Streamlining Cloud-Native Application Development and Deployment with Robust Encapsulation

Pawissanutt Lertpongrujikorn[1,*], Hai Duc Nguyen[2,*], and Mohsen Amini Salehi[1]
[1] HPCC Lab, University of North Texas, USA
[2]Argonne National Laboratory and University of Chicago, USA
{pawissanutt.lertpongrujikorn, mohsen.aminisalehi}@unt.edu, ndhai@cs.uchicago.edu

ABSTRACT

Current Serverless abstractions (e.g., FaaS) poorly support non-functional requirements (e.g., QoS and constraints), are provider-dependent, and are incompatible with other cloud abstractions (e.g., databases). As a result, application developers have to undergo numerous rounds of development and manual deployment refinements to finally achieve their desired quality and efficiency. In this paper, we present Object-as-a-Service (OaaS)—a novel serverless paradigm that borrows the object-oriented programming concepts to encapsulate business logic, data, and non-functional requirements into a single deployment package, thereby streamlining provider-agnostic cloud-native application development. We also propose a declarative interface for the non-functional requirements of applications that relieves developers from daunting refinements to meet their desired QoS and deployment constraint targets. We realized the OaaS paradigm through a platform called Oparaca and evaluated it against various real-world applications and scenarios. The evaluation results demonstrate that Oparaca can enhance application performance by 60× and improve reliability by 50× through latency, throughput, and availability enforcement—all with remarkably less development and deployment time and effort.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; **Cloud computing**; **Self-organizing autonomic computing**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '24, November 20–22, 2024, Redmond, WA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1286-9/24/11…$15.00
https://doi.org/10.1145/3698038.3698552

KEYWORDS

serverless, cloud computing, function-as-a-service, object-as-a-service, cloud-native programming, abstraction

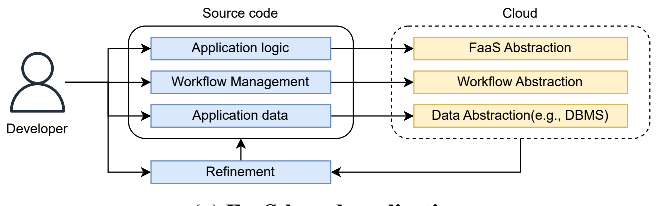
ACM Reference Format:
Pawissanutt Lertpongrujikorn[1,*], Hai Duc Nguyen[2,*], and Mohsen Amini Salehi[1]. 2024. Streamlining Cloud-Native Application Development and Deployment with Robust Encapsulation. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3698038.3698552

1 INTRODUCTION

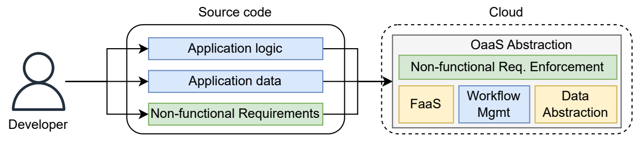
Function-as-a-Service (FaaS), or serverless computing, has emerged as a transformative paradigm in cloud computing, redefining how businesses and individuals develop and deploy applications. Unlike traditional virtualized infrastructure (e.g., virtual machines), FaaS enables on-demand code execution in response to events, eliminating the need to manage servers or underlying infrastructure. Developers leverage FaaS through high-level abstractions provided by cloud platforms, allowing them to focus on writing and running functions rather than managing complex systems, thereby significantly enhancing productivity.

Unfortunately, beyond hiding complexity, FaaS plays a very limited role in other aspects. Primarily, FaaS functions only offer resource and computation abstraction, which is insufficient for a complete application deployment [23, 45, 81]. Developers must rely on additional cloud services, such as databases [3, 5] and orchestrators [4, 15], to manage application states and workflows. Yet, there are limited integration supports among these abstractions. In stateful applications, for example, FaaS performance depends on data locality, but current FaaS implementations provide no means to help FaaS functions cooperate with cloud data abstractions in this regard, negatively affecting productivity and efficiency. Furthermore, the FaaS abstraction is implemented by cloud providers, who usually prioritize system metrics, such as resource utilization, which can result in unpredictable and uncontrollable quality degradation on the application side [71]. This leads to counterproductive interactions between developers and

[*]These authors contributed equally to this work



(a) FaaS-based application



(b) OaaS-based application

Figure 1: OaaS extends the FaaS abstraction to encapsulate everything into a single deployment with built-in non-functional enforcement, boosting productivity and efficiency.

the cloud, such as over-provisioning and over-subscription [74, 99]. These limitations force developers to manually reconfigure their deployments through multiple rounds of refinement for their *non-functional requirements* (e.g., Quality of Service, a.k.a. QoS). The process lacks proper guidance and relies heavily on resource-domain expertise and experience [50], making cloud application development and deployment complex and costly [73] (see Figure 1a).

To resolve the problem, we propose Object-as-a-Service (OaaS) abstraction, a new cloud computing paradigm that borrows the concepts of object-oriented programming to let the cloud applications include their logic (i.e., functions), data (i.e., state), and non-functional requirements into a single deployment package (see Figure 1b). The OaaS abstraction allows developers to unify the application functionality implementations into single packages, eliminating the multi-abstraction barriers that hinder productivity and efficiency. The abstraction also comes with a "non-functional requirement interface" that allows applications to declare the expected QoS requirements and constraints as high-level and measurable metrics. Developers can use the interface to express their requirements, and then the cloud provider will enforce them automatically, removing the need for repeated refinements. This unlocks new opportunities for cloud optimization. With OaaS deployment, the cloud providers are given a clear set of optimization objectives (from the "non-functional interface") with a rich set of information (from the "object") so that they will know the right direction to optimize their system metrics while still be able to meet their customer requirements.

