Object as a Service (OaaS): Enabling Object Abstraction in Serverless Clouds

Pawissanutt Lertpongrujikorn, Mohsen Amini Salehi
High Performance Cloud Computing (HPCC) Lab,
School of Computing and Informatics, University of Louisiana at Lafayette
{*pawissanutt.lertpongrujikorn*1, *amini*}*@louisiana.edu*

***Abstract*—Function as a Service (FaaS) paradigm is becoming widespread and is envisioned as the next generation of cloud computing systems that mitigate the burden for programmers and cloud solution architects. However, the FaaS abstraction only makes the cloud resource management aspects transparent but does not deal with the application data aspects. As such, developers have to intervene and undergo the burden of managing the application data, often via separate cloud services (e.g., AWS S3). Similarly, the FaaS abstraction does not natively support function workflow, hence, the developers often have to work with workflow orchestration services (e.g., AWS Step Functions) to build workflows. Moreover, they have to explicitly navigate the data throughout the workflow. To overcome these inherent problems of FaaS, our hypothesis is to design a higher-level cloud programming abstraction that can hide the complexities and mitigate the burden of developing cloud-native application development. Accordingly, in this research, we borrow the notion of object from object-oriented programming and propose a new abstraction level atop the function abstraction, known as Object as a Service (OaaS). OaaS encapsulates the application data and function into the object abstraction and relieves the developers from resource and data management burdens. It also unlocks opportunities for built-in optimization features, such as software reusability, data locality, and caching. OaaS natively supports dataflow programming such that developers define a workflow of functions transparently without getting involved in data navigation, synchronization, and parallelism aspects. We implemented a prototype of the OaaS platform and evaluated it under real-world settings against state-of-the-art platforms regarding the imposed overhead, scalability, and ease of use. The results demonstrate that OaaS streamlines cloud programming and offers scalability with an insignificant overhead to the underlying cloud system.**

***Index Terms*—FaaS, Serverless paradigm, Cloud computing, Cloud-native programming, Abstraction.**

I. Introduction

A. FaaS and Its Problems

Function-as-a-Service (FaaS) paradigm is getting widespread and is envisioned as the next generation of cloud computing systems [27] that mitigates the burden for both programmers and cloud solution architects. Major public cloud providers offer FaaS services, and several open-source platforms for on-premise FaaS deployments are emerging too. FaaS offers the function abstraction that allows users to develop their business logic and invoke it via a predefined trigger. In the back end, the serverless platform hides the complexity of resource management details and deploys the function in a seamless and scalable manner. In particular, the platform enables FaaS to be truly pay-as-you-go via scale-to-zero and charging the user only upon a function invocation. FaaS is proven to reduce development and operation costs, thus, is in alignment with modern software development paradigms, such as CI/CD and DevOps [8].

As the FaaS paradigm is primarily centered around the notion of stateless *functions*, it naturally does not deal with the *data*. Thus, the developers have to intervene and undergo the burden of managing the application data using separate cloud services (e.g., AWS DynamoDB [6] and AWS S3 [5]). That is, although FaaS makes the resource management details (e.g., load balancing and scaling) transparent from the developer's perspective, it does not do so for the data. Even though stateless functions make the system scalable and manageable, the state still exists in the external data store, and the developer must intervene to connect the running service to the data store. For instance, in a video streaming application, in addition to developing the functions, the developer has to maintain the video files, their metadata, and manage the access to them.

Apart from the data management aspect, current FaaS systems do not offer any built-in semantics to limit access to the internal (a.k.a. private) mechanics of the functions. Nevertheless, providing unrestricted access to the developer team has known side effects, such as function invocation in an unintended context and data corruption via direct data manipulation. To overcome such side-effects, developers again need to intervene and undergo the burden of configuring external services (e.g., AWS IAM [2] and API gateway [1]) to enable access control.

Last but not least, current FaaS abstractions do not natively support function workflows. To pipeline functions and form a workflow, for each function, the developer has to generate an event that triggers another function in the workflow. However, for large workflows, configuring and managing the chain of events become complex and add a burden to the developer. Although function orchestrator services (e.g., AWS Step Function [4] and Azure Durable Function [10]) can be employed to mitigate this burden for the developers, the lack of data management in FaaS forces the developer to intervene and employ other cloud services to navigate the data throughout the workflow manually.

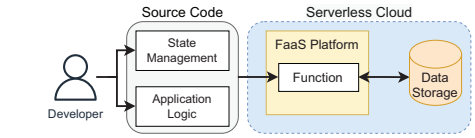
B. Our Motivation and Proposed Solution

To overcome these inherent problems of FaaS, we propose a new paradigm on top of the function abstraction that not

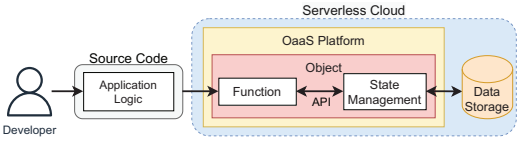
2159-6190/23/$31.00 ©2023 IEEE
DOI 10.1109/CLOUD60044.2023.00035

only mitigates the burden of resource management but also mitigates the burden of data management from the developer's perspective. *We borrow the notion of "object" from the object-oriented programming, and develop a new abstraction level within the serverless paradigm, known as* ***Object as a Service (OaaS)***. Incorporating the application data into the object abstraction unlocks opportunities for built-in optimization features, such as data locality, data reliability, caching, and software reusability [16]. Moreover, objects in OaaS offer developers encapsulation and abstraction benefits in addition to the ability to transparently define workflows of functions (a.k.a. dataflow programming [43]) in the cloud.

Our motivation in this study is a cloud-based video streaming system [15], [37] that needs developers to implement new streaming services for the available video content rapidly. A few examples of such services are: Generating multilingual subtitles for safety-related videos; Removing harmful and illicit content from child-safe videos, And developing a face detection service on the surveillance videos. Implementing these services using FaaS entails dealing with the data (i.e., videos), in addition to developing the business logic. In this scenario, the OaaS paradigm can mitigate the developer's job by offering the encapsulation semantic. The video is defined as object that contains its state and is bound to a set of functions that can be called by the viewer's application and potentially change the object (video) state. For instance, the request to generate Chinese subtitles for a video object invokes `subtitle(chinese)` function of that particular object.



(a) Function as a Service (FaaS)



(b) Object as a Service (OaaS)

Fig. 1: A bird-eye view of FaaS vs. OaaS. In FaaS, the developer must implement the application logic and the state management in the form of function(s) interacting with the developer-provisioned storage. In OaaS, the developer only develops the application logic and deploys it as an object with builtin state management.

As we can see in Figure 1, unlike FaaS, OaaS segregates the state management from the developer's source code and incorporates it into the serverless platform to make it transparent from the developer's perspective. In this case, the object's function only includes the business logic, and upon invocation, the OaaS platform executes the function and then manages the object state (via API calls). In addition to enabling encapsulation of the function and the state to form the *object* abstraction, the OaaS platform offers features including: *macro functions* to objects that facilitate dataflow programming; and *templates* (analogous to the notion of *class* in OOP) to developers that simplify defining properties and functions of the object(s). We note that object abstraction is not a replacement for FaaS. Instead, it is a complement for it. There are use cases that are naturally stateless (e.g., mathematical functions), and FaaS is the appropriate solution.

