Object as a Service: Simplifying Cloud-Native Development through Serverless Object Abstraction

Pawissanutt Lertpongrujikorn, Mohsen Amini Salehi
High Performance Cloud Computing (HPCC) Lab, University of North Texas

***Abstract*—The function-as-a-service (FaaS) paradigm is envisioned as the next generation of cloud computing systems that mitigate the burden for cloud-native application developers by abstracting them from cloud resource management. However, it 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 storage services. To further streamline cloud-native application development, in this work, we propose a new paradigm, known as Object as a Service (OaaS) that encapsulates application data and functions into the cloud object abstraction. OaaS relieves developers from resource and data management burden while offering built-in optimization features. Inspired by OOP, OaaS incorporates access modifiers and inheritance into the serverless paradigm that: (a) prevents developers from compromising the system via accidentally accessing underlying data; and (b) enables software reuse in cloud-native application development. Furthermore, OaaS natively supports dataflow semantics. It enables developers to define function workflows while transparently handling data navigation, synchronization, and parallelism issues. To establish the OaaS paradigm, we develop a platform named *Oparaca* that offers state abstraction for structured and unstructured data with consistency and fault-tolerant guarantees. We evaluated Oparaca under real-world settings against state-of-the-art platforms with respect to the imposed overhead, scalability, and ease of use. The results demonstrate that the object abstraction provided by OaaS can streamline flexible and scalable cloud-native application development with an insignificant overhead on the underlying serverless system.**

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

I. Introduction

A. FaaS and Its Shortcomings

Function-as-a-Service (FaaS) is envisioned as the next generation of cloud computing (Cloud 2.0) [18], mitigating burdens for programmers and architects. Major cloud providers offer FaaS services (e.g., AWS Lambda, Google Cloud Function, Azure Function), while open-source platforms (e.g., OpenFaaS, Knative) enable on-premise deployments. FaaS provides function abstraction for developing business logic triggered by predefined events, with serverless platforms handling resource management transparently and scalably. By implementing scale-to-zero and pay-as-you-go charging, FaaS reduces costs and aligns with modern development paradigms like CI/CD and DevOps [8].

Since FaaS centers on stateless *functions*, it does not address *data* management. However, most applications require state data, and FaaS's limited inter-function communication necessitates remote storage for intermediate data. Consequently, developers must manage data through separate cloud services (e.g., AWS S3 [6]). For instance, video streaming applications [13] require developers to maintain video files, metadata, and access control alongside function development.

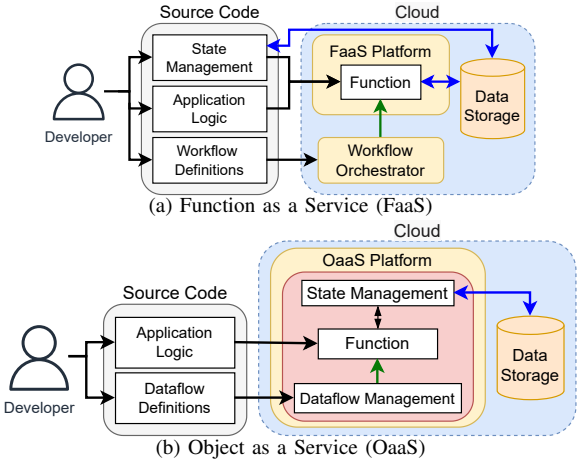


Fig. 1: A bird-eye view of FaaS vs. OaaS.

Beyond data management, FaaS lacks built-in sematic for access control to functions' internal mechanics. Unrestricted access causes side effects like unintended function invocation and data corruption. Developers must configure external services (e.g., AWS IAM [4]) for access control.

Finally, FaaS lacks native workflow support. Developers must chain functions through events, which becomes cumbersome for large workflows. While orchestrator services (e.g., AWS Step Function [5], Azure Durable Function [10]) help, FaaS's inherent limitations–(a) restricted inter-function communication and (b) absent state management (Figure 1)–force developers to use external storage for intermediate data and manually navigate data throughout workflows [29]. Thus, while FaaS abstracts resource management (e.g., auto-scaling), it leaves data, access control, and workflow management to developers.

B. Proposed Paradigm

To address these FaaS limitations, we propose a paradigm that relieves developers from resource, data, and workflow management burdens. *We borrow the notion of "object" from object-oriented programming (OOP) and develop a new abstraction level within the serverless cloud, called* ***Object as a Service (OaaS)***, a new serverless cloud abstraction.

As Figure 1 shows, OaaS separates state management from developer code, incorporating it into the serverless platform.

OaaS integrates workflow orchestration through *dataflow abstraction* with built-in cross-function data navigation. Unifying application data and workflow in object abstraction enables built-in optimizations–data locality, reliability, caching, Quality-of-Service targets [26], software reusability [14], and access control. OaaS lets developers declare object behavior and properties as `class` and `function`. For complex workflows orchestrating multiple objects and functions, OaaS provides native dataflow interfaces that abstract data navigation and synchronization details.

An exemplar use case that can take advantage of OaaS is a cloud-based video streaming systems (e.g., [13], [28]), requiring rapid implementation of services like harmful content blurring, face detection, and transcoding pipelines. FaaS implementations require managing state data (videos) alongside business logic. In this scenario, the OaaS paradigm can mitigate the developer's burden by offering encapsulation. The videos are defined as persistent objects bound to a set of functions that can be invoked by the viewer's application and potentially change the object's (video's) state. For example, to blur gore segments in the video object `v1`, the developer invokes `v1.detect_gore()` to obtain the output object `g1` that contains a list of time stamps of gore segments—next, blurring the video via invoking `v1.blurs(g1)`.

C. Research and Contributions

The *first* challenge is to establish OaaS by providing the "object" abstraction while ensuring the platform's modularity and extensibility. The primary requirement is to offer an interface for developers to declare object behavior and state as classes, functions, and dataflow. With the addition of access modifiers, OaaS can encapsulate internal mechanics, allowing objects to reference other objects and form dataflow functions. This high-level abstraction enforces strong encapsulation, preventing invalid access and thereby unraveling complexity. To realize the OaaS paradigm, we develop **Oparaca** (**O**bject **Para**digm on Serverless **C**loud **A**bstraction), that is driven by flexibility and modularity. It must accommodate multiple use cases, remain extensible, and integrate seamlessly with various execution (FaaS) and storage modules. By leveraging pure functions, Oparaca avoids tight coupling with execution modules and offloads tasks without side effects. A data tiering scheme further abstracts storage, ensuring that changes in storage types do not necessitate modifications in the function code.

The *second* challenge is to reduce the overhead of data movement between functions and the platform's components. For that purpose, the data tiering scheme within the Oparaca platform diminishes the latency of accessing the object by employing a distributed in-memory hash table [35]. However, for unstructured data, also known as BLOB (e.g., multimedia), that cannot fit in memory, Oparaca persisted them in a separate object storage. To further mitigate the overhead of accessing objects, Oparaca is equipped with a presigned URL with a redirection mechanism that reduces unnecessary data movements within the platform instead of relaying (i.e., transferring) the object state.