We implement Oparaca, an open-source platform that realizes ideas of OaaS to simplify application deployment. Oparaca integrates various object deployment and management approaches, each specialized for specific object structures and requirement combinations, and then optimizes them for different deployment scenarios. Thus, making object deployment portable and efficient. We systematically evaluate Oparaca versus state-of-the-art solutions through various experiments on real Cloud testbeds. We found that Oparaca often outperforms the other baseline approaches in efficiently meeting the targeted performance objectives (e.g., throughput) of multiple services for different application types. In contrast, other baselines struggle due to resource contention issues arising from the lack of service-specific awareness of the targeted performance objective.

In sum, the contributions of this research are as follows:

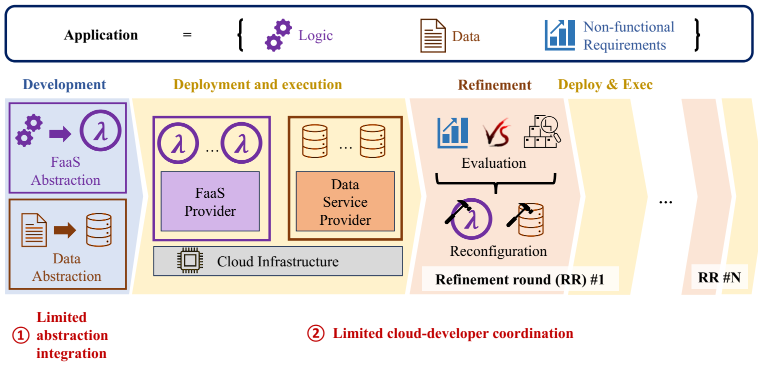
- Object-as-a-Service (OaaS) abstraction that exploits the Object-oriented programming concepts to encapsulate functional and non-functional requirements into one deployment package, enhancing application development and deployment productivity.
- A non-functional requirements interface that lets developers express their non-functional requirements in a human-friendly and measurable manner, thus enabling application portability and opportunities for cooperative cloud-application interactions.
- Oparaca – an OaaS prototype implementation that enables simple, scalable, and QoS-aware applications development.
- Evaluating and analyzing the Oparaca from the QoS enforcement, efficiency, and productivity perspectives. By leveraging the OaaS abstraction, applications can improve their performance by 60× and availability by 50× with much less deployment time and effort.

The remainder of the paper is organized as follows. In Section 2, we present the issues of the current FaaS abstraction that make delivering efficient solutions complicated and expensive. We show how OaaS abstraction resolves these issues in Section 3 and with more details in Sections 4. Section 5 evaluates Oparaca against state-of-the-art solutions under various scenarios. We briefly present related work in Section 6 and summarize the paper in Section 7.

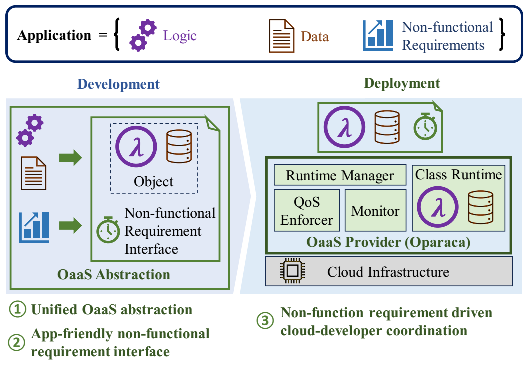
2 MOTIVATION AND PROBLEM STATEMENT

Figure 2a shows a typical life cycle of a FaaS-based application that consists of three primary phases:

- *Development*: The application developers design suitable logic/algorithms and data structure for the application and then encapsulate them into *separated deployment packages* (e.g., FaaS deployments, database schema, etc.).
- *Deployment and Execution*: cloud provider receives deployment packages and executes them separately across their infrastructure through *service providers* (e.g., FaaS



(a) Current FaaS-based application life cycle and its problems



Application = { Logic Data Non-functional Requirements }
Development
Deployment
Object
Non-functional Requirement Interface
OaaS Abstraction
Runtime Manager
Class Runtime
QoS Enforcer
Monitor
OaaS Provider (Oparaca)
Cloud Infrastructure
① Unified OaaS abstraction
② App-friendly non-functional requirement interface
③ Non-function requirement driven cloud-developer coordination

(b) OaaS and non-functional requirements abstraction

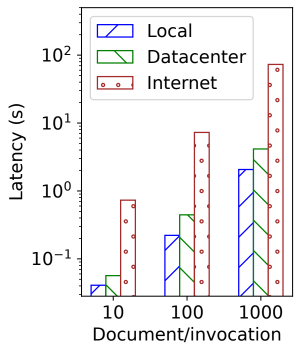
Figure 2: Limited non-functional support from the cloud introduces repeated and complex refinement processes for quality of service. In contrast, the OaaS abstraction outsources the refinement to the cloud provider with a well-defined set of information and refinement objectives. Thus making the application development and deployment more productive.

and data service providers). Each one is implemented and optimized specifically for a cloud service abstraction (e.g., AWS Lambda for FaaS abstraction, MySQL for Relational DB, etc.).