C. Challenges and Contributions

The *first* and foremost challenge that has to be addressed for OaaS is how to offer the object abstraction as a new cloud-native programming paradigm? Addressing this challenge entails dealing with other problems: (a) How to enable encapsulation of data and functions at the cloud level such that the internal mechanics of the object is hidden and only the necessary functionalities are exposed? (b) How to handle workflow functions that potentially include multiple other function calls and seamlessly manage data navigation throughout these functions? (c) How to define and handle high-level objects that are composed of other objects?

Although enabling the OaaS paradigm is advantageous, it is not free of charge. The challenge is that developing the OaaS platform on top of FaaS entails unavoidable overheads. This is because moving the data between OaaS components increases the overhead of function invocation. This is particularly important for unstructured (binary) data that is usually persisted on a different type of cloud storage, such as object storage. In fact, the *second* challenge is how to design the OaaS platform such that the overhead is minimal and tractable? The *third* challenge is the scalability of the object access. Specifically, concurrent accesses to an object can lead to the race condition on the state and must be controlled to avoid data inconsistency. Synchronization mechanisms to order the invocations can protect the state. However, they cause a bottleneck and downgrade the scalability of the OaaS platform.

For the first challenge, OaaS offers an interface for developers to declare the behavior of objects in the form of `class` and `function`. This interface also includes native (built-in) workflow semantics and access modifiers to enable encapsulation over objects. In workflow management, OaaS instead offers it based on the dataflow semantics that hides the detail of synchronization via defining the flow of data. The dataflow is registered as a function and can be called in the same way as any function. Thus, it hides the implementation details from other developers and users. Additionally, the object can have the references linked to other objects and form the dataflow function on top of them. This object is exposed as a high-level object and hides the detail of a low-level object, which can be achieved by declaring the access modifiers. Therefore, OaaS will reject any function calls or dataflow declarations that involve invalid access.

To address the second challenge, in our initial experiments, we realized that the overhead of the OaaS platform is mainly due to the latency of accessing the object state. To reduce the overhead, we develop a *data tiering mechanism* within the OaaS platform that diminishes the latency of accessing the object. The tiering mechanism uses a key-value database to store the object specifications (a.k.a. metadata) that are accessed frequently, in addition to an in-memory caching to

accelerate accessing the infrequently-updated but frequently-accessed metadata (e.g., class and function specifications). OaaS also reduces unnecessary data movements within the platform via employing a *redirection mechanism* instead of relaying (transferring) the object state.

To address the third challenge and keep the object scalability in check, we design the OaaS based on the microservices architecture with the minimum contention between the self-contained services. OaaS also minimizes object state synchronization by implementing the *immutable* data processing model. That is, upon invoking an object function, the platform outputs a new/updated state instead of updating the existing one. Implementing this semantic makes the function perform a stateless operation and keep the state consistent without synchronization, thereby appearing stateful at a high level.

In sum, this research proposes the OaaS paradigm that extends FaaS to offer object abstraction to cloud developers. The OaaS platform provides stateful objects with minimal overhead while maintaining serverless characteristics. The key contributions of this research are as follows:

1) Developing the OaaS paradigm to hide data and resource management complexity from the user's view.
2) Implementing a working prototype of the OaaS platform [1] that can support both structured and unstructured states.
3) Devising mechanisms based on data tiering and caching and object immutability to minimize the imposed overhead of OaaS and improve its scalability.
4) Analyzing the performance of OaaS from the scalability, overhead, and ease-of-use perspectives.

In the rest of this paper, Section II reviews the state of the art in the serverless paradigm. Section III discusses the conceptual design and the architecture of OaaS. Section V evaluates the overhead, scalability, and development efficiency of OaaS. Finally, we conclude this paper in Section VI.

II. Background and Prior Studies

The FaaS paradigm allows the developer to implement the application as a set of independent functions transparently provisioned in isolation on the cloud infrastructure. FaaS is offered by public cloud providers (e.g., AWS Lambda [3], Azure Function [39], Google Function [12]). FaaS can also be self-hosted via open-source platforms (e.g., OpenFaaS [19], and OpenWhisk [21]). FaaS invokes the function upon receiving the event that matches its predefined trigger(s).

A variant of FaaS, Container as a Service (CaaS) [29], does not offer the function development framework. Instead, the user must provide the already-containerized function. Kubernetes [13] is a widely-used platform that automates container provisioning and manages the life cycle of containerized services. Knative [23] complements Kubernetes by enabling CaaS and is composed of two main components: *Knative Serving*, and *Knative Eventing*. The former enables the auto-scaling, scale to zero, and minimal configuration of the containerized services. The latter enables pipelining and routing events to streamline developing event-driven applications.

[1]The OaaS source code is available here: https://github.com/hpcclab/OaaS

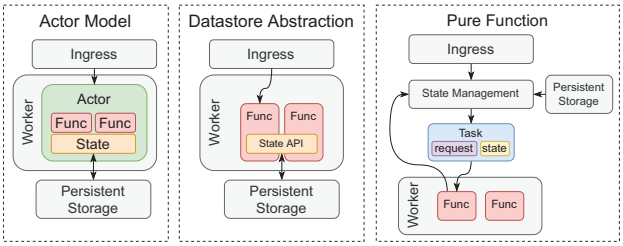


Fig. 2: The illustrated comparison of three different models of stateful serverless.

	Actor Model	Datastore Abstraction	Pure Function
Data placement	worker instances	platform services or database (depend on impl.)	platform services or database
Complexity	high	depend on impl.	low
Data locality	high	depend on impl.	low
Unstructured data support	difficult	yes	yes
Deployment granularity	actor (multiple function)	function	function
Maintain-ability	low	high	high
Solutions	Kalix [30], Azure Entity Func. [40]	Cloudburst [44], FAASM [42], Apiary [36], OaaS	Kalix [30] eigr [18], Statefun [7], OaaS

TABLE I: Comparing properties of various design patterns to build a stateful serverless platform.

The idea of stateful serverless is explored in several research works (e.g., [9], [38], [46]). As noted in Figure 2 and Table I, these works can be categorized into *actor model*, *datastore abstraction*, and *pure function* approaches depending on where the platform stores the state data and how the function accesses the data. According to Figure 2, the actor model places the state inside the worker instance to achieve the data locality. In the pure function, the state is placed on other services (e.g., database) and is transferred to the worker instance upon invocation. Hence, the state appears as part of the function input argument, and the modified state appears as its output. Thus, the function is still stateless while exhibiting stateful features. Lastly, the datastore abstraction is a hybrid approach where the platform provides the API for the function to access the data on demand. Depending on the design, the state can be stored in the database but can be cached in the worker too.

