes.org",
  Name:    "user1",
}
exportedUser, err := lib.Export[User](user)
exportedUser.Method()
```

Listing 2. Initializing, exporting, and using an object.

Clients can also retrieve objects that have been exported by themselves or a another client. This is done with the the `Load` function, which takes in input the unique identifier of the object (class name and key). Finally, Nubes provides a `Delete` function to permanently delete an exported object. Developers should not interact with deleted objects, and invocations to methods on a deleted object always return an error.

The behavior of `Export` and `Delete` for a given class can be overwritten with a custom implementation. For instance, the `Export` function for an `Order` class may be customized to decrease the availability of a product every time an order for that product is exported.

2.3 Relationships

A key feature in object-oriented design is the ability to express relationships between objects, which translates to objects holding references to other objects as part of their state. Nubes fully supports this feature and offers suitable constructs to build various types of references.

Unidirectional relationships. In unidirectional relationships, an object references another object, but there is no inverse relationship. Nubes supports both one-to-one and one-to-many unidirectional relationships through the special `nubes.Ref` and `nubes.RefList` types, which can be used when defining classes. The `nubes.Ref` class has an associated `Get` method that returns a reference to the object associated to the relationship. Likewise, the `nubes.RefList` class has a `GetAt` method that takes in input an integer i and retrieves the reference to the i^{th} object associated to the relationship. Finally, the `nubes.RefList` class provides the `Get` method to return all references associated to the relationship with a single storage access.

Bidirectional relationships. Bidirectional relationships between two objects o_1 and o_2 are used to define both a reference from o_1 to o_2 and a reference from o_2 to o_1. Nubes defines bidirectional relationships using annotations: given a relationship between class c_1 and class c_2, (defined in c_1) developers can make the relationship bidirectional by adding a field of type `BiRefList` within c_2 with an annotation to indicate the name of the original relationship. The `hasOne` annotation prefix marks the relationship as one-to-many, while `hasMany` marks it as many-to-many.

To exemplify, the following Listing 3 defines a `Product` class and a `Shop` class. The `Product` has unidirectional references to one `Shop` and to a list of `Discount` objects. The `Shop` holds the references to all of its products and indicates that the relation is bidirectional one-to-many and has its inverse reference in the `SoldBy` attribute of `Product`.

```
type Product struct { /* ...attributes */
  SoldBy     nubes.Ref[Shop]
  Discounts  nubes.RefList[Discount]
}
type Shop struct { /* ...attributes */
  Name     string
  Products
     nubes.BiRef[Product] `nubes:"hasOne-SoldBy"`
}
```

Listing 3. Example of a bidirectional relationship.

3 System Implementation

Nubes is implemented in Go and is available as an open-source project in GitHub[1]. Although it can be adapted to use different technologies, the current implementation works in the AWS Lambda serverless environment and persists the state of objects in DynamoDB [7].

Consider again Figure 2, which shows the components and artifacts involved in the development of a Nubes application: dark gray components and artifacts are provided by Nubes, light gray artifacts are automatically generated by Nubes, white artifacts are provided by the application developers.

The `compiler` is responsible for translating classes into a form that can be executed in the serverless environment: the developers write the classes using the types and methods provided by the nubes library (`nubes.Ref`, `Export`, ...), the compiler then performs source-to-source compilation to generate (i) the definition of *serverless functions* that the `deployer` can install onto the serverless environment. (ii) a *client library* with the modified classes that client programs import and use to interact with the objects; The `compiler` uses the `ast` and `text/template` packages from the Go standard library to read and modify the abstract syntax tree of the defined

[1] https://github.com/deib-polimi/Nubes

classes. It modifies the functions to correctly interact with the remote storage and to invoke remote functions when needed.

Serverless functions. The compiler analyzes each class that implements the `GetTypeName` method to generate serverless functions. Each public method of the class is compiled to a serverless function that is uniquely identified with the combination of the type name and the name of the method. The generated function take as input a JSON struct, which contains (i) the unique identifier of the object on which the method is invoked (see Section 2.1); (ii) the input parameters of the method (if any). The code of the generated functions is similar to the original methods, however, accesses to the object state are converted to invocations to the storage service, as discussed later in Section 3.1. Additionally, all invocations of other object methods in the code are executed synchronously within the same serverless function.