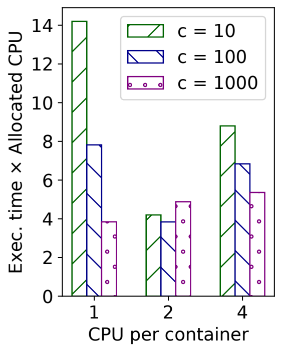
- *Refinement*: Applications typically have non-functional requirements specifying their QoS (e.g., desired throughput, availability, etc.) and execution constraints (e.g., budget, Carbon footprint, jurisdiction, etc.). To ensure these requirements, developers have to evaluate them against monitored data. Refinement, which includes reconfiguring and redeploying the application packages, is needed if any of these requirements fail to be met.

In practice, the refinement phase consists of multiple rounds of reconfiguration and deployments that cost a lot of time and effort [50, 73]. This is because *current FaaS implementations and their supportive services offer limited supports that make it complicated and expensive to deliver efficient cloud applications*. The problem manifests in many aspects of cloud application life cycles, as outlined below.

Limited Abstraction Integration. FaaS applications are formed based on FaaS functions and additional cloud services, such as databases and workflow orchestrators. However, these abstractions typically operate independently. This independence creates challenges, as even a single functionality may involve multiple abstractions but lacks the capability to integrate them effectively. In stateful applications, for example, FaaS functions typically need access to an external database to read and update its state. This makes the application performance strongly depend on efficient data transmission between the FaaS invocations and the database. Figure 3a illustrates the end-to-end latency of a FaaS invocation chain modifying JSON documents from a database (see Section 5.1), varying the database's location relative to the function's containers



(a) Locality Impact



(b) Configuration Impact

Figure 3: FaaS limitations (a) prevent applications from exploiting locality for performance and (b) complicate deployment refinement.

and the number of JSON documents modified. Clearly, executing a function on the same machine as the database ("Local") is significantly faster than running the function from a different machine within the same data center (2×) or across the Internet (35×). The latency difference increases substantially as more data is transmitted (e.g., the "Datacenter" to "Local" latency ratio increases by 1.4× when we increase the number of modified documents per invocation from 10 to 1000). Based on the results, FaaS should leverage data locality by dispatching functions close to the database to minimize end-to-end latency. Unfortunately, current FaaS abstractions lack support for such cross-abstraction optimization, forcing developers to create their own solutions, which is both challenging and effort-consuming [17, 43, 61, 69, 76, 94].

Limited Cloud-Developer Coordination. Developers refine application deployments through primitive resource-domain settings, like per-container CPU allocation. On the other hand, non-functional requirements are typically measured and evaluated using application-domain metrics, such as throughput, latency, and monetary cost. Translating these requirements into effective FaaS configurations is challenging. Figure 3b illustrates the resource cost, measured by the average *execution time* $\times$ *CPU* allocated to the JSON randomization deployment under different per-container concurrency and CPU allocation configurations. Considering only one factor for cost minimization is insufficient; for instance, with one CPU per container, varying concurrency (c) can change costs by up to 4.3$\times$, but doing so has little effect with two CPUs per container. Configuring these factors together is necessary, but there are no clear insights into how to do so. For example, increasing concurrency (c) generally allows more invocations per container, reducing costs. However, setting the concurrency too high can lead to resource contention, which prolongs invocation execution and increases costs. The optimal concurrency thresholds vary with different CPU allocations. With two CPUs per container, $c = 1000$ is too high, but it works well for configurations with one or four CPUs per container. Therefore, configuring such low-level parameters to produce a reliable and robust deployment demands significant effort and expertise (e.g., [14]), often necessitating numerous rounds of refinement [48, 50, 58].

Worse, the implementation and configuration of FaaS and supporting cloud abstractions are influenced by the cloud providers' objectives, potentially hindering the fulfillment of application non-functional requirements. For example, many cloud resource managements employ over-subscription [11, 30, 56], which implicitly commits more resources to users than the cloud can actually provide for better utilization. However, this practice increases the risk of interference when multiple applications peak simultaneously, leading to uncontrollable and unpredictable QoS degradation [21, 24, 80, 99]. To counteract this, many applications request more resources than they need [74, 78], prompting providers to oversubscribe even more aggressively [13, 14, 51]. This creates a harmful cycle of overestimation and mistrust, negatively affecting both applications and the cloud infrastructure [33].

3 OBJECT-AS-A-SERVICE ABSTRACTION

To establish an agile and cost-efficient application delivery, the two challenges presented in Section 2 must be properly addressed. In this section, we propose solutions for each challenge and then combine them to form a novel approach for FaaS-based application development and deployment.

3.1 Unified OaaS Abstraction

We extend the FaaS abstraction, called Object as a Service (OaaS), that borrows the object-oriented programming (OOP) concepts to unify application logic and data within a single abstraction. Specifically, each application is defined as a collection of cloud objects where its data (a.k.a. state) is modeled as "attributes" with supported data types in current cloud data abstraction, and its logic is modeled as methods realized by serverless functions. In this manner, OaaS abstraction alone is sufficient for the entire application development phase—eliminating the need for multiple distinct abstractions and the complexities of effectively gluing them.