According to Table I, the actor model serverless platform needs to maintain the availability of each actor where both data and compute reside. Maintainability is particularly difficult for bulky unstructured data because the platform needs to balance the computing and storage utilization on each node. In addition, the platform has to support a routing mechanism to navigate a function call to the actor's location. Alternatively, the pure function approach disaggregates the state management and compute (function) for the sake of system design

simplicity. However, it compromises the data locality aspect. Similar to pure functions, datastore abstraction also relaxes the need to store the state on the worker node. Regardless, it utilizes caching techniques to preserve data locality. The deployment granularity of the actor model approach is an actor with multiple functions that share the same state, whereas the granularity in other approaches is a single function.

The actor model approach has been popular in programming languages and OOP because it spurs asynchronous messaging across actors, and it lends itself to distributed deployments. That is why it has been an attractive choice for stateful serverless platforms, even though it poorly supports unstructured data. Kalix [30] and Azure Entity Functions [40], which are part of Azure durable functions, are example platforms implemented based on the actor model approach. The serverless platforms based on datastore abstraction are mostly popular in the research area. Cloudburst [44] offers stateful functions using a shared distributed key-value database to keep track of the state. FAASM [42] optimizes the function-state interaction overhead via employing web assembly [26] instead of containers [25], [41] for function isolation. Even though web assembly enables multiple functions to share the memory and achieve data locality, it implies compiling the code into web assembly, which limits the usage of operating system APIs. Apache Flink Stateful Function (StateFun) [7], eigr [18], and Kalix [30] are solutions based on the pure function approach. StateFun is built atop Apache Flink, which is based on the actor model. However, it offloads the function code to a dedicated node, thus, is categorized under the pure function approach.

As OaaS intends to support both unstructured and structured state data efficiently, we chose to develop it based on the pure function approach. However, for unstructured data, OaaS allows the function to fetch the state on demand, as opposed to including it as an input argument. Hence, OaaS is practically between the pure function and datastore abstraction approaches. Furthermore, OaaS supports the notion of object that is beyond only stateful functions and provides abstraction, encapsulation, inheritance, dataflow programming, and polymorphism within the serverless paradigm.

Cherrier et al. [11] used the notion of Object as a Service to establish Services Oriented Computing in the context of IoT. They model the IoT system using objects where sensors are data-gathering objects and actions (functions) are the actuators. This differs from our OaaS that borrows the notion of object from OOP to establish the object abstraction in serverless.

III. Object as a Service (OaaS) Paradigm

A. Design Goals

To accomplish the goal of providing a high-level abstraction for cloud developers, OaaS should fulfill five objectives:

First, developing the concept of *object* in OaaS that can provide abstraction and encapsulation across data and functions in the cloud. Moreover, developing the notion of *class* to define a group of objects with the same characteristics. For instance, using the notion of class, a video stream provider who is developing an application for disabled viewers [15] can define the `accessible_videos` class and assign all the accessibility functions to it (e.g., `gen_subtitle(lang)` for deaf viewers; and `inc_contrast()` for color-blind users). Without the notion of class, the developer has to assign functions to each individual video, which is tedious and error-prone, whereas using class, several videos are defined as the object instances of the class. That is, the notion of class provides a "type" for a set of objects that are otherwise untyped. Furthermore, class enables the notion of *access modifier* for each function, thereby realizing encapsulation and access control for them.

Second, OaaS needs to provide transparency in the object state management and workflow defining. Fulfilling this objective realizes the notion of dataflow programming [43] that allows developers to define a workflow without getting involved in the concurrency and synchronization complexities. To allow the developer to access an object in the workflow without the knowledge of its status (i.e., whether or not the object is instantiated in the workflow), the OaaS platform exposes the *object access interface* (OAI) that enables the developer to invoke a function, request the object state, or both in a single request. For instance, while the first user is invoking the `inc_contrast()` function for `video1` and the new object (`video2`) is being created in the output, the second user can invoke the `gen_subtitle(CN)` function on `video2`, and OaaS handles the ordering of invocations transparently.

Third, OaaS must efficiently support both *structured* (e.g., JSON) and *unstructured* state data (e.g., video contents) for the objects to make them usable for a wide range of applications.

Fourth, to maximize the *extensibility* via employing the pure function model that separates the control plane from the execution plane. This enables OaaS to be extensible and can accommodate various types of execution planes optimized for the requirements of different use cases, e.g., supporting latency-constrained function calls.

Fifth, to accomplish *robustness*, OaaS must be designed with modularity and scalability in mind. To that end, OaaS is developed as a set of loosely-coupled services on top of the Knative serverless system. Each OaaS component is stateless and preserves the state on a scalable distributed database.

B. Conceptual Modeling of OaaS

In OaaS, an *object* is defined as an immutable entity with a *state* (i.e., data) that is associated with one or more *functions*. The state is, in fact, the application data that can be in a structured or unstructured form. Upon calling a function, a task is created that can take action on the state. A function can have one or more objects as its input. However, it cannot modify them. Each object is instantiated from a class and is bound to the set of functions and state(s) declared in that class.

To enable higher-level abstractions for the users or developers, the OaaS platform allows combining (nesting) objects into one. The high-level object holds a reference to the lower-level object(s), and the invoked function can leverage the reference to fetch the lower-level object as the input. Moreover, it is possible that the high-level object implements a new function

(called *macro function*) and invokes a chain of functions from the lower-level objects. This resembles configuring a workflow in conventional FaaS systems. The major difference between macro functions and function workflows is that macro functions introduce the flow of execution via the flow of data (transferring state) rather than the invocation order. Given the dataflow semantic and immutable nature of the objects, the execution flow in a macro function is determined by the flow of data, and the developers only need to introduce the flow. Then, in the background, OaaS takes care of the concurrency and synchronization and guarantees state consistency.

Listing 1: An example script that declares a class, named `test1`, and a function for it, named `concat`, in the YAML format.

```
1  classes:
2    - name: test1
3      stateSpec:
4        provider: s3
5        keySpecs:
6          - name: str
7      functions:
8        - access: PUBLIC
9          function: concat
10 functions:
11   - name: concat
12     type: TASK
13     outputClass: test1
14     provision:
15       knative:
16         image: concat:latest
17         ...
18 package: example
```

As shown in Figure 3, OaaS supports two user scenarios: (A) The service provider (developer) who declares the class and its functions for developing the application. (B) The end-user who accesses the objects (e.g., via an application or a web front-end) and calls their functions via the object access interface. Declaring a new class and its functions in OaaS are achieved using the YAML (or JSON) format. Listing 1 represents a declaration example for a class called `test1` that has a state named `str` (Line 6) and a function named `concat` (Line 11). The state is named `str` and is a `s3` object storage. The class has a public function called `concat`. The specifications of the function are declared in Lines 10—16. The `type` of a function (Line 12) can be a `task` (or a `macro`). Because the objects in OaaS are immutable, Line 13 specifies that the output of the function is another object instance of type `test1`. Line 16 declares the function container image URI. Declaring the function input(s) makes this example long.