The *third* challenge is to ensure fault tolerance and data consistency. Concurrent requests to the same state can lead to data inconsistencies due to race conditions, and system failures may result in mismatched states across multiple data stores. Oparaca introduces a *fail-safe state transition* mechanism that maintains data consistency and fault tolerance to tackle these issues. Additionally, Oparaca combines lightweight optimistic locking [25] and *localized locking* to prevent concurrent modifications to the same object. Another aspect of fault tolerance is the recovery mechanism from failure via retrying. However, such a remedy can potentially lead to another problem—repeating the execution more than once and falling into an undesirable state. To cope with this challenge, we develop a mechanism within Oparaca that establishes consistent state transitioning by guaranteeing "exactly-once" execution for the function calls.

In summary, the key contributions of this research are as follows:

1) Proposing **OaaS**, a paradigm simplifying cloud-native development by unifying application data, business logic, and workflows into object abstraction. OaaS applies OOP principles to serverless computing through cloud class inheritance, enhancing software reusability across diverse functions and runtimes.
2) Developing **Oparaca**[1], an OaaS prototype supporting structured and unstructured data with ensuring data consistency via fail-safe state transitions and scalable localized locking that avoids cluster-wide coordination overhead.
3) Devising a declarative dataflow abstraction that enables developers to define workflows based on data dependencies rather than task dependencies. Oparaca implements this through efficient techniques, such as presigned URLs with redirection and distributed in-memory caching, to minimizing performance overhead and improve scalability.
4) Analyzing Oparaca's performance from the scalability, overhead, and ease-of-use perspectives.

II. Background and Prior Studies

The idea of stateful serverless is explored in several research works. These approaches primarily aim to address the stateless nature of the FaaS model, where the burden of managing application data, access control, and function workflows is often shifted to the developer through separate cloud services. As noted in Figure 2, these works can be categorized into actor model, datastore abstraction, and pure function approaches depending on how the platform manages data and allows functions to access it.

Actor Model. In the actor model, the platform fetches the state from persistent storage and places (i.e., caches) it inside a worker instance, then dispatches the request to where the state resides to achieve data locality. This approach, however, can make maintainability difficult for bulky unstructured data. The deployment granularity is an actor that contains multiple functions sharing the same state. The foundational platform in this space is Orleans [9], which introduces "virtual actors". Its influence is evident in modern platforms such as Azure

[1]The source code and example: https://github.com/hpcclab/OaaS

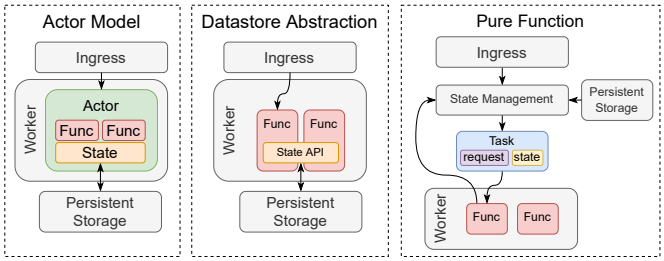


Fig. 2: The illustrated comparison of three different models of stateful serverless.

Entity Functions [16]. While effective for certain use cases, this model can tightly couple state and compute within a single language and environment. In contrast, Oparaca manages the object abstraction at the platform level, allowing functions implemented in different languages or runtimes to operate on an object, thereby offering greater flexibility.

Datastore Abstraction. The datastore abstraction is a hybrid approach where the platform provides an API for the function to access data on demand. Like pure functions, it relaxes the need for state and function co-location, but can utilize caching to preserve data locality. Several systems utilize this pattern. Cloudburst [34] uses a shared distributed key-value database, while Boki [22] provides API access to a distributed logging system. This log-based approach for consistency differs from Oparaca's mechanisms, which use a combination of fail-safe state transitions for unstructured data and localized locking to handle race conditions. Beldi [39], on the other hand, provides the database and transaction API to the state. While providing direct transactional APIs offers strong guarantees, Oparaca abstracts these concerns from developers through its fail-safe versioning scheme and guarantees of exactly-once execution, aiming for a higher-level resilience model. FAASM [32] optimizes the function-state interaction by using WebAssembly [17]. Although this enables multiple functions to share memory and achieve data locality, it requires compiling code to WebAssembly, which can limit library compatibility. Oparaca's hybrid model avoids this limitation by combining the pure function and datastore abstraction approaches, offering broader language support.

Pure Function. In the pure function approach, the state is placed on another system and transferred to the worker instance upon invocation, appearing as part of the function's input arguments. This disaggregates state management and computation for system design simplicity, but can compromise data locality. Apache Flink Stateful Function (StateFun) [15] is a solution based on this approach. Oparaca combines the pure function approach for structured data with a lazy-fetch mechanism for unstructured data, practically employing both the pure function and datastore abstraction approaches.

Process Abstractions. While OaaS simplifies development by *increasing* abstraction, a competing trend in recent work involves stepping back from abstraction to grant developers more control for performance. This trend of using OS-inspired abstractions is evident in various contexts. For serverless platforms, Process-as-a-Service (PaaS) [11] introduces a *cloud process* to optimize execution, while for traditional always-on services, Nu [31] utilizes migratable *logical processes*.

Serverless Workflow and Dataflow Beyond state management for individual functions, composing them into complex applications is another significant challenge. Commercial orchestrators, such as AWS Step Functions and Azure Durable Functions, can mitigate the burden of chaining events; however, developers are often still required to navigate data between steps manually. The research community has proposed more integrated solutions. For instance, DataFlower [29] introduces a dataflow paradigm specifically for serverless workflow orchestration to optimize data movement between functions. Netherite [10] focuses on the efficient execution of these workflows through techniques like partitioned state management and collocated execution. While these systems significantly advance workflow performance and orchestration, OaaS approaches the problem from a different angle by integrating workflow directly into its core object abstraction.

Oparaca's dataflow functions allow developers to declaratively define a workflow as a DAG based on data dependencies rather than task dependencies. The platform transparently manages parallelism, data navigation, and state consistency, abstracting these complexities from the developer.

III. Object as a Service (OaaS) Paradigm

A. Conceptual Modeling of OaaS

To realize OaaS, *first*, we need to establish the notion of *cloud object* as an entity that possesses a *state* (i.e., data) and is associated with one or more *functions*. We empower objects to support both structured (e.g., JSON records) and unstructured (e.g., video) forms of state. Upon calling an object's function, OaaS creates a task that can safely take action on the state.

Second, OaaS provides the *class* semantic as a framework to develop objects. Inspired by OOP, the developer has to define a set of functions and states within the class. Then, an arbitrary number of objects—that is bound to the functions and states declared in that class—can be instantiated. To improve cloud software reusability and maintainability, we enable class *inheritance* for cloud functions and states from other classes, plus the ability to *override* any derived function.

Third, OaaS offers built-in *access control* to provide the ability to declare the "scope of accessibility" for a state or function. Importantly, when defining a set of classes, the developer can declare it within a single package that includes the access modifier to prevent unauthorized access from other packages. This is particularly useful when cloud application developers utilize imported third-party packages.

Fourth, OaaS enables higher-level abstractions by allowing cloud objects to be nested, where a high-level object references lower-level objects. Functions can use these references to fetch inputs or invoke dataflow functions (called *macro functions*) that chain operations across lower-level objects. Unlike traditional FaaS workflows [5], macro functions determine execution flow based on dataflow rather than task (i.e., function call) dependency. Developers only define the data flow, while OaaS manages parallelism, data navigation, and state consistency transparently.