OaaS also offers the notions of abstract class, inheritance, and polymorphism to establish software reuse across cloud objects, thereby reducing redundancy and enhancing development productivity at the FaaS workflow level. This is in contrast to traditional FaaS, which typically limits software reuse to the function or invocation level (e.g., through shared libraries). Beyond these, OaaS transformation unlocks new opportunities for deployment optimizations that were previously difficult or impossible. This is because the object abstraction provides richer information for optimization and grants the cloud greater control over the deployment to exploit them. For example, OaaS lets application data and logic be encapsulated and managed together under the object abstraction. Thus, OaaS can easily find the data associated with each method and proactively distribute them across the cloud database instances that are close to the deployed method, thereby minimizing the data transmission overhead.

3.2 Non-functional Requirement Interface

Within the OaaS abstraction, we develop a non-functional requirement interface that lets the developer express their non-functional requirements in a human-friendly manner. Through the interface, developers can declare their non-functional requirements for a whole object or even for a specific part (attribute or method) of it. The requirements are defined as high-level and measurable metrics either in the form of QoS (e.g., availability and throughput) requirements or deployment constraints (e.g., budget and jurisdiction). During the deployment, the cloud provider takes these non-functional requirements as input to its internal services and adjusts their operations to meet the requirements. The benefits are threefold:

- *Productivity*: applications no longer need to consider low-level resource configuration for non-functional requirements. This relieves the burden of performance optimization from their deployment process, thus improving productivity.
- *Portability*: as long as the cloud provider supports OaaS, the application can rely on the object abstraction to maintain its

Name	Value Type	Unit	Definition
QoS Requirements			
Throughput	**Integer**	**Rps**	**Minimum number of invocations guaranteed to be executed per second**
Availability	**Real**	**Percent**	**The percentage of time an object/function must be available for service.**
Locality	**{Local, None}**	**N/A**	**How function invocations are dispatched with respect to object state location.**
Deployment Constraints			
Persistent	**Yes/No**	**N/A**	**Should the data associated with the object persistent**
Runtime Req.	**Dict**	**N/A**	**Specific object runtime configuration. (e.g., choice of FaaS engine)**
Budget	Integer	Credit	Object deployment and operation budget. All costs must not exceed this value.
Consistency	Enumerate	N/A	Object consistency model: eventual, sequential, linearization, or none.
Jurisdiction	Enumerate	N/A	Candidate places to deploy an object
Data Encryption	Enumerate	N/A	Specify or disable the encryption algorithm for the stored data

Table 1: Potential Non-functional requirements and constraints. Those with bold font are currently supported by Oparaca.

functionality, meet its QoS and constraint expectations (via the non-functional requirement interface), and comfortably deploy across scenarios with minimal changes.

- *Cloud-application symbiosis*: since applications use cloud resources for execution, the common sense is that the cloud should fulfill the non-functional requirement, as it has sufficient knowledge and privilege on the underlying infrastructure. With the non-functional requirement interface, however, the cloud does not take this responsibility alone. Here, the interface acts as a "glue" to make a symbiosis between the cloud and the application developer. Specifically, the requirements declared through the interface are valuable guidelines for cloud service providers to know which optimization they should follow so as not to impact the applications negatively. On the other hand, the interface is a useful means of communication that lets the developer actually configure for performance and quality, as opposed to going through multiple rounds of playing a "trial-and-error" game with the cloud providers to meet the desired outcomes.

3.3 Simplified, Refinement-Free Deployment

Based on the ideas above, as shown in Figure 2b, we propose a novel paradigm to develop and deploy cloud applications. In this paradigm, cloud applications are modeled as a set of objects, each can be developed and deployed independently. An object can possess deployment constraints and QoS requirements declared through the non-functional requirement interface. The object is deployed and managed on the cloud by means of the OaaS abstraction. Specifically, an OaaS-based platform (we call it Oparaca and introduce it in Section 4) receives the object deployment packages from the developer, deploys them on the cloud, and also automatically configures and monitors their resource allocation to meet the defined non-functional requirements.

The proposed paradigm greatly simplifies the process of delivering cloud-native applications. Instead of having multiple logic/data deployments with multiple rounds of development-deployment-evaluation that are subjected to many uncertainties caused by the cloud's shared environment and uncooperative abstraction realization, the application now needs to deal with only one type of abstraction. Moreover, with the non-functional requirements serving as the driving force for the underlying OaaS orchestration, no re-deployment or re-configuration is needed to meet the desired non-functional requirements.

4 OPARACA: AN OAAS REALIZATION

In this part, we first describe the design goals of Oparaca—an open-source platform realizing the ideas of the OaaS paradigm. Then, we introduce new concepts and interfaces needed for this realization, and finally, we discuss its development details.

4.1 Design Goals and Requirements

We use Oparaca as a proof of concept to (1) illustrate how OaaS can reshape cloud application deployment, making it more productive and cost-effective; and (2) highlight how OaaS unlocks new opportunities for a more efficient, collaborative application deployment optimization. To achieve these objectives, we outline the following requirements and try to ensure Oparaca meets them throughout the entire design and implementation process.

(1) *Simplicity*: Extend the concept of *object* in OOP to a service abstraction that allows application developers to encapsulate their application logic, data, and non-functional requirements into a single deployment entity.
(2) *Declaratory*: Provide a simple, human-friendly interface for non-functional requirements that allows developers to

express and achieve desired non-functional requirements with minimum configuration/deployment effort.