C. OaaS Architecture

The OaaS platform is designed based on multiple self-contained microservices communicating within a serverless system. Figure 3 provides a birds-eye view of the OaaS architecture that is composed of four modules: (a) *Object Control Module* serves as the interface to instantiate, manage, and use the objects; (b) *Function Execution Module* works based on a serverless engine (e.g., Knative) to execute tasks and report the results back to the Object Control Module; (c) *Data Management Module* handles the object state; And (d)

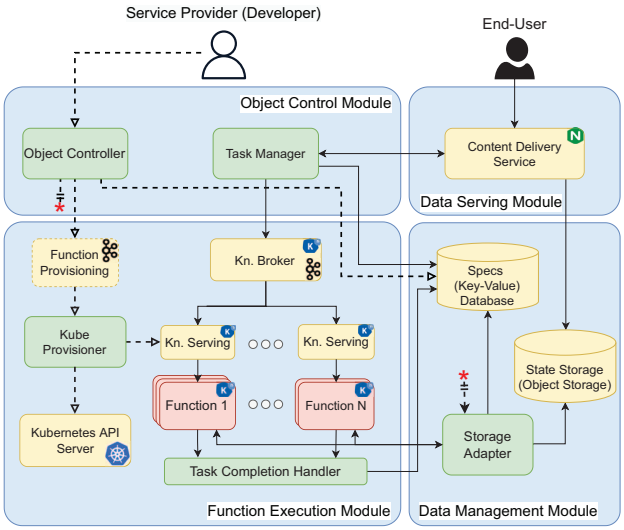


Fig. 3: A bird-eye view to the architecture of OaaS. **Dashed** lines show the workflow of actions taken by the developer, and **solid** lines show them for the end user. the **yellow** ones represent existing open-source tools; the **green** ones represent our implemented components; and the **red** ones are the containerized functions within OaaS.

Data Serving Module that is the end user's interface to OaaS. Details of these modules and their interactions are described in Figure 4 and the following subsections.

D. Object Control Module

1) Object Controller: Object Controller is a key component of the OaaS platform that: (a) interfaces with the developer via REST APIs to manage object abstraction as a class; (b) manages the deployment process of the class and function; (c) provides object instantiation to service providers or users

Upon defining a class by the developer, it is first *registered* by validating the specifications of its functions and state; and then persisting them into a shared key-value database (called *Specs Database* in Figure 3). Next, the class is *deployed* via introducing the containers of its functions to the Kubernetes orchestrator. To make the deployment process robust against transient failures of the underlying system, it is carried out asynchronously via a Kafka broker (*Function Provisioning* in Figure 3) that guarantees the deployment is handled by the next component (Kube Provisioner) in OaaS. Object Controller is also responsible for instantiating objects. For that, upon receiving the object specifications, Object Controller uses the Storage Adapter to allocate a presigned URL where the developer can upload the object state (e.g., video file).

2) Task Manager: Task Manager is the central component of the Object Control Module that is primarily responsible for handling the function invocations. Upon receiving an invocation that includes the object ID, function name, and input values, Task Manager augments it with other necessary information to execute the function, including the necessary details for accessing unstructured data. It spawns one (or more) task(s) and submits it (them) to the Function Execution Module, where Knative Broker routes the task(s) to the

corresponding container.

Enabling macro functions and dataflow abstraction within OaaS involves dealing with the concurrency and ordering of the function execution handled by the Task Manager. Upon receiving a macro function invocation, the Task Manager component generates the *invocation graph* as its internal state and uses it to coordinate the ordering of the invocations. For that purpose, once the task completion event (from the Task Completion Handler) is received, the Task Manager readily generates the next task based on the invocation graph. In the case of a task failure, the Task Manager propagates the failure status to the dependent tasks in the invocation graph.

Task Manager exposes the object access interface (OAI) to enable end-users transparently access the object's state and functions. OAI operates based on the web services and provides two modes of object access: (*i*) *Synchronous mode* that the Task Manager holds the HTTP connection with the user application until the output object state is ready. It is suitable for interactive function calls and retrieving the object state. For instance, let `video1` be a video object identifier, `transcode(var=int)` be one of its functions, and `src.mp4` be the video content held in the output object. Then, a synchronous function call to the object is in the form of: `video1:transcode(var=1024)/src.mp4`. (*ii*) *Asynchronous mode* that is suitable for non-interactive function calls (e.g., macro function invocations). In this case, the Task Manager does not hold the HTTP connection. Instead, responds immediately with the specifications of the prospective output object. The user application can use the associated ID to check the object status at a later time. An asynchronous function call to the object of the previous example is in the form of: `video1:transcode(var=1024)`.

To reduce the overhead in accessing the unstructured content of the output object, the Task Manager avoids unnecessary data movements via leveraging the HTTP redirect mechanism [35] to make the URL of the content provided by the Storage Adapter available to the Content Delivery Service. This way, the unstructured content bypasses Task Manager, and Content Delivery Service can fetch the content in one hop and provide it to the user application.

With all these responsibilities of the Task Manager, it can potentially become the bottleneck. To avoid that, we design the Task Manager to be scalable by making it stateless. Hence, its container can be easily scaled out to multiple instances. The problem in making the Task Manager stateless is the "internal state" that it has to be maintained to support macro functions. To overcome this problem, we configure Task Manager to persist its internal state in the *Specs* Database.

E. Function Execution Module

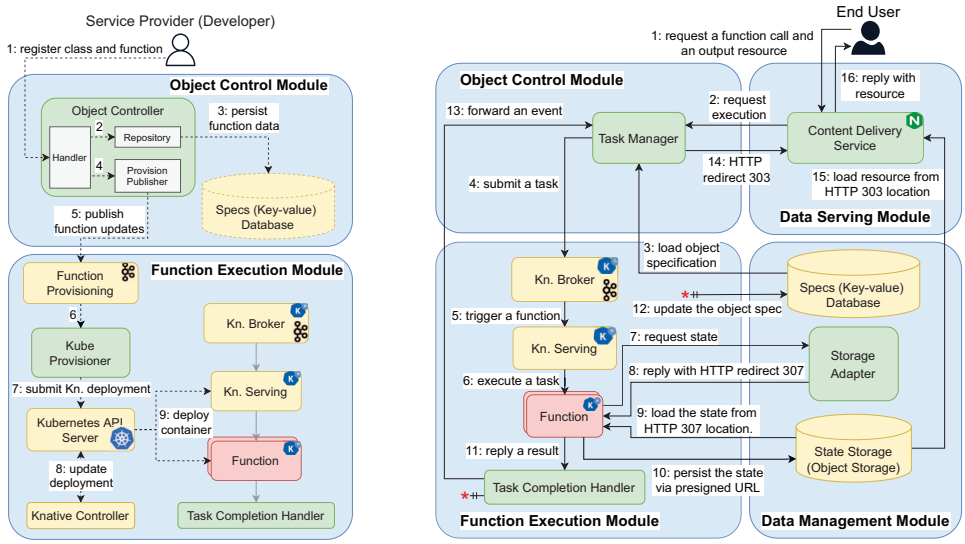
1) Handling Task Execution: For a given function call on an object, the Object Control Module is in charge of converting it to a *task* that is composed of detail of the function call and structured states of related objects. Then, the Function Execution Module receives the created task and takes care of its successful completion. Schematic view of the steps taken to handle a function call is noted in Figure 4b. This module utilizes Knative Broker, a component of Knative Eventing, consumes the "task event" generated by the Task Manager in the *Cloud Events* format [22], and routes the received task to the associated function container. Knative Serving is utilized to enable auto-scaling (and scale-to-zero) on the function container. It is noteworthy that the OaaS is modular, and other serverless engines can replace Knative without any major change to the system.