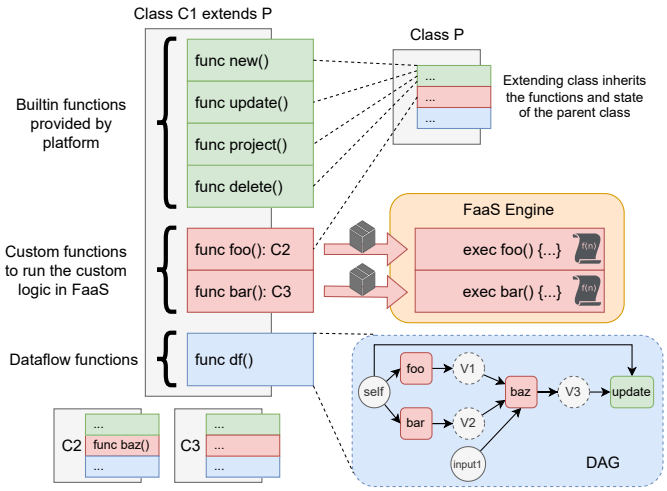


Fig. 3: Different types of functions supported by OaaS.

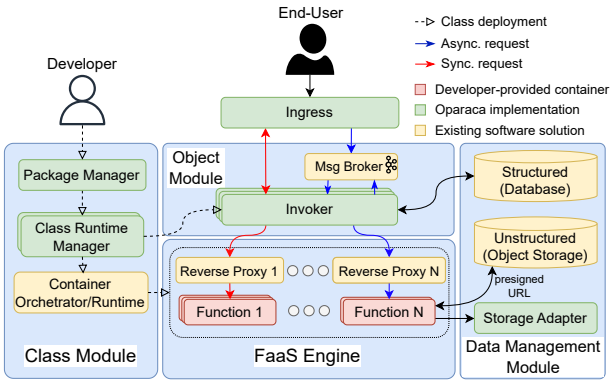


Fig. 4: A bird-eye view of the Oparaca architecture. Dashed lines show actions of the developer defining classes and objects, and solid lines show actions using objects and invoking functions.

B. Developing Classes in OaaS

In OaaS, developers define one or more classes within a package using configuration languages like YAML or JSON. The package definition contains the class section and the function section. The functions section defines the configuration and deployment details of each function. The class section defines the object's structure, which includes the state and function it links to.

As shown in Figure 3, OaaS supports three function types. First, *built-in* functions that are provided by the platform. These functions could be the standard functions such as `CRUD` (create, read, update, and delete), which are the common data manipulation operations. The platform manages the execution of these functions without intervention from the developer. Second, *custom* (a.k.a. *task*) functions that are developed by developers (OaaS users) to provide their business logic. To handle the invocation of these functions, OaaS employs existing FaaS engines in its underlying layers to exploit their auto-scaling and scale-to-zero capabilities. Third, *dataflow* (macro) functions are defined as a DAG representing the chain of invocations to objects.

As an example of package definition, Listing 1 represents a declaration example for a package that includes one class called `video` that has a state named `mp4` (Line 6), *built-in* function named `new` (Line 9), and *custom* function named `transcode` (Line 1). The state `mp4` refers to video data that is unstructured data. The class has a public *custom* function called `transcode`. The definitions of the *custom* function are declared in Lines 15—17. The `type` of a function (Line 16) can be a `task` (or a `macro`, as noted earlier). This function creates another object instance of type `video` as an output. Line 17 declares the container image URI for executing function code.

Listing 1: An example simplified script that declares `multimedia` package with a `video` class, and a `transcode` function for it in the `YAML` format.

```
1  name: multimedia
2  classes:
3    - name: video
4      stateSpec:
5        keySpecs:
6          - name: mp4
7            access: PUBLIC
8      functions:
9        - function: new
10         access: PUBLIC
11       - function: transcode
12         access: PUBLIC
13         outputCls: .video
14 functions:
15   - name: transcode
16     type: TASK
17     image: transcode-py:latest
18     ...
```

IV. Oparaca: A Platform for the OaaS Paradigm

A. Design Goals

The Oparaca platform is designed with its foundational goal of providing object abstraction with two additional design goals: *backward compatibility* and *extensibility*. *first*, while OaaS simplifies cloud-native application development, it is not always a replacement for FaaS; thus, Oparaca supports stateless FaaS and direct data access to storage systems. *Second*, for extensibility, Oparaca decouples the control plane from the execution plane, allowing the execution plane to operate independently via standardized APIs. This platform-agnostic design accommodates various execution planes optimized for specific use cases, such as latency-constrained function calls [33] or access to hardware accelerators [36].

B. Overview of the Oparaca Architecture

The Oparaca platform is designed based on multiple self-contained microservices that communicate within a serverless system. Figure 4 provides a birds-eye view of the Oparaca architecture that is composed of five modules:

- **Class Module** serves as the interface for developers to create and manage classes and their functions.

- **Object Module** serves as the cornerstone of Oparaca that has two main objectives: (a) providing the "object access interface" for the user application to access an object(s); and (b) offering the object abstraction while transparently handling function invocation and state manipulation.
- **FaaS Engine** is the underlying execution engine of Oparaca, which can be any existing FaaS system (e.g., Knative).
- **Data Management Module** is to manage object data persistence via employing database (e.g., document database) and object storage (e.g., S3-compatible storage). To bind these storages to the functions, the Invoker abstracts data access for structured data, while the Storage Adapter is employed to handle access to unstructured data in the object storage
- **Ingress Module** whose purpose is to provide a single endpoint for the user application.

Details of these modules, their interactions, and how they fulfill the consistency and fault-tolerance objectives (described in Section I) are elaborated in the following subsections.

C. Class Module

To define a new class and its functions in Oparaca, the developer defines them as a package definition and registers it to the *Package Manager*, shown in Figure 4. Upon successful package validation by the Package Manager, the *Class Runtime Manager* (termed *CRM* for brevity) performs the class registration process that includes two operations:
(**a**) Informing the Object Module about the new/updated class. Upon receiving a class registration, the Object Module creates a handler instance to be prepared for handling object invocation. We elaborate on this process in Section IV-D.
(**b**) Registering the custom functions of the new class in the FaaS engine for future invocation. Recall that we aim to make Oparaca agnostic from the underlying FaaS engine. We design Oparaca to host a dedicated CRM for each FaaS engine. Accordingly, a new FaaS engine can be integrated into Oparaca by simply plugging its dedicated CRM into the system. When a function registration event occurs, the corresponding CRM processes this event by translating the function configuration into the specific format for that engine (e.g., Knative) and forwards it. Consequently, the underlying FaaS Engine creates the actual function runtime to be invoked by the Object Module.

D. Object Module and FaaS Engine

Recall that OaaS needs to support three types of functions: built-in, custom, and dataflow. Unlike built-in functions and dataflow functions that can be executed without the direct need of the FaaS engine, custom functions need to execute the developer-provided code on the FaaS engine. Thus, Oparaca requires a mechanism to utilize the FaaS engine to execute the custom function code while allowing it to access the object state transparently and with the minimum data transfer overhead. Needless to say, this mechanism also maintains the separation between the Object Module and the FaaS engine.