As an example, when a client calls the `Buy` method as defined in Listing 4, it will trigger the execution of one serverless function, which will execute the body of the method, including the two invocations to the `shop` identified by the `SoldBy` reference.

```
func (p *Product) Buy() error {
  shop, err := p.SoldBy.Get()
  shop.DecreaseAvailability(p.Id)
  shop.IncreaseIncome(p.Price)
  return nil
}
```

Listing 4. Example of method invocation.

Client library. The client library contains a modified version of each defined class, intended to be imported by the client code. The modified classes (i) keep track of the lifecycle of the object, to know whether the object is exported or not (see Section 2.2); (ii) have modified methods that transparently invoke the corresponding serverless function for exported objects.

Even without the library, developers can interact with serverless functions from other programming languages if they use the correct input parameters – the standard approach in public cloud services. They still benefit from an object-oriented definition of server-side components and from their automated deployment on the cloud.

3.1 State management

Nubes stores the state of exported objects in the DynamoDB storage service. Each class corresponds to a DynamoDB table, where columns represent the attributes of the class and rows represent objects instances of that class. Every table has a key that stores the unique identifier of an object instance, determined through the set of key attributes as explained in Section 2.1. We implemented several optimizations to reduce the interaction with the storage when possible. First, when the `compiler` can statically determine that a method does not modify the state of the object, it avoids writing back the values of attributes to the storage at the end of the method execution. Moreover, in the case of getters and setters, the database interaction is restricted to the attribute being read or written, avoiding access and (de-)serialization of other attributes. Finally, in a chain of methods invocations where each method invokes (directly or indirectly) another method of the same object, only the outermost method interacts with the storage service.

References. We carefully optimized the case of bi-directional relationships: in order to efficiently retrieve the inverse of a one-to-one or one-to-many relationship from c_1 to c_2, we define a secondary index on c_2 based on the key of c_1. This has two advantages: (i) it is efficient to navigate the relationship in both directions; (ii) relationship changes are automatically propagated by the storage service with optimized mechanisms. In case of many-to-many bi-directional relationship, we use an intermediate table storing the key pairs that define the relationship between the two classes.

4 Evaluation

We evaluate Nubes considering two criteria: (i) effectiveness of the programming model in reducing the complexity of developing serverless applications; (ii) performance and scalability of applications developed with Nubes.

We compare Nubes with the standard methodology for developing stateful serverless functions, where state management is explicit and developers need to manually control the interactions with an external storage service.

4.1 Experiment setup

For our evaluation, we developed a realistic serverless application using SSF and Nubes, and we deployed it on the Amazon AWS cloud infrastructure.

Application. The application we use for our evaluation derives from DeathStarBench [2], an open-source benchmark suite for microservices applications[2]. It implements the server side for a hotel booking service and exposes the functionalities presented in Table 1. We chose this application for two reasons: (i) it has been used to asses several recent proposals in the area of serverless functions [4, 9]. (ii) it is complex enough to capture all the features of Nubes, including all kinds of relationships between objects, both in cardinality and in directionalty.

[2] https://github.com/delimitrou/DeathStarBench

Task	Features under test	Access mode
register-user	export	`rw`
delete-user	delete	`rw`
set-hotel-rate	update	`rw`
login	object method	`ro`
get-hotels	get all 1:n rel.	`ro`
recommend	get inverse 1:n rel.	`ro`
reserve	export, update, n:m rel.	`rw`
get-user-reservations	get all 1:n rel.	`ro`

Table 1. List of tasks (`ro`: read-only, `rw`: read-write).

The database is populated with 50000 users, 5 cities, 100 hotels per city, 25 rooms per hotel, and 20 reservations per room, totaling over 12.5k rooms and 250k reservations.