The Task Completion Handler component tracks the function execution and updates the execution status in the *Specs Database*. We note that each function container is an HTTP server to handle the messages in the Cloud Events format. Upon completing a task, the HTTP server issues a `2xx` status code, otherwise, the task is deemed failed.

2) Deploying Functions: Recall that, in addition to handling tasks, the Function Execution Module is in charge of deploying developer-defined functions. The key component of OaaS that is responsible for this is the Kube Provisioner. As it is expressed in Figure 4a, Kube Provisioner receives a function deployment request (that includes function specifications) from the Function Provision component via subscribing to Kafka Topic [28]. Upon receiving the request, Kube Provisioner translates the requested function specifications into the Kubernetes configuration format and forwards it to the Kubernetes API server, where the function container image is fetched from the container registry and is deployed. This process makes the function ready for invocation by Knative Broker.

F. Data Management Module

The Storage Adapter component is responsible for efficient and secure access to the object state. It also communicates with the Specs Database (see Figure 3) to retrieve the class specifications required to verify authorized accesses to the object state. Any component of OaaS that needs to access the state of an object has to do it through the Storage Adapter. We originally designed the Storage Adapter to work with S3-compatible object storage systems (e.g., Ceph [32], and MinIO [31]), however, the adapter can be extended to support other storage types too. To mitigate the overhead of retrieving the object state data, Storage Adapter avoids unnecessary data movements. That is, instead of relaying state data to the requester component—because S3 protocol is HTTP-based—the Storage Adapter can employ the *HTTP redirect* mechanism and only send the URL of the state data to the requester. For that purpose, the Storage Adapter digitally signs the URL of the state data with a secret key to generate the authorized presigned URL. As such, the presigned URL only grants access to the state data addressed by the URL. In this manner, the Storage Adapter preserves the object state security by preventing unauthorized access of a function to another object's state through learning the URL pattern. Accordingly, this mechanism decouples the object state storage from the function logic, such that in implementing a function, the developer does not need to know the storage details, such as the storage type, location, organization, and authentication.



(a) deploying a new function for a class

(b) invoking a function of an object

Fig. 4: The interaction flows between the OaaS components for two scenarios.

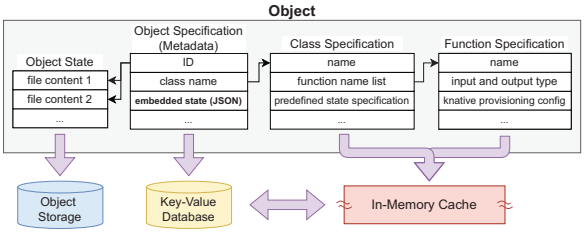


Fig. 5: Data modeling of objects in OaaS. The top part shows different data OaaS has to handle for an object. The bottom is the data tiering of OaaS based on the data size and access frequency.

G. Data Serving Module

The Content Delivery Service is to handle the object access requests of the end user. It is implemented using the Nginx server [20] that can load balance requests across multiple instances of Task Manager. Moreover, it includes a caching mechanism to increase the object access efficiency when multiple users request access to the same object. Recall that the synchronous-mode object access is replied to by Task Manager through HTTP redirection. Content Delivery Service explores the redirected location to retrieve the object state data from the storage. Then, Content Delivery Service updates its local cache and replies to the user with the object state data.

H. Object Data Modeling in OaaS

OaaS has to deal with four types of persisted data for each object. Accessing such data frequently, if not handled properly, can cause a slowdown of the platform. Hence, we develop the data modeling scheme to efficiently organize different types of data associated with each object such that the system slowdown is minimized. As shown in the top part of Figure 5, the following four types of data have to be maintained for each object: (a) *Object State*, which is the unstructured data the object represents; (b) *Object Specification (Metadata)* defines the object's characteristics, including the execution status and class name (which is linked to the class specification data). Objects whose state is in the structured form piggyback this metadata to store their structured state in the JSON format; (c) *Class Specification* is the developer-provided details to introduce the state and functions for the objects of a specific class; (d) *Function Specification* includes the function signature (i.e., the type of inputs and output) and its deployment configuration (e.g., the function container URL that is accessible to OaaS).

As the class and function specifications are common across objects, they are accessed more frequently than the (often large-size) unstructured object state. Accordingly, we employ the object storage (e.g., S3 [5]), which offers a high space-per-cost ratio, to persist the unstructured state. For other frequently-accessed data, which are generally smaller in size, we configure a fast and efficient key-value database (e.g., Infinispan [33]) for persistence. The class and function metadata are frequently-accessed but infrequently updated. Besides, they are small in size and quantity. These features make them suitable for in-memory caching. Hence, as depicted in Figure 5, we configure every component of OaaS that deals with the class and function metadata to locally cache them via an in-memory.

IV. Discussions

a) Fault Tolerance: Since OaaS allows running data transformation workloads, the first leading concern is the fault tolerance property to guarantee that the accepted function call will be executed or fail gracefully. The goal in this context is usually an Exactly-Once guarantee that the system will be run to the same result as if failure never happened. Since OaaS use Kafka as the message broker, it will guarantee that the received function call will never be lost by writing it to disk and replicating it across multiple broker nodes. Regardless, it can still be processed more than once, which normally can lead to data inconsistency. However, OaaS is designed to have an object as the immutable record, making the function invocation innately idempotent. This property would prevent data inconsistency even if the execution happens repeatedly.

In future work, we plan to extend the fault tolerance feature, such as the atomicity guarantee, across the workflow.

b) Security: Security is another primary concern when working with cloud service that is shared between multiple parties. We do not focus on security details in this paper, but there are the following aspects that can be done or have been done to harden the system. The first aspect is reducing attacking surface by limiting the necessary outbound traffics from the function container since it only requires access to Storage Adapter and object storage. Therefore, the network policy can be configured to block outbound traffic except for the Storage Adapter and object storage. The second aspect is avoiding reusing secret tokens. We employ the presigned URL mechanism for object storage to prevent the function container from accessing undesirable data. Thus, the security of the object storage in OaaS is more than FaaS where the same secret key is used for every request. To make the Storage Adapter secure, we can make Task Manager to generate a unique secret token for each task, and every request for Storage Adapter must be authenticated via the secret token.

c) Cold Start in Object Invocation: Not only the developer functions, but also the OaaS components can benefit from scale-to-zero to reduce the cost when there is no usage. However, this has the side-effect of more col starts. Since OaaS components are shared across functions, we can effectively keep it warm to eliminate the additional cold start impact. In such a case, the cold start performance is entirely derived from the underneath serverless execution engine.