To fulfill the above expectations, we design the object invocation mechanism in the Object Module by distinguishing between structured and unstructured states and managing it

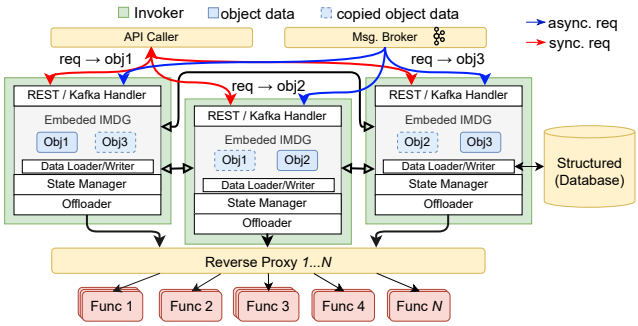


Fig. 5: The cluster of Invokers replicates and distributes object data across the cluster via IMDG with consistent hashing. The invoker offloads the invocation to a corresponding FaaS function.

so that the data access overhead is minimized. We develop a hybrid approach that leverages the "pure function" technique for structured data access and the "datastore abstraction" technique for unstructured data access. The rationale of this design choice is that the unstructured state (i.e., BLOB) is usually large and expensive to transfer; hence, to maintain efficiency, the FaaS engine should retrieve the state directly from the object storage (e.g., S3) in a lazy, on-demand manner. This differs from the structured state, for which we include the state as an input argument to maintain a clear separation between the Object Module and the FaaS engine and let the FaaS engine maintain its statelessness.

In the Oparaca architecture (Figure 4), he mechanism for handling invocation and state management is managed by the *Invoker* component. In particular, to offload the object invocation to the FaaS engine, Invoker bundles the request and the related structured object data as a "*task*", as described in the next part, and passes it to the associated FaaS engine for execution.

1) Task Generation in the Invoker: Upon receiving a function call, the Invoker bundles the invocation request and associated object data into the task and offloads it to be executed on the FaaS engine. To further reduce the data transfer overhead of providing the object abstraction in the task generation process, we design Invokers to maintain the object data (i.e., state and metadata) in a distributed hash table [19], thereby reducing the cost of data transfer in a scalable manner. As shown in Figure 5, we equip each Invoker instance with an embedded in-memory data grid (IMDG) [38]. IMDG partitions the entire data space into multiple segments and distributes them across Invoker instances. The Invoker with IMDG determines the segment for a given object by consistent hashing of the object ID and assigns the object data to the selected segment. Similarly, to retrieve the object data, IMDG determines the owner of the data and then fetches it from the owner of the segment in one hop.

2) Unstructured Data Accessing: To minimize the overhead of accessing unstructured data, Oparaca allows function code to access the unstructured data on-demand and directly through a *presigned URL* and *redirection* mechanism. The presigned URL is the specific HTTP URL that includes the digital signature in query parameters to grant permission for anyone with this URL to access the specific data without the secret token. When a function needs to access the unstructured

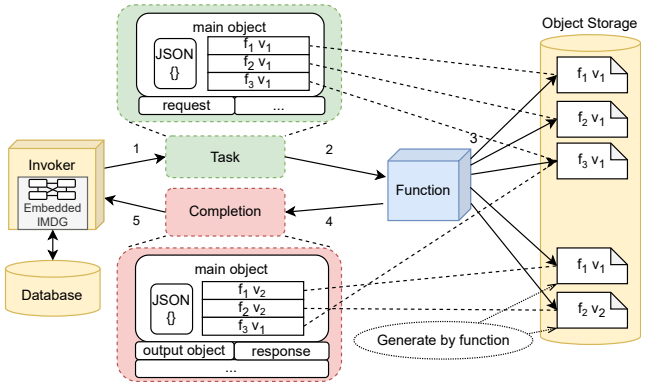


Fig. 6: The process of offloading invocation task into the function runtime. Invoker bundles the request input and object state into a task and offloads it to the function to be executed. With *fail-safe state transition*, when the function needs to update the file in object storage, it creates a new file and updates the corresponding version ID via the returning completion message.

data, it sends an HTTP request to the storage adapter to receive the redirection response that points to the presigned URL of specific state data. Then, the function code can fetch the content directly from object storage via the given presigned URL. In addition to minimizing the overhead, using the presigned URL is important in protecting the function container from unauthorized access to other objects' data by analyzing their URL patterns.

3) Task Completion: After the FaaS engine completes the task, it sends the task completion data to the Invoker to update the state. If the function reports a failed task, the state remains unchanged. Otherwise, the Invoker updates the object data in IMDG and then writes it to the persistent database immediately or asynchronously. If an invocation involves both structured and unstructured states, we use pure and datastore techniques together, which can potentially lead to "state inconsistency challenges". We address this challenge in section IV-F.

4) Synchronous and Asynchronous Invocation: As mentioned in Section IV-A and shown in Figure 5, we designed Oparaca to offer synchronous and asynchronous function invocations. In synchronous mode, the function is executed immediately upon invocation and returns the result to the caller. Meanwhile, in asynchronous mode, the invocation ID is provided to the caller as a reference so they can check the invocation result later. The request is placed into the message broker to be reliably processed at a later time. To accommodate both modes, the Invoker utilizes the handler instance to accept the invocation request for either the REST API (synchronous) or the message broker (asynchronous). Subsequently, the handler instance forwards the request to be processed in the same way by the other part of the Invoker.

E. Ingress Module

To provide the end user with a single access point, we position the Ingress Module in front of the cluster of Invokers. Additionally, to minimize data movement, the Ingress Module is designed to be aware of the object data distribution through consistent hashing of DHT. This allows the Ingress Module to correctly forward the object invocation request to the Invoker that owns the primary object data. As a result, the designated Invoker is able to access the data in its memory.

F. Resilience Measures of Oparaca

Oparaca is prone to the data inconsistency problem that stems from both *failure* and *race* conditions. In this section, we describe the internal mechanisms of Oparaca designed to make it resilient against these conditions.

1) Resilience against failure: Data inconsistency from failure can happen if the system stops while performing multiple update operations, causing some of the update operations to be incompletely executed. The pure function model, used for structured data in Oparaca, is inherently immune to this problem because a function returns the modified state to the platform only when its execution is complete. Nonetheless, the datastore abstraction used for the unstructured data in Oparaca is still prone to the data inconsistency problem between the structured database and object storage.

Maintaining data consistency across two data storages implies guaranteeing both storages are either successfully updated or fail for the same invocation. Otherwise (i.e., if only one of them succeeds), it leads to data inconsistency. To overcome this problem, we develop the *fail-safe state transition* mechanism that disregards the data update in the object storage if Invoker fails to update the structured part of the object data in the structured database. For that purpose, the mechanism uses a two-phase versioning scheme to keep track of the unstructured data. As shown in Figure 6, in the first phase, the mechanism creates a version ID for each file (unstructured data) and keeps them as structured data (metadata of object data) to track the current version of the file. In the second phase, which occurs upon function completion, Invoker changes all version IDs associated with the updated files (unstructured data) and then writes them to IMDG and the structured database.

For example, consider object o_1 that has file f_1 with the version ID v_1. Upon function invocation, f_1 is updated and written to the object storage with version ID v_2. After the execution, the Invoker must change the version ID from v_1 to v_2 and commit the new structured object data. If any operation fails within this process, the next invocation still loads o_1 with version ID v_1, as if the previous invocation never happened. In the last step, when the invocation is complete, the Invoker purges the old and unused versions of data.