Systems under test. We consider three implementations of the hotel booking service. (1) **Nubes** adopts the development methodology presented in this paper. (2) **SSF** represents a basic serverless implementation using the AWS Lambda and DynamoDB services. This implementation uses the standard methodology for developing stateful services with explicit state management. This version has similar access patterns to the Nubes implementation. (3) **SSF-custom** adopts the same methodology as SSF, but customizes the layout of the state within the storage service to the specific workload (tasks) we are evaluating. The main difference between Nubes and SSF-custom is that SSF-custom adopts a de-normalized approach when it is beneficial for performance (as discussed in the DynamoDB documentation[3]) and uses composite keys that allow range queries when possible.

Cloud environment. We run all tests using the AWS public cloud: we use AWS Lambda functions for the computation and DynamoDB as a storage service. Lambdas are configured with 1GB of memory and DynamoDB is configured in *on-demand* to allow both compute and data to scale freely. The client used for the evaluation and latency measurement is an EC2 virtual machine (`t3a.2xlarge`, 8 vCPU, 32GB RAM, 5Gbps burst bandwidth) located in the same region as the serverless application.

4.2 Code complexity

In this section, we analyze and compare the programming models of Nubes and SSF. We recognize that assessing code complexity is difficult and highly subjective, and we approach the problem by (i) measuring the number of lines of code of the applications as coarse-grained indication of complexity; (ii) discussing the differences in the concerns that each programming models exposes to the developers that may contribute to code complexity.

System	LoC written	LoC generated	LoC total
SSF	1020	0	1020
Nubes	369	603	972

Table 2. Lines of code of the hotel booking application.

Table 2 shows the number of lines of code to develop the hotel booking application. We report and comment only the numbers for SSF since the lines of code in SSF-custom are nearly identical. The SSF approach requires developers to write 2.75× more lines of code than Nubes. This is due to the fact that SSF requires developers to write code that is otherwise generated in Nubes: interactions with the storage, serialization, deployment scripts, storage table setup, etc.

[3] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html

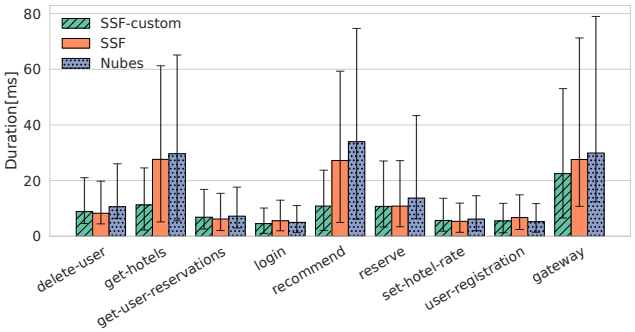

Figure 3. Lambda duration (10th, 50th, 90th percentile)

4.3 Performance comparison

In the evaluation, the client submits the tasks presented in Table 1 with uniform distribution at a rate of 100 invocations per second for over 2 minute, with 15s warmup that is discarded. As usual in the function-as-a-service model, the requests are submitted to a gateway that dispatches them to other functions.

Duration As a first metric of performance, we measure the execution time of each serverless function involved in the application, as returned by the *duration* metric within the AWS CloudWatch. As Lambdas are billed by execution time, this measure is directly correlated to the cost of running the application. Figure 3 shows the results we measured, grouped by task. For all tasks, Nubes presents results that are comparable to the baseline SSF approach. The maximum overhead we measure is about 23% of the median duration, in the `recommend` task, where Nubes first accesses an index to obtain the references of hotels for a specified city, and then retrieves the desired hotels, whereas SSF can directly query the table containing all hotels and filter by city. As a metric of the average overhead over all tasks, we can consider the duration of the `gateway` function: we see that Nubes introduces a mean overhead of 10% with respect to the SSF median duration.

SSF-custom achieves faster execution in the `get-hotels` and `recommend` tasks, this is expected given the manually optimized, service-specific layout that stores information about cities and hotels in a single table. However, this approach complicates the development process and must be adapted to specific tasks and storage solutions. Conversely, Nubes offers a general solution that is easy to develop and modify, and can be ported to different storage services.

Latency. When looking at the latency measured from the client, we see a mean response time of 43.1 ms for Nubes, compared to 40.4 ms for SSF and 37.3 ms for SSF-custom. The absolute differences between the three implementations reflect the differences in the duration of functions as reported above. However, as the communication between the client and the functions introduces a delay for all implementations, when looking at the relative differences, using Nubes only introduces a 7.5% increase in latency with respect to SSF and a 16% increase in latency with respect to SSF-custom.