V. Performance Evaluation

A. Experimental Setup

We deploy the OaaS platform on three machines of Chameleon Cloud [34], each with 2 sockets of 24-Core AMD EPYC7352 processors that collectively have 144 cores, 768 GB memory, and NVMe storage. The cluster includes three VMs with 16 vCPUs, 32 GB memory, and Kubernetes. We configured Rook [24] and Ceph [32] for persistence. Infinispan [33] is a distributed key-value database that we employed for the Specs Database. OaaS is implemented using Java.

Baselines. Apache Flink Stateful Function (StateFun) [7], OpenWhisk [21], and Knative [23] are configured as the baselines. Unlike OaaS and OpenWhisk, that focus on API calls and event handling, StateFun is an open-source stateful serverless focusing on stream processing. Because StateFun does not manage the function instances out of the box, we configure Knative to complement it. OpenWhisk is a FaaS platform that we use to represent the case where the function state management is performed explicitly by the developer.

We used Gatling [14] for load generation and implemented three scenarios to serve as the workload. Firstly, we developed a video transcoding function using FFmpeg [45] that is CPU-intensive and aligns with the motivation of this paper; Secondly, we developed a lightweight text concatenation function that concatenates the content of a text file with an input string. Thirdly, we developed a JSON update function that randomly puts the data into the JSON state data. The other workload characteristics are specific to each experiment and are explained in the respective sections. As StateFun does not support unstructured data as the state, we exclude it for the video transcoding and text concatenation functions.

B. Analyzing the Imposed Overhead of OaaS

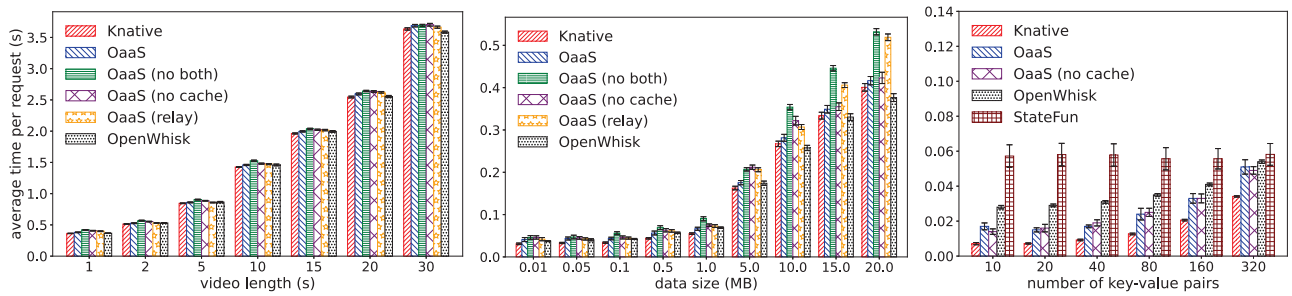
The abstractions provided by OaaS impose an overhead to the underlying system that we aim to measure in this experiment. The extra latency of a function call in OaaS is the metric that represents the overhead. We mainly study two sources of the overhead: (a) The *object state size* that highlights the overhead of OaaS in dealing with the stored data; and (b) The *concurrency of function calls* that highlights the overhead of the OaaS platform itself.

We examine three types of objects: (i) An object with a one-second-long video file (105 KB with resolution 1920×1080) as its state and a `transcoding` function, which exhibits a compute-intensive behavior; (ii) An object with a text file (10 KB) as its state and a function that `concatenates` the state with its input string (8 Bytes) argument. Because the processing time is only a fraction of the data loading time, we consider it as data-intensive; (iii) An object with structured (JSON) data as the state and a JSON `update` function that doubles the amount of persisted random key-value pairs.

The impact of changing the state size is shown in Figure 6. To generate objects with various state sizes, we increased the input video length from 1—30 seconds. To remove the impact of video content on the result, the longer videos were generated by repeating the 1-second video. Similarly, the text files are from 0.01—20 MB. In the JSON object, the key and value sizes are 10 and 40 bytes, respectively, and the number of key-value pairs varies from 10—320. To concentrate only on the overhead of data access and avoid other sources of overheads, we configure Gatling to assign only one task at a time and repeat this operation 100 times. To analyze the improvements offered by the URL redirection and data tiering (particularly metadata caching), we examine four versions of OaaS: the full version; without metadata caching (expressed as *OaaS (no cache)* in Figure 6); without URL redirection (expressed as *OaaS (relay)*); and without both URL redirection and metadata caching (expressed as *OaaS (no both)*).

In Figure 6, in general, the average task execution time increases for larger state sizes. We also observe that the caching impact on OaaS is insignificant because there is no function concurrency where caching can become effective. For both video and text (Figures 6a and 6b), OpenWhisk outperforms Knative and OaaS. For video, the gap is negligible because the time is dominated by the transcoding operation. For text, however, OpenWhisk directly interacts with the storage without any adapter layer in place. In Figure 6b, the gap between OaaS (relay) and OaaS widens for larger state sizes. For the 20 MB file, the redirection mechanism can reduce the execution time by 24%, and collectively with the caching, it can cause up to 27% improvement.

In the JSON update function (Figure 6c), the redirection mechanism is not used, hence, *OaaS (relay)* is excluded from



(a) Video transcoding function (b) Text concatenation function (c) JSON update function

Fig. 6: The average execution time of invocations with various state sizes on three types of workloads. Four versions of OaaS are examined.

the chart. We observe that OaaS imposes the least overhead across stateful solutions (StateFun and OpenWhisk). However, the gap between OaaS and Knative widens for the larger state sizes because OaaS has to read and write the state from/to the Specs Database, and both task and state have to travel through multiple components. This is why we chose to separate the unstructured state, which is generally bulky, from the object specification. We also see that, unlike other platforms, the execution time of StateFun does not change by increasing the state size. This is because StateFun stores the state on the local datastore without involving the external database.

The impact of concurrent function invocations on the OaaS overhead is shown in Figure 7. We increase the number of concurrent invocations of the same function and measure the average time to complete one task. For the transcoding function (Figure 7a), OaaS does not impose any significant overhead in comparison to Knative. However, in Figure 7b, the difference is noticeable (around 48 ms or 19% at 160 concurrencies) for Concatenation. The difference is because Concatenation is IO-intensive with short run-time (high-throughput) and high network bandwidth demand that is also needed by OaaS to store the object metadata. In OpenWhisk, however, each container with the Python runtime is used just to handle one function at a time, hence, it yields much higher execution times for all the functions.