2) Resilience against race condition: Race conditions in Oparaca can occur when multiple invocations modify the same object data simultaneously, resulting in potential data inconsistency. One way to prevent this issue is by using database transactions; however, this method lacks abstraction as it allows direct function code access to the database and is tightly dependent on the type of database. An alternative approach to avoiding race conditions is the cluster-wide pessimistic locking mechanism to synchronize the locking state for all invokers. Nevertheless, this approach necessitates additional network communication to coordinate the locking state, which can lead to scalability issues. Alternatively, we develop an improved version of this mechanism, called "localized locking," which

relies on consistent hashing to direct the invocation request to the invoker that owns the primary copy of the targeted object data. Each invoker will only need to lock the object locally without additional network communication, making it more scalable than the cluster-wide version. Additionally, our localized locking approach guarantees that requests to the same object are executed in the arrival order, which is necessary in certain use cases where order matters, such as seat reservations. This is difficult to achieve with cluster-wide locking.

3) Failure recovery in Oparaca: To further establish resilience against failures, Oparaca is equipped with a mechanism to self-recover from the failure. Broadly speaking, a function invocation failure can be recovered by simply retrying the invocation. However, this approach can cause data incorrectness owing to the execution of the function more than once. The retrying approach could be undesirable for synchronous invocations because the failure can be handled on the client side. For asynchronous invocations, however, we need to guarantee that any invocation is only executed *exactly once*.

To achieve the *exactly-once* guarantee, we have to prevent three sources of the problem that are: (a) losing messages, (b) duplicating messages, and (c) processing messages more than once. Message brokers with stable storage (e.g., Kafka [24]) have features that can be leveraged to address these problems. To solve the first problem, upon failure occurrence, the Invoker can detect and reprocess the incomplete request using an offset number that is automatically generated by the message broker. The offset number is the auto-incremental number based on the message's arrival order and can be used to track the message's position in the queue. The second problem of producing duplicated request messages can be resolved using the message broker's "idempotent producer" feature.

However, the message broker cannot completely address the third problem. That is, the Invoker can process the same invocation request more than once when the message broker has not acknowledged the completed one before the system failure occurs. We prevent this problem by tracking the offset number of the last processed request and adding it to each object metadata. In this manner, before processing an invocation request, Invoker checks the offset number of the target object to see if it is lower than the offset number of an incoming request. When the condition is met, the Invoker can detect that it has not been processed and perform the normal operation. Otherwise, it must be skipped to avoid reprocessing.

G. Dataflow Abstraction in Oparaca

To offer a high-level abstraction to declare a workflow, Oparaca provides the dataflow abstraction as a built-in feature that enables developers to declaratively define the invocation steps as a directed acyclic graph (DAG) in a domain-specific language (DSL) with YAML format. In every step, the developer can declare the output of each invocation as a temporary variable within the workflow. Then, the next invocation can use the temporary variables from previous steps as the input or target to call the function. Upon registering a dataflow function by the developer, Invoker constructs the DAG by having the invocation step as the edge and the objects as nodes.

Upon calling the dataflow function, one of the Invokers takes on the role of orchestrator, similar to the orchestrator pattern [30] in microservices. It breaks down the dataflow into multiple lower-level invocations and forwards them based on the topological order of DAG. Using consistent hashing, the invoker can determine the address of the target object and send the request directly to another Invoker that holds the target object. When each step is completed, the orchestrator keeps track of the intermediate dataflow state to transparently operate the data exchange between invocation steps. With the orchestrator pattern, the dataflow control logic is centralized into a single invoker, simplifying the management, monitoring, and error-handling implementation.

When using the orchestrator pattern, the exact-once guarantee may be compromised because the object data is stored separately from the dataflow state. If the guarantee is needed, Oparaca allows flagging all invocation steps as immutable. Upon handling the dataflow request, Oparaca can generate the output ID in advance for each step, making each step of dataflow execution idempotent and safely re-executable.

V. Discussion

1) Security: Certain security measures can be implemented in Oparaca to strengthen it against potential attacks. The *first* measure is to reduce the attacking surface by limiting the necessary inbound traffic to the function container. As the function container is only accessed by the Invoker, the traffic policy can be configured to block inbound traffic except from the Invoker. The *second* measure is to avoid reusing secret tokens. To prevent the function container from accessing out-of-context data via analyzing the URL path, we use the presigned URL mechanism for object storage. Thus, object storage in Oparaca is more secure than in FaaS, where the same secret key is used for every request. To secure the storage adapter, we can make the Invoker generate a unique secret token for each task, and every request for the storage adapter must be authenticated via the secret token.

2) Multi-tenancy: The primary concern of multi-tenancy is ensuring data and resource isolation. The fundamental idea is to prevent sharing classes and functions among tenants. Since custom functions are offloaded and executed in a FaaS engine that provides strong isolation—with no shared functions—the execution environment is effectively contained within the FaaS engine. Regarding the Invoker and data management module, it is possible to share these components, as the data is stored separately in each class. However, depending on the billing model and isolation requirements, we can enhance security and resource isolation by separating these components for each tenant.

3) Cold Start: The developer functions and the Oparaca components can benefit from scale-to-zero to reduce the cost when there is no usage. However, this has the side effect of causing more cold starts. Since Oparaca 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 entirely depends on the underlying serverless execution engine.

VI. Performance Evaluation

A. Experimental Setup

We deploy the Oparaca platform on 4 machines of Chameleon Cloud [23], each with 2 sockets of 24-Core Intel(R) Xeon(R) Gold 6240R CPU processors that collectively have 192 cores, 768 GB memory, and SSD SATA storage. We set up the Kubernetes cluster, which includes 15 VMs with 16 vCPUs and 32 GB of memory. We made another 2 VMs for the S3-compatible storage (Minio [20]) for unstructured data and ArangoDB ([2]) for structured data. Oparaca is implemented using Java with Infinispan [21] for IMDG.

Baselines. We configure Apache Flink Stateful Function (StateFun) [15], OpenWhisk [1], and Knative [3] to serve as the baselines. Unlike Oparaca and OpenWhisk, which focus on API calls and event handling, StateFun is an open-source stateful serverless system focusing on stream processing. Because StateFun does not manage the function worker instances out of the box, we configure Knative to complement it. OpenWhisk and Knative are popular open-source stateless FaaS platforms that we use to represent the state management done by the developer.

We used Gatling [12] for load generation and implemented three applications to serve as the workload. First is the video transcoding function, which utilizes FFmpeg [37], a CPU-intensive application. The second is a text concatenation function that concatenates the content of a text file (state) with an input string. This function represents a highly IO-intensive workload. Third is the JSON update function, which uses only structured data in JSON and is used to insert key-value pairs into the JSON state data randomly. The remaining workload characteristics are specific to each experiment and are explained in the respective sections. All three functions are implemented in the Python language.

B. Analyzing the Imposed Overhead of Oparaca

The abstractions provided by Oparaca are not free of charge and introduce some time overhead to the applications using these abstractions. In this experiment, our aim is to measure this overhead and see how the efficient design of Oparaca can mitigate this overhead. The latency of a function call is the metric that represents the overhead. We mainly study two sources of the overhead: (a) The *state data size* that highlights the overhead of OaaS in dealing with the data, and (b) The *concurrency of function calls* that highlights the overhead of the Oparaca system itself.