(3) *Efficiency*: Oparaca can enforce application requirements at comparable cost versus state-of-the-art solutions.

(4) *Portability*: Oparaca allows applications to deliver proper functionality with desired QoS anytime, anywhere.

Oparaca is implemented in Java and comprises approximately 20,000 lines of code. The platform offers a YAML-based OaaS API for defining objects and their non-functional requirements. Oparaca operates with FaaS functions at the container level using Knative and Kubernetes, and it provides a supported SDK for working with Python. The source code is available at https://github.com/hpcclab/OaaS.

4.2 OaaS Abstraction Interface

To fulfill the first two requirements (i.e., simplicity and declaratory), we provide a deployment interface for OaaS to help developers define the entities of their cloud-native application and non-functional requirements akin to OOP concepts. To that end, the cloud-native application is built on the foundation of *classes*. Each class defines the structure of independent executable objects that are responsible for carrying out one or multiple functionalities. Upon deployment, Oparaca allocates appropriate cloud resources to realize the corresponding objects of the class and manage them to handle workloads. Moreover, Oparaca supports *inheritance* and *polymorphism* for its classes.

Listing 1: OaaS Deployment for Image Processing

```yaml
classes:
  - name: Image
    qos:
        availability: 99.9
    constraint:
        persistent: true
    keySpecs:
      - name: image #File Image;
    functions:
      - name: resize
        qos:
            throughput: 100  #rps
        #container image
        image: img/resize
      - name: changeFormat
        image: img/change-format
      - name: detectObject
        qos:
            throughput: 100
        image: img/detect-object
  - name: LabelledImage
    parent: Image
    keySpecs:
      - name: labels #File labels;
    functions:
      - name: analyze
        qos:
            throughput: 50
```

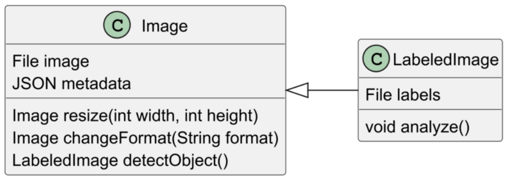

C Image
File image
JSON metadata
Image resize(int width, int height)
Image changeFormat(String format)
LabeledImage detectObject()
C LabeledImage
File labels
void analyze()

Figure 4: Class diagram for the image processing example. The developer can translate the class diagram directly to cloud deployment in Listing 1 through OaaS abstraction.

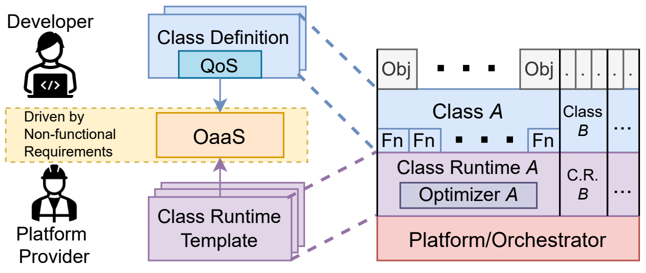


Figure 5: Realizing objects with class runtime and template: OaaS maintains templates customized for various deployment scenarios. For a specific class, Oparaca uses one of its predefined templates to create a class runtime to manage the deployed classes optimally.

Within each class, we can define *methods* and *attributes* to encapsulate the application logic and state (that can be in the form of structured or unstructured data, i.e., BLOB), respectively. For structured state data, Oparaca allows the developer to keep the data as a JSON-based document, similar to the document database [18]. For unstructured data, however, object storage is employed to store them. We model each method as a serverless function[1]. Oparaca shares object states among methods of the same object following the OOP encapsulation principles.

In Oparaca, application QoS and constraints are declared through the *non-functional requirement interface*. The interface allows the developer to associate a class or its methods with one or a set of requirements that the cloud provider has to meet once objects of the assigned class or methods are deployed successfully. Table 1 shows the list of QoS and constraints currently supported by Oparaca. Non-functional requirement declarations are treated as properties of classes or methods, so they are enforced according to the OOP inheritance principles. If a method and its class have conflicting requirements, then the method-level requirement prevails.

Figure 4 shows the class diagram of an example application providing image processing functionalities, such as resizing and changing the format. A developer can translate

[1]we use the term function and method interchangeably in this paper

the diagram directly to OaaS classes. Specifically, OaaS allows images to be wrapped inside the `Image` class abstract where the image itself can be defined as a single unstructured file and its metadata is structured data. The `resize` function receives width and height as its inputs and produces a new image object as its output. The `changeFormat` function receives the new format name as input and produces a new image as the output object. The developer can add a new class `LabelledImage` for the image that can have the label data of image content. This class extends the `Image` class with the additional `labels` data and `analyze` function. The `Image` class also has a `detectObject` function to perform object detection to create the `labels` data and create the `LabelledImage` object as an output. The `analyze` function is to perform further analysis to label data. Oparaca currently supports the OaaS Abstraction Interface in `YAML` format. The class declaration of the example is in Listing 1.

Based on inheritance, in this example, the `LabelledImage` class inherits the non-function parameters from `Image` class (i.e., availability=99.9). The `resize` and `changeFormat` functions that the class `LabelledImage` inherit also maintain the non-functional parameter from class `Image`.