For the structured data (JSON update in Figure 7c), the difference in overhead of OaaS and Knative (162 ms or 43%) is attributed to the time OaaS needs to persist the state and metadata. Note that the reported time for Knative on structured data only includes the function execution time (stateless part). In contrast, StateFun imposes a lower overhead than OaaS at the high concurrency because it uses the local datastore to reduce the cost of persisting state and uses Protobuf [17] to encode the data between the platform service and the function, which is more efficient than JSON (used by other and OaaS).

Takeaway: *The overhead analysis testifies that OaaS can operate with an insignificant latency overhead, specifically, for objects with unstructured state. Importantly, the redirection mechanism is decisive in mitigating the latency overhead for objects with large state sizes.*

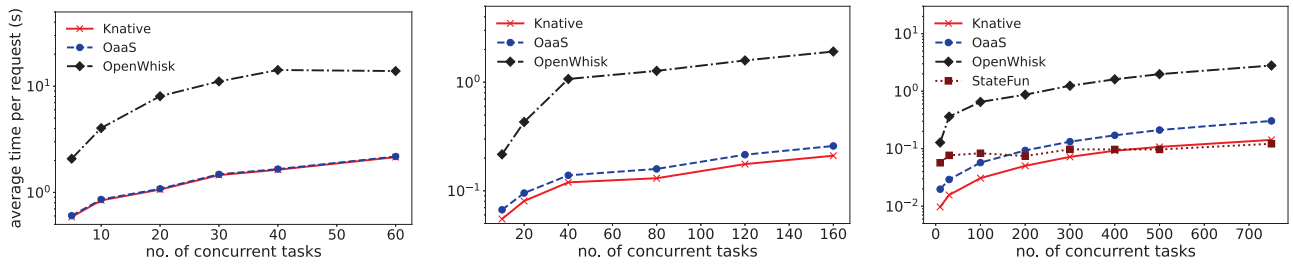
C. Scalability of the OaaS Platform

To study the scalability, we scale out OaaS from 3—12 VMs, each one with 16 vCPU cores (in total 48—192 vCPUs), and measure the speedup. We examine the JSON update function because it is supported by all the baselines, and its computing and I/O parts are balanced. We assume three VMs as the base with speedup=1, and the speedup of other configurations is calculated with respect to the base value. In each case, we measure the throughput (i.e., the average number of completed update operations per second). Then, the speedup value (Figure 8a) is calculated relative to the throughput of three VMs. We continuously increase the concurrency until the throughput stops growing, then choose the peak as the maximum throughput of a specific cluster size (see Figure 8b). In this figure, the Knative throughput is calculated by excluding the state persistence part, and it serves only as the theoretic benchmark by providing the ideal upper bound throughput. According to Figure 8a, all platforms have a similar speedup at 6 VMs. After that, StateFun offers the highest speedup and throughput in comparison to OaaS and OpenWhisk. The reason is that StateFun is built on top of Apache Flink, a mature stream processing platform. While we observe that Knative speedup slows down at 12 VMs, potentially due to limitations in its internal mechanics, OaaS continues to scale. According to Figure 8b, this is because both OaaS and StateFun are far to be bottlenecked by the limitations of Knative. For OpenWhisk, even though more VMs added, its autoscaling stops deploying new workers after reaching a certain number of containers. Thus, the speedup stops increasing after 6 VMs.

Takeaway: *The scalability analysis testifies that OaaS is as horizontally scalable as its underlying Knative framework.*

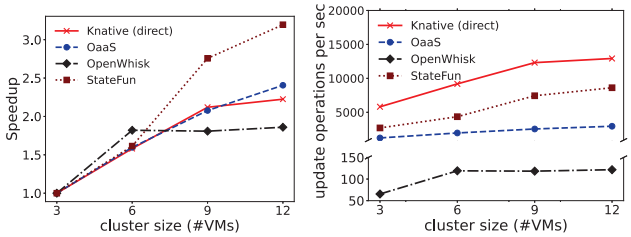
D. Case Study: Development Efficiency Using OaaS

In this part, we provide a real-world use case of object development using OaaS and its FaaS counterpart and then demonstrate how OaaS makes the development process of cloud-native serverless applications easier and faster. The use case is a video processing application that employs a machine learning model to perform face detection on video content. Figure 9 shows the workflow of functions needed: `Function1` to split the input video into multiple video segments that can



(a) Video transcoding function (b) Text concatenation function (c) JSON update function

Fig. 7: The average execution time of invocations with various concurrent intensities on three types of workloads.



(a) Speedup (b) Throughput

Fig. 8: Evaluating the scalability of OaaS against other baselines.

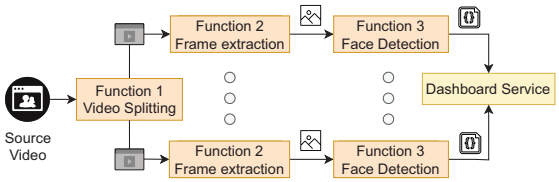


Fig. 9: Use case of developing a face detection workflow for a video.

be processed concurrently on multiple instances of `Function2` whose job is to extract the frames of each video segment; `Function3` is in charge of performing the face detection on the requested video frames and generating an object in the `JSON` format. These functions have to persist their output object so that the next function in the workflow can consume it.

FaaS implementation. The developer must implement the following steps: (i) Configuring cloud-based object storage and maintaining the credential access token for the functions to use. (ii) The functions' business logic has to be implemented and configured to react to the trigger events. (iii) Data management within the functions that itself involves three steps: (a) allocating the storage addresses to fetch or upload data; (b) authenticating access to the object storage via the access token; and (c) implementing the fetch and upload operations on the allocated addresses. Upon implementing these functions, the developer has to connect them as a workflow via a function orchestrator service (e.g., AWS Step Functions). Finally, the dashboard service invokes the workflow upon receiving a request from the end user and collects the results.

OaaS implementation. The developer defines three classes, namely `Video`, `Image`, and `Detection_Result` in form of the three following classes: (a) `Video` class with `split_video()` and `extract_frame()` functions; and a macro function, `df_detect_face(detect_interval)`, that includes the whole workflow of function calls, with the requested face detection period as its input, and a `Detection_Result` object, as the output. (b) `Image` class with the `detect_face()` function. (c) `Detection_Result` class that does not require any function. The Dashboard Service calls the `df_detect_face(detect_interval)` macro function directly using the object access interface, and receives the `Detection_Result` object as the output. We note that in developing the class functions, the developer does not need to involve in the data locating and authentication steps.

Takeaway: *The OaaS paradigm aggregates the state storage and the function workflow in its platform and enables cloud-native dataflow programming. As such, the developers are relieved from the burden of state management, learning the internal mechanics of the functions and pipelining them.*

VI. Conclusions

In this research, we presented the OaaS paradigm that incorporates state management into cloud functions and offers cloud object abstraction. We developed a prototype of the OaaS platform that relieves the developer from the burden of state management, hence, improving the cloud-native applications development efficiency. To make the OaaS scalable, we make the object state immutable. This approach preserves the object state consistency without requiring any synchronization mechanism that limits the scalability. Moreover, OaaS enables cloud-based dataflow programming where a workflow of functions can be transparently defined without concurrency and synchronization concerns. We evaluated the OaaS in terms of ease of use, imposed overhead, and scalability. The evaluation results demonstrate that OaaS streamlines cloud programming and is ideal for the use cases that require persisting the state or defining a workflow. OaaS offers scalability with negligible overhead, particularly, for compute-intensive tasks. In the future, we plan to develop an object-based platform via replacing the underlying software platforms with our customized solutions to further improve data locality, invocation efficiency, and scheduling optimizations.