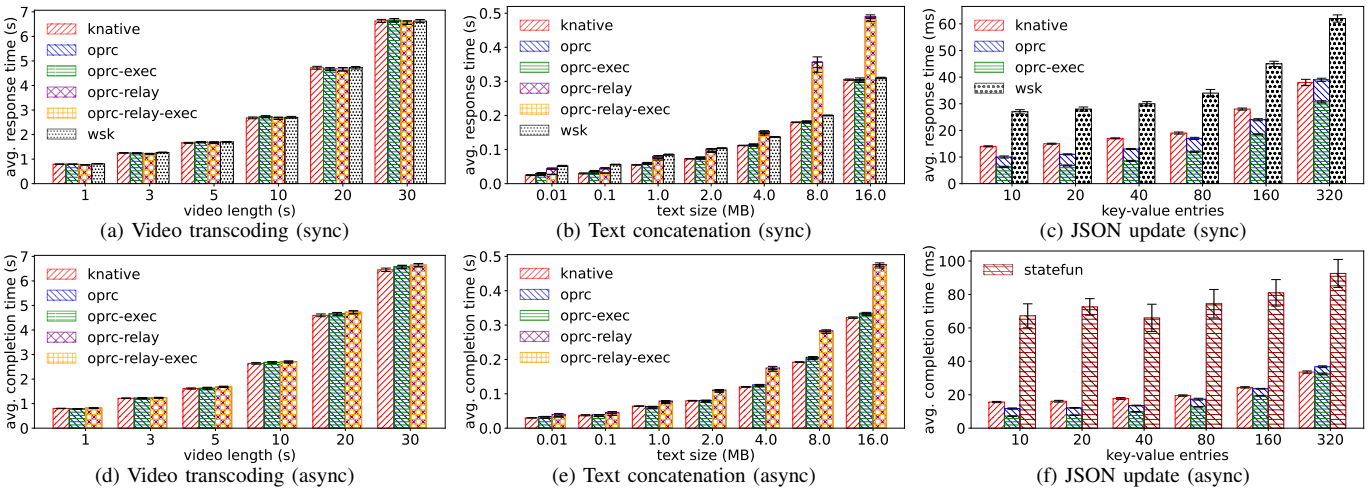
The impact of changing the state size is shown in Figure 7. To generate objects with various state sizes, we increased the input video length from 1—30 seconds. To remove the impact of video content on the result, the longer videos were generated by concatenating the same 1-second video. Similarly, the text files are from 0.01—16 MB. For 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 pairs. 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 set it to repeat this operation 100 times. To analyze the improvements offered by the URL redirection, we examine two versions of Oparaca: the full version (expressed as *oprc*) and without URL redirection (expressed as *oprc-relay*). The horizontal axes represent different state sizes for video, text, and JSON, respectively, and the vertical axes represent the average response/completion time (latency).

In Figure 7, the average task execution time increases for larger state sizes. For the video transcoding function, all of the platforms perform with similar latency, which is expected because of the compute-intensive nature of the video transcoding that dominates the completion time. In a text concatenation function, however, Knative performs slightly better than Oparaca because of the overhead of unstructured state access by the redirection of the presigned URL. However, if we compare Oparaca with another version that uses a relay mechanism to provide the state abstraction, it performs much lower than its alternative with an average of 30% lower response time. Lastly, we can see all the described trends happen similarly for synchronous and asynchronous request types.

In the JSON update function (Figures 7c and 7f), Oparaca can perform with lower latency than Knative because the function does not need to fetch the object data from the database because of the pure function semantic. Nevertheless, Knative can catch Oparaca by increasing the key-value entries to 320. The reason is that the gain from eliminating the database connection is surpassed by the overhead of moving the data to the function code for larger records. OpenWhisk and Knative have the same pattern because both of them are FaaS, but OpenWhisk performs significantly worse. In Figure 7f, the Statefun shares the same pattern with Oparaca with the consistent gap because it also relies on local storage to keep the function state without the need to fetch the data from the database. We also observe that Statefun performance degraded compared to our initial results [27]. This is because the storage hardware being used for the experiment has a lower through, which impacts the performance of Statefun.

The impact of concurrent function invocations on the Oparaca overhead is shown in Figures 8. In synchronous invocation, we increase the number of concurrent invocations of the same function (horizontal axes), whereas, for asynchronous invocation, concurrency depends on the system implementation which cannot be forced directly; thereby, we use the request arrival rate to increase the concurrency of invocations. To remove the impact of any randomness, We disabled the auto-scaling and limited the number of worker instances to 6. We also exclude OpenWhisk from this section because the Python runtime in OpenWhisk does not support container-level concurrency.

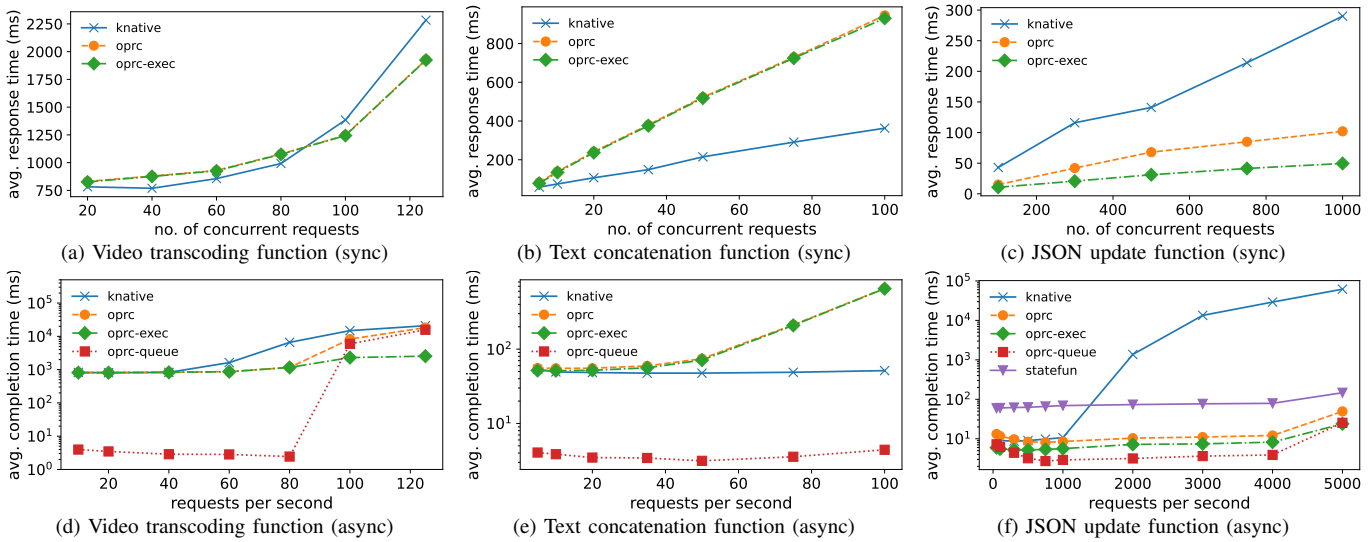
For the transcoding function (Figures 8a and 8d), at the low concurrency levels ($<$ 80 invocations), Oparaca has average response times higher than Knative, but for the higher concurrency levels, the response time of Knative grows faster than Oparaca due to computing resource limitations. Oparaca doesn't need to fetch video file metadata, giving it an edge at high concurrency. In the concatenation function (Figures



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

(d) Video transcoding (async) (e) Text concatenation (async) (f) JSON update (async)

Fig. 7: The average execution time of functions for objects with various state sizes in synchronous and asynchronous invocations. Two versions of Oparaca are examined: the full version and the version without URL-redirection (*oprc-relay*). We also capture the time used by the internal Knative in both Oparaca versions and show them with the suffix *-exec* and plot them in the same bar as their Oparaca version.