The relative difference will also be lower in a geographically distributed scenario, as the contribution of network latency becomes more dominant.

5 Related Work

In this section, we present abstractions for application development and for state management, in the context of serverless computing, which are the most strictly related to our work.

Workflows. Most abstractions proposed for serverless functions target the composition of functions in workflows. Popular public cloud providers support function workflows[4][5][6]. We see this line of investigation as orthogonal to our proposal, as it targets the composition and coordination of functions to achieve data processing tasks rather then abstracting away and hiding data management concerns.

Abstractions for stateful functions. Several programming abstractions proposed for serverless functions include state-management capabilities and share some similarities with what we propose with Nubes. Cloudburst [8] offers a programming interface that simplifies the interaction between functions and state. In contrast to Nubes, developers still need to explicitly define and export state elements in their code. Objects as a Service (OaaS) [5] is a research proposal that shares several similarities with Nubes. However, it deals with immutable data elements: functions and do not modify the objects, but output they new state. Additionally, it does not assist in the configuration and deployment of the functions as is done in Nubes. Cloudflare Durable Objects[7] offers a programming abstraction that builds on the concept of *objects* as stateful entities that are uniquely identified by an id. The use of object-oriented concepts, however, ends with encapsulation of the state in objects. Particularly, the interactions with the objects happen solely through a *fetch* method defined for the specific object type, which enables clients to retrieve the content of a specific object. The Durable Function programming model [1] target the expressivity of workflows definition. Nubes shares with Durable Functions the idea of defining an application using a standard programming language. However, Durable Functions expose the concepts of activities (functions) and entities (objects) to developers, while Nubes fully hides the adoption of serverless abstractions.

Execution guarantees. A problem related to serverless environments is the lack of guarantees on their actual execution: for instance, public cloud environments introduce timeouts that limit the maximum execution time of functions, to prevent the cost of long-running executions caused by bugs [9]. To address these issues, several research works propose mechanisms to offer some forms of atomicity and concurrency control in workflows of stateful serverless functions [4, 9].

[4] https://aws.amazon.com/step-functions/
[5] https://learn.microsoft.com/azure/azure-functions/durable
[6] https://cloud.google.com/composer/
[7] https://developers.cloudflare.com/durable-objects/

The above works are orthogonal to the idea of providing new programming abstractions to simplify the development of applications in serverless environment. Leveraging their contributions, we plan to study fault-tolerance and isolation semantics for the Nubes model in the future: we foresee the possibility to define critical sections within methods that need to be executed in isolation.

6 Conclusions

This paper proposed and evaluated a novel programming model for serverless computing that builds on object-oriented abstractions. Nubes significantly reduces complexity with respect to the standard development process for stateful serverless functions while introducing limited overhead even with respect to custom implementations that sacrifice generality and reusability for performance under specific use cases.

In summary, Nubes offers a familiar programming model to implement distributed applications, which are automatically deployed and operated in a serverless environment, obtaining the benefit of automated scaling based on the actual load. Our plans for future work include the integration of fault-tolerance and consistency guarantees within Nubes, with the goal of covering a wider range of applications requirements.

References

[1] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (2021), 27 pages.

[2] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems *(ASPLOS '19)*. ACM, 3–18.

[3] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back *(CIDR'19)*.

[4] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs *(SOSP '21)*. 691–707.

[5] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. 2022. Object as a Service (OaaS): Enabling Object Abstraction in Serverless Clouds.

[6] James Lewis and Martin Fowler. 2016. Microservices, a definition of this new architectural term. https://martinfowler.com/articles/microservices.html

[7] Swaminathan Sivasubramanian. 2012. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service *(SIGMOD '12)*. ACM, 729–730.

[8] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of VLDB Endow.* 13, 12 (2020), 2438–2452.

[9] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-Tolerant and Transactional Stateful Serverless Workflows *(OSDI'20)*. USENIX Association, Article 67, 18 pages.