4.3 Object Realization

4.3.1 Class Runtime and Template. Oparaca uses *class runtime* to deploy and manage objects derived from user-defined classes (Figure 5). To meet the third requirement (i.e., efficiency), the class runtime must be optimized to fulfill the non-functional requirements within a reasonable cost and overhead. However, given the non-functional requirements that Oparaca supports, there is a vast diversity of possible non-functional requirement combinations that need different specializations to satisfy. Thus, it is impractical to have a single design for the class runtime that can efficiently satisfy all of the requirements.

To resolve the problem, Oparaca introduces *class runtime template*, which provides a configurable class runtime design optimized for a specific set of requirement combinations. Oparaca maintains a list of different templates to support as many requirement combinations as possible. When deploying a class, Oparaca will choose from the list the most suitable template to realize the class requirement and then follow the template design to create a dedicated class runtime for this class. This approach allows Oparaca to satisfy both portability and efficiency design requirements.

In terms of portability, the class runtime template enables Oparaca to have freedom and flexibility in realizing objects. Instead of seeking a one-size-fits-all object realization mechanism, Oparaca decomposes the object realization into a set of sub-problems, each one aiming to find the optimal solution (i.e., class runtime template) for a specific infrastructure setting and requirement combinations. The approach makes Oparaca's implementation modular and flexible. One can upgrade existing solutions, extend the implementation to include new non-functional requirements, or even adjust for new infrastructure by adding/modifying templates without worrying about compatibility issues.

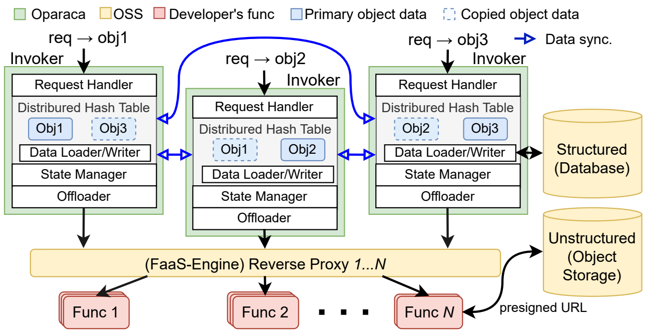


Figure 6: LTAG (Latency, Throughput, and Availability Guarantee): An example of a class runtime template designed for enforcing class latency, throughput, and availability requirements (OSS: Open-source software).

In terms of efficiency, Oparaca can use off-the-shelf solutions to implement its class runtime templates. This allows Oparaca to take advantage of a vast diversity of existing state-of-the-art solutions, which have been proven to be efficient in practice, to reliably enforce non-functional requirements at minimum time, cost, and effort. Further, since class runtime templates are configurable, depending on specific object deployment scenarios, the class runtime derived from the template can be customized for further efficiency. Oparaca also allows platform provider to customize the template configurations, selection conditions, and priority for their operation objective (e.g., resource utilization).

4.3.2 Class Runtime Example. Figure 6 shows LTAG (Latency, Throughput, and Availability Guarantee)—a class runtime template that Oparaca currently uses to enforce class latency, throughput, and availability requirements. Each class runtime derived from the template has three modules: *invoker*, *FaaS engine*, and *data storages*. The invoker is responsible for handling all of the object-related operations. For each operation, the invoker finds its corresponding function and offloads the operation to that function managed by the FaaS engine. LTAG can maintain the object state in both unstructured and structured databases.

In the offloading mechanism, the invoker utilizes the pure function approach that bundles the invocation request and the object attributes as a standalone task within a FaaS engine. Each invocation takes the object attributes as input, modifies them, and then returns the updated attributes as the output to the invoker. The invoker maintains an internal in-memory distributed hash table (DHT) [34] to keep the object data (i.e., attributes and metadata) for reducing database access operation, thereby speeding up the object invocation.

Throughput Enforcement. OaaS currently supports throughput enforcement by allowing applications to specify a guaranteed invocation rate A per FaaS function [73]. Oparaca ensures that sufficient resources are available so that at least one invocation can start immediately (i.e., without cold-start delays) every $\frac{1}{A}$ seconds. LTAG customizes the Invokers and FaaS engine based on Real-time Serverless [71, 73] to estimate and periodically adjust resource allocation for each class and its functions, ensuring they can handle operation requests up to the specified rate guarantee.

Latency Enforcement. Recent work on latency QoS aims to minimize end-to-end latency in a best-effort manner [44, 52, 57, 96, 98], giving no guarantee to construct/realize non-functional requirements. Besides, other efforts try to keep latency within a specific target deadline [8, 13, 67, 88, 90], but this is extremely difficult from the cloud provider's perspective due to the highly dynamic and unpredictable nature of invocation logic [28, 46, 82], data size [13, 27, 70], and communication requirements [94]. Thus, to enforce the latency in a feasible and controllable way, OaaS offers guarantees to minimize the system overhead of invocation executions, focusing on cold-start and communication, enabling the developers to optimize their functionality execution time barely based on improving their codes. The developer can address cold-start via throughput enforcement, as described above. For communication, OaaS provides a *locality* guarantee, allowing developers to specify the location for invocation dispatch. This can be either (i) *local*: attributes are read and written as if they are in the same FaaS container executing the function logic, and (ii) *none*: no locality restriction.

LTAG enforces the *local* guarantee by exploiting the class function-attribute relationships. Specifically, Oparaca uses consistent hashing, maintained by invokers, to track object data locations and route invocation requests to the corresponding place.