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

(d) Video transcoding function (async) (e) Text concatenation function (async) (f) JSON update function (async)

Fig. 8: The average completion time of functions upon varying the rate of incoming requests in synchronous and asynchronous invocations. *oprc-queue* is the queuing time that requests stay within the message queue

8b and 8e), however, this phenomenon does not happen. The difference is that text concatenation is IO-intensive and desires high network bandwidth. The overhead of unstructured data access overwhelms the performance gain from eliminating structured data fetching.

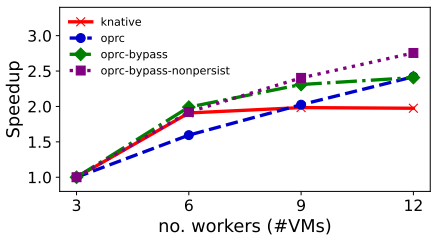
For the JSON update function (Figures 8c and 8f), Oparaca can effectively reduce the latency by eliminating the need to fetch from the database. In Figure 8f, because Statefun also shares this invocation scheme and, therefore, offers less completion time than Knative. However, since it relies on local storage to keep the state, while Oparaca uses the memory, Statefun's completion time is higher than Oparaca's.

In sum, Oparaca improves performance by eliminating database fetching but adds overhead by accessing unstructured data for secure state abstraction. Depending on the workload, this can either improve or impair object function invocation performance. The overhead may outweigh I/O-intensive workloads, but Oparaca can improve latency by up to 2.27x compared to Knative for workloads without unstructured data.

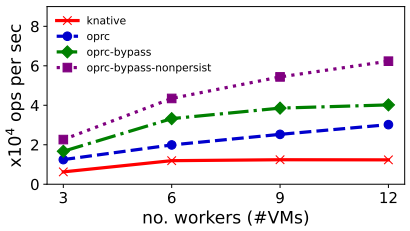
Takeaway: *Object abstraction can be provided with an insignificant latency overhead for objects with only a structured state. The main object overhead occurs as a result of securing unstructured data access.*

C. Scalability of the Oparaca Platform

To study the scalability, we scale out the Kubernetes workers from 3—12 VMs, each with 16 vCPU cores (in total 48—192 vCPUs). We measured throughput and speedup metrics, focusing on the JSON update function, which does not rely on slow object storage, which becomes the bottleneck of this experiment. We measure the throughput by continually increasing the concurrency until the throughput stops growing (Figure 9b). We assume three VMs as the base speedup=1, and the speedup of other cluster size is calculated with respect to the base value. Moreover, we add two other versions of Oparaca: first is *oprc-bypass* that uses a standard Kubernetes deployment as its underlying function execution instead of Knative; Second is *oprc-bypass-nonpersist* that does not persist the object data to the database to measure if Oparaca is



(a) Speedup results by horizontal scaling



(b) Throughput results from horizontal scaling

Fig. 9: Evaluating the scalability of the OaaS platform against other baselines.

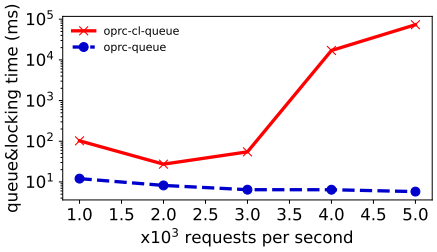


Fig. 10: Evaluating the performance of localized locking compared to cluster-wide locking

not bottleneck by the database write operation.

According to Figure 9a, the speedup of Knative plateaus after reaching 6 VMs. We realized that this plateau is attributed to the database write operation throughput bottleneck. Conversely, Oparaca exhibits the potential for higher speedup enhancement due to its reliance on the distributed in-memory hash table to consolidate data for batch write operations. This approach can boost maximum throughput by up to 3.27x when comparing *oprc-bypass* with *knative*.

Figure 9b shows that *oprc-bypass* yields a higher throughput over the baseline Oparaca. This is because Oparaca sends task data through the Knative internal proxy to offload the task to Knative. While this setup allows for scale-to-zero functionality, bypassing these components leads to even higher throughput. Furthermore, by disabling the database writing operation, which is the bottleneck, *oprc-bypass-nonpersist* can achieve even higher throughput. Although there isn't linear scalability due to the limitations of the database write performance, Oparaca significantly improves maximum throughput compared to traditional FaaS systems.

Takeaway: *In addition to offering a higher-level abstraction, Oparaca can improve the throughput and response time of its underlying Knative engine via reducing database operations, thereby, mitigating its bottleneck.*

D. Performance of Localized Locking

To analyze effectiveness of localized locking, we created a variation of Oparaca, called *oprc-cl*, that has cluster-wide locking and evaluated it using a cluster of 12 Invokers while increasing the request arrival rate to measure the locking overhead. To generate requests involving the locking mechanism, we created multiple requests targeting the same object. From Figure 10, the overhead of localized locking remains mostly constant, while the overhead of cluster-wide locking rises for higher request rates. The cluster-wide version does not exhibit this behavior, as the network communication overhead limits the throughput and hinders high invocation rates.

E. 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.

Case Study # 1. expression detection system. This case study is a video processing workflow that performs face detection and facial expression recognition. Figure 11 shows the automatic system uploads the video file to the object

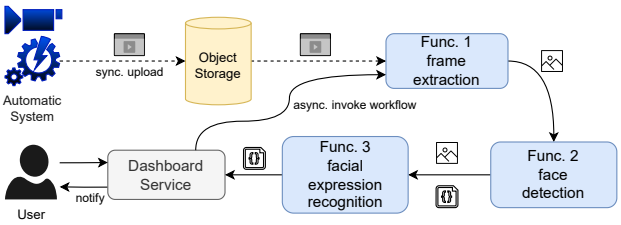


Fig. 11: The use case of developing a face expression recognition workflow for an input video.

storage to be processed by the workflow of functions. The workflow includes: `Func_1` to split the video into multiple image segments; `Func_2` to detect the face on each sample image frame; and `Func_3` to perform facial expression recognition on the detected face image and generate a `JSON` format label. These functions must persist their output object so that the next function in the workflow can consume it.

FaaS implementation. The developer must implement the following steps: (a) Configuring cloud-based object storage and managing access tokens. (b) Implementing business logic to respond to trigger events. (c) Manage data within the functions that involve three steps: allocating the storage addresses, authenticating access to the object storage, and performing fetch and upload operations to the allocated addresses. Upon implementing these functions, the developer has to connect them as a workflow via a function orchestrator service. Finally, the dashboard service invokes the workflow upon receiving a user request and collects the results.

OaaS implementation. The developer defines three classes, namely `Video`, `Image`, and `Expression`, in the form of the three following classes: (a) `Video` class with `frame_extract()` functions; and a macro function, `df_exp_recog(detect_interval)`, that includes the whole dataflow of function calls, with the requested sampling period as its input, and an `expression_data` object as the output. (b) `Image` class with the `face_detect()` and `exp_recognize()` functions. (c) `Expression` class that does not require any function. The dashboard service calls the `wf_exp_recognize(detect_interval)` dataflow function directly using the object access interface and receives the `Expression` object as the output. We note that the developer does not need to be involved in the data locating and authentication steps when developing the class functions because of the abstraction that OaaS provides.