References

[1] Amazon. Amazon API Gateway – Amazon Web Services. https://aws.amazon.com/api-gateway/. Online; Accessed on 10 Dec. 2022.

[2] Amazon. AWS IAM – Identity and Access Management – Amazon Web Services. https://aws.amazon.com/iam/. Online; Accessed on 10 Dec. 2022.

[3] Amazon. AWS Lambda – Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/. Online; Accessed on 10 Dec. 2022.

[4] Amazon. AWS Step Functions – Serverless Microservice Orchestration. https://aws.amazon.com/step-functions. Accessed on 10 Dec. 2022.

[5] Amazon. Cloud Object Storage – Amazon S3 – Amazon Web Services. https://aws.amazon.com/s3/. Online; Accessed on 10 Dec. 2022.

[6] Amazon. Fast NoSQL Key-Value Database – Amazon DynamoDB – Amazon Web Services. https://aws.amazon.com/dynamodb/. Online; Accessed on 10 Dec. 2022.

[7] Apache. Apache Flink Stateful Functions. https://nightlies.apache.org/flink/flink-statefun-docs-stable. Online; Accessed on 10 Dec. 2022.

[8] S. Bangera. *DevOps for Serverless Applications: Design, deploy, and monitor your serverless applications using DevOps practices*. Packt Publishing, 2018.

[9] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54. Association for Computing Machinery, 2019.

[10] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Serverless workflows with durable functions and netherite. *arXiv preprint:2103.00033*, 2021.

[11] Sylvain Cherrier and Yacine M Ghamri-Doudane. The "object-as-a-service" paradigm. In *2014 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–7. IEEE, 2014.

[12] Google Cloud. Cloud Functions – Google Cloud. https://cloud.google.com/functions/. Online; Accessed on 10 Dec. 2022.

[13] Cloud Native Foundation. Kubernetes. https://kubernetes.io/. Online; Accessed on 30 Jul. 2022.

[14] Gatling Corp. Gatling - Professional Load Testing Tool. https://gatling.io/. Online; Accessed on 30 Jul. 2022.

[15] Chavit Denninnart and Mohsen Amini Salehi. SMSE: A Serverless Platform for Multimedia Cloud Systems. *arXiv preprint:220.0194*, 2022.

[16] Chavit Denninnart and Mohsen Amini Salehi. Harnessing the potential of function-reuse in multimedia cloud systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):617–629, 2021.

[17] Google Developers. Protocol Buffers. https://developers.google.com/protocol-buffers. Online; Accessed on 1 Aug. 2022.

[18] eigr. eigr.io. https://eigr.io. Online; Accessed on 10 dec. 2022.

[19] Alex Ellis. OpenFaaS – Serverless Functions Made Simple. https://www.openfaas.com/, Online; Accessed on 24 Jul. 2022.

[20] Martin Bjerretoft Fjordvald and Clement Nedelcu. *Nginx HTTP Server: Harness the power of Nginx to make the most of your infrastructure and serve pages faster than ever before*. Packt Publishing Ltd, 2018.

[21] Apache Software Foundation. Apache OpenWhisk is a serverless, open source cloud platform. https://openwhisk.apache.org/, Online; Accessed on 24 Jul. 2022.

[22] Cloud Native Foundation. CloudEvents. https://cloudevents.io/. Accessed on 10 Dec. 2022.

[23] Cloud Native Foundation. Knative. https://knative.dev/. Online; Accessed on 10 Dec. 2022.

[24] Cloud Native Foundation. Rook. https://rook.io. Online; Accessed on 18 Jul. 2022.

[25] Davood Ghatrehsamani, Chavit Denninnart, Josef Bacik, and Mohsen Amini Salehi. The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, 2020.

[26] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[27] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(1):1–29, 2021.

[28] Bhole Rahul Hiraman et al. A study of apache kafka in big data stream processing. In *1st International Conference on Information, Communication, Engineering and Technology (ICICET)*, pages 1–3, 2018.

[29] Mohamed K Hussein, Mohamed H Mousa, and Mohamed A Alqarni. A placement architecture for a container as a service (caas) in a cloud environment. *Journal of Cloud Computing*, 8(1):1–15, 2019.

[30] Lightbend Inc. High performance microservices and APIs | Kalix.io. https://www.kalix.io. Online; Accessed on 10 Dec. 2022.

[31] MinIO Inc. MinIO | High Performance, Kubernetes Native Object Storage. https://min.io/. Online; Accessed on 10 Dec. 2022.

[32] Red Hat Inc. Ceph. https://ceph.io/. Online; Accessed on 10 Dec. 2022.

[33] Red Hat Inc. Infinispan. https://infinispan.org/. Online; Accessed on 10 Dec. 2022.

[34] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC '20. USENIX Association, July 2020.

[35] Martin Koop, Erik Tews, and Stefan Katzenbeisser. In-depth evaluation of redirect tracking and link usage. *Proceedings on Privacy Enhancing Technologies*, 2020(4):394–413, 2020.

[36] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, et al. Apiary: A dbms-backed transactional function-as-a-service framework. *arXiv preprint arXiv:2208.13068*, 2022.

[37] Xiangbo Li, Mohsen Amini Salehi, Yamini Joshi, Mahmoud K Darwich, Brad Landreneau, and Magdy Bayoumi. Performance analysis and modeling of video transcoding using heterogeneous cloud services. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):910–922, 2018.

[38] Manisha Luthra, Sebastian Hennig, Kamran Razavi, Lin Wang, and Boris Koldehofe. Operator as a service: Stateful serverless complex event processing. In *8th IEEE International Conference on Big Data*, pages 1964–1973, 2020.

[39] Microsoft. Azure Functions Serverless Compute. https://azure.microsoft.com/en-us/services/functions/. Online; Accessed on 10 Dec. 2022.

[40] Microsoft. Durable entities - Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities. Online; Accessed on 10 Dec. 2022.

[41] Davood G. Samani and Mohsen Amini Salehi. Exploring the impact of virtualization on the usability of the deep learning applications. In *Proceedings of the 22th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, CCGrid '22, May 2022.

[42] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*, USENIX ATC '20, pages 419–433, 2020.

[43] Tiago Boldt Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, volume 130, 2012.

[44] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 2020.

[45] Hao Zeng, Zhiyong Zhang, and Lulin Shi. Research and implementation of video codec based on ffmpeg. In *2nd international conference on network and information systems for computers (ICNISC)*, pages 184–188, 2016.

[46] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent 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.