Availability Enforcement. OaaS provides availability enforcement as a reliability guarantee, defining the percentage of time that an object (or its methods) are available for invocation execution. LTAG enforces availability through replication. Specifically, given an object with availability requirement A (e.g., 99.99%), we enforce A by creating N replicas of the object with N is defined according to Meroufel and Belalem [64] as follows.

$$N = 1 - (1 - P)^A \qquad (1)$$

where P is the stability of the resources used to deploy the object. LTAG replicates the object data and uses the DHT to manage them. However, it keeps only one object replica, called *primary*, active at a time. To enforce consistency, the primary object handles all state modifications and then commits the results across all replicas. If the primary replica fails, Oparaca chooses one of the remaining replicas as the new primary.

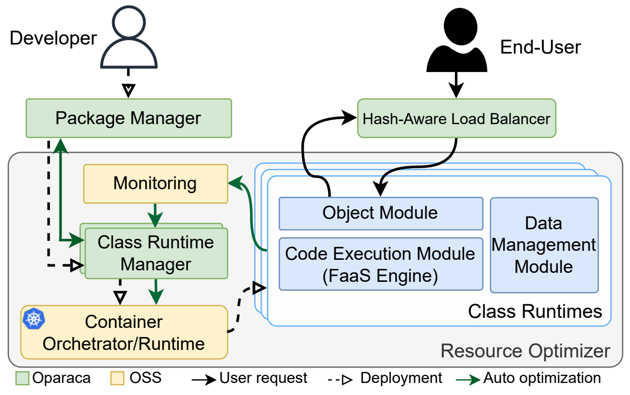


Figure 7: A bird-eye view of Oparaca's architecture

4.4 Oparaca Architecture

Oparaca's architecture, shown in Figure 7, includes the following key components: *(1) Package Manager*: responsible for managing classes registered in Operaca and their corresponding deployment packages. This component also acts as a gateway and offers APIs to develop and deploy OaaS-based applications. *(2) Class Runtime*: turns the class descriptions and corresponding packages into the actual object deployments on the cloud. *(3) Class Runtime Manager*: create dynamic class runtime from existing templates (e.g., LTAG). It is also responsible for class runtime deployment and management. *(4) Monitoring System*: gathers the performance metrics from class runtime. *(5) Hash-aware Load Balancer* and *Container Runtime*: responsible for scheduling and managing function execution. Once a function invocation is issued, the hash-aware load balancer routes the request to the corresponding class runtime by using consistent hashing that, in turn, forwards the request to the corresponding container for execution.

Given the interface and architecture, the application lifetime on the cloud now consists of two phases:

(a) Registration: The developer registers their class to Oparaca. Upon registration, the *package manager* unpacks the deployment, extracting the class logic (e.g., functions), state (e.g., data schema), and non-functional requirements (e.g., QoS and constraints). The extracted information is then forwarded to the *class runtime manager* to find an appropriate

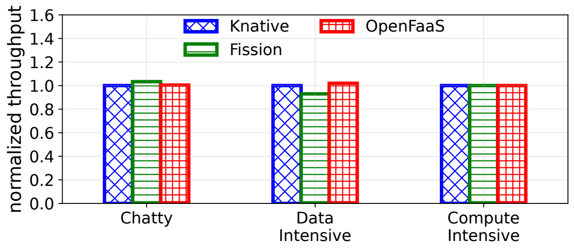


Figure 8: Oparaca does not significantly differ in throughput performance across the FaaS engines.

class runtime template to generate a dedicated *class runtime* to handle the object realization for the class.

(b) Execution: Once a *class runtime* is created, it is responsible for managing the execution and state of all objects generated from the associated class. Every interaction with the application users is handled through the class runtime, independent from other Oparaca components. To ensure reliability, the *class runtime manager* periodically collects monitoring metrics from class runtime. Based on the information, Oparaca can adjust the *Container Orchestrator/Runtime* to improve efficiency and take administrative actions (e.g., to recover from failure, etc.) if needed.

Note that the above procedures are performed solely by Oparaca platform. Application developers do not have to intervene or refine their configuration for both functional and non-functional requirements. This greatly simplifies application deployment.

5 EVALUATION

In this section, we seek to learn the performance of Oparaca in the following aspects: non-functional requirement enforcement (Section 5.2.1), implementation efficiency (Section 5.2.2), deployment productivity (Section 5.2.3), and development productivity (Section 5.2.4).

5.1 Experimental Setup

We prepare the experimental environment on 4 machines on Chameleon Cloud [47], each with 2 sockets of Intel(R) Xeon(R) Gold 6240R CPU processors that collectively have 192 cores, 768 GB memory, and SSD SATA storage. We use 3 machines to install the Kubernetes cluster (RKE2 [40]) for deploying applications. The last machine generates load using Gatling [22]. Regarding data management, we use Minio [39] (S3-compatible storage) for unstructured data and ArangoDB [36] (document database) for structured data.

Workloads. To make sure our evaluation is comprehensive, we consider the following three classes of applications that exhibit different behaviors:

- *Chatty*: characterized by frequent small communications that impose significant overhead on network transmission [65]. As a representative workload for the application class, we utilize *JSON randomization* [60], which involves a sequence of ten invocation requests, each randomly updates a JSON key-value pair to the document database.
- *Data Intensive*: characterized by substantial data access operations [35]. We use an *image resizing* workload [9, 85], which resizes images stored in object storage through FaaS invocations, to represent this class of applications
- *Compute Intensive*: demand extensive computational resources throughout their lifecycle (e.g., ML [19] and HPC [72] applications). To represent this class, we use *video transcoding* [68, 93], which involves changing the resolution of a video file stored in object storage.