Case Study # 2. content moderation system. Moderating the content at scale in various formats, including image, document, and video. We first present how the application is deployed in FaaS, the limitations of this approach, and how

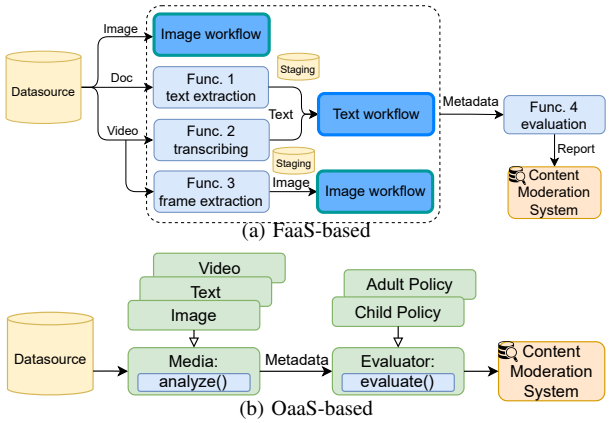


Fig. 12: The automatic content moderation system.

OaaS can resolve those limitations.

FaaS implementation. To simplify complex multimedia processing workflows [7], we abstract the workflow to extract the metadata from the image files as `Image workflow`, and the workflow to extract metadata from the text files as `Text workflow` as shown in Figure 12a. Before using both workflows, the content must be pre-processed to extract raw data via using the pertinent FaaS functions: (a) `text extraction` to extract text from the document. (b) `transcribing` to extract text from the video. (c) `frame extraction` to sample image frame from the video. After feeding the data into both workflows to extract the metadata, we use the `evaluation` function to generate a report to the content moderation system.

The FaaS implementation has three main drawbacks: **(A)** developers must explicitly manage application state and data using separate storage services that increase the complexity. **(B)** even though the common workflow can be reused, the intermediate data management is not abstracted. That is, if the developer needs to separate/change the staging storage, it must be done manually. **(C)** functionalities may require numerous and heterogeneous FaaS deployments—e.g., requiring a separate workflow for each content type, where the Video format requires a more complicated workflow than the other types. These drawbacks complicate development, deployment, and management as the application evolves to handle various document types and integrates more functionalities and options (e.g., using multiple evaluation services instead of one).

OaaS implementation. To demonstrate the efficacy of OaaS, we transform the given FaaS-based solution into OaaS with minimal effort to resolve the aforementioned drawbacks.

- **Workflow Construct.** We encapsulate related FaaS functions and states into classes representing two key functionalities: `Media` to extract the metadata and `Evaluator` to evaluate metadata and report to the content moderation system. The two classes form the critical path of the application processing pipeline, as shown in Figure 12b.
- **Object Encapsulation.** We use inheritance and polymorphism to enhance software reuse by encapsulating FaaS functions and states in classes derived from two base classes. This approach hides the need for storage services behind object abstraction and allows their implementation to be managed in the cloud. It simplifies development, as developers only build the processing pipeline once in the base classes. They can then focus on specific functionalities in the derived classes, avoiding **redundant** implementation when adding new content types or evaluator services.

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

VII. Conclusions

In this research, we present OaaS paradigm that simplifies cloud-native development by unifying state and workflow management into object abstractions. Our prototype, Oparaca, supports both structured and unstructured data with consistency guarantees through fail-safe state transitions, presigned URLs for secure low-overhead data access, and DHT-based caching for improved locality. Fault tolerance is ensured via *exactly-once* execution and *localized locking* schemes. The declarative dataflow abstraction transparently handles concurrency and synchronization. Evaluation results demonstrate that Oparaca streamlines development with scalability and negligible overhead, particularly for compute-intensive tasks. Future work includes supporting multi-datacenter deployments for large-scale applications.

Acknowledgement

This project is supported by National Science Foundation (NSF) through CNS CAREER Award# 2419588.

References

[1] Apache OpenWhisk. https://openwhisk.apache.org/.
[2] Arangodb. https://www.arangodb.com. Accessed 11 June '24.
[3] Knative. https://knative.dev/. Online; Accessed on 11 June 2024.
[4] Amazon. AWS IAM | Identity and Access Management | Amazon Web Services. https://aws.amazon.com/iam/. Accessed on 11 June. 2024.
[5] Amazon. AWS Step Functions | Serverless Microservice Orchestration. https://aws.amazon.com/step-functions. Accessed on 23 Jul. 2022.
[6] Amazon. Cloud Object Storage | Amazon S3 – Amazon Web Services. https://aws.amazon.com/s3/. Online; Accessed on 11 June. 2024.
[7] Amazon. Dynamic Video Content Moderation and Policy Evaluation Using AWS Generative AI Services. Accessed 22 December 2024.
[8] S. Bangera. *DevOps for Serverless Applications: Design, deploy, and monitor your serverless applications using DevOps practices*. Packt Publishing, 2018.
[9] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSRTR2014*, 41, 2014.
[10] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proc. of the VLDB Endowment*, 15(8):1591–1604, 2022.
[11] Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman Böhringer, Rodrigo Bruno, and Torsten Hoefler. Process-as-a-service: Unifying elastic and stateful clouds with serverless processes. In *Proceedings of the 15th ACM Symposium on Cloud Computing*, pages 223–242, 2024.
[12] Gatling Corp. Gatling - Professional Load Testing Tool. https://gatling.io/. Online; Accessed on 11 June. 2024.
[13] Chavit Denninnart and Mohsen Amini Salehi. Smse: A serverless platform for multimedia cloud systems. *Concurrency and Computation: Practice and Experience*, 36(4), 2024.
[14] 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.
[15] Apache Flink. https://nightlies.apache.org/flink/flink-statefun-docs-stable. Accessed 11 June 2024.
[16] Durable Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities. Accessed on 11 June 2024.

[17] 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.

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

[19] Yahya Hassanzadeh-Nazarabadi, Sanaz Taheri-Boshrooyeh, Safa Otoum, Seyhan Ucar, and Öznur Özkasap. Dht-based communications survey: architectures and use cases. *arXiv preprint arXiv:2109.10787*, 2021.

[20] MinIO Inc. https://min.io/. Accessed 11 June 2024.

[21] Red Hat Inc. Infinispan. https://infinispan.org/. Accessed 11 June 2024.

[22] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[23] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. Lessons learned from the chameleon testbed. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 219–233, 2020.

[24] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[25] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.

[26] Pawissanutt Lertpongrujikorn, Hai Duc Nguyen, and Mohsen Amini Salehi. Streamlining cloud-native application development and deployment with robust encapsulation. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 847–865, 2024.

[27] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. Object as a service (oaas): Enabling object abstraction in serverless clouds. In *16th International Conference on Cloud Computing*, pages 238–248, 2023.

[28] 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.

[29] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 57–72, 2023.

[30] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[31] Zhenyuan Ruan, Seo Jin Park, Marcos K Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving {Microsecond-Scale} resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, 2023.

[32] 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.

[33] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 138–152, 2021.

[34] 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.

[35] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[36] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

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

[38] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Trans. on Knowledge and Data Eng.*, 27(7):1920–1948, 2015.

[39] 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, Nov. 2020.

Pawissanutt Lertpongrujikorn received B.Eng. in computer engineering from Kasetsart University, Thailand in 2019. Currently, He is working toward his Ph.D. at HPCC Lab, University of North Texas. His research focuses on developing a cloud-native programming paradigm and serverless computing.

Dr. Mohsen Amini Salehi is an Associate Professor and the director of the HPCC Lab, at the Computer Science and Engineering dpt., University of North Texas. His team focuses on democratizing cloud-native application development and building smart and trustworthy systems across edge-cloud. He is an NSF CAREER Awardee and, so far, he has had 11 research projects local and federal agencies.