Approaches. To ensure generality, we integrated Oparaca with various FaaS engines—Knative [31], Fission [75], and OpenFaaS [29], all backed by Kubernetes—to host object functions. Figure 8 shows the maximum throughput achieved by workloads mentioned above when deployed over Oparaca using these different FaaS backends under identical resource configurations (each deployment can scale up to five Kubernetes pods, each with 4 CPUs). The throughputs, normalized to Knative, are nearly equivalent across all FaaS engines for all three workloads. This confirms that Oparaca can be configured to work with various FaaS engines with negligible performance differences, making it flexible for deployment across different cloud environments. Thus, due to space limits, we report only the experimental results for Oparaca's Knative variant. Also, for fair comparison, we use Knative with various deployment configurations as experiment baselines:

- *Knative:* Default Knative configuration that declares only per-container resource requirements (i.e., CPU and memory) and leaves the rest to the auto-scaling system.
- *Knative-con:* Default Knative configurations plus applying per-container concurrency limit to avoid overloading.
- *Knative-rts:* adopt Real-time Serverless resource management [73] to enforce throughput guarantee.
- *Oprc* is Oparaca, which allows the applications to enforce their throughput, latency, and availability in their class definitions. Since Oparaca needs to learn the workload metrics before properly optimizing the class runtime, we perform one more extra round of load generating in each experiment. The first round acts as the warm-up for Oparaca to properly gather the metrics.

Beyond ensuring a fair comparison, we choose Knative as a FaaS baseline because it offers a rich set of configuration options to capture diverse deployment scenarios often unsupported by other engines. Additionally, varying Knative settings demonstrate how current FaaS implementations address

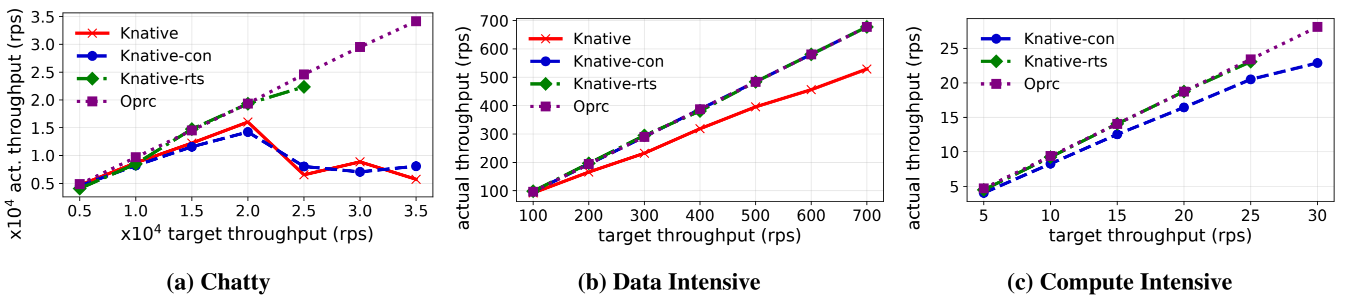


Figure 9: Achievable throughput varying target throughput. Oparaca ensures the actual throughput matches the target one across settings, while the other approaches fail to do so at high throughput targets.

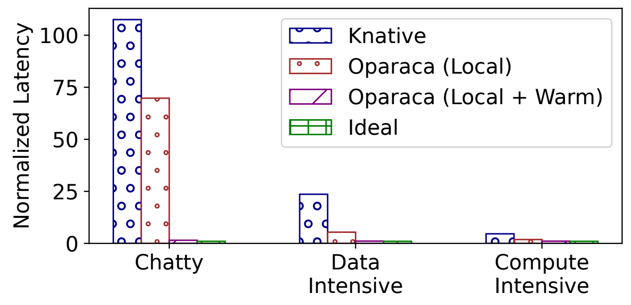


Figure 10: Oparaca can exploit data locality to provide various latency guarantees.

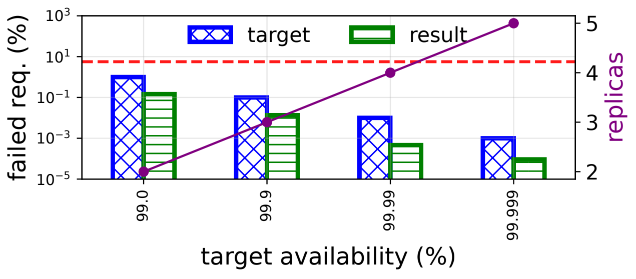


Figure 11: Successful invocation rate at different availability targets with availability enforcement. Resource stability (P) is 94.36% (red line).

non-functional requirements—by adjusting low-level configurations (e.g., per-container concurrency) in a best-effort manner. Configuring Knative allows us to explore a broad range of FaaS deployment configurations, whether these adjustments are made by developers (if the FaaS engine exposes the configurations) or by cloud providers (if it does not—for example, Microsoft Azure doesn't allow developers to configure per-container concurrency). Thus, although our evaluation results are specific to Knative, the insights and implications are generalizable to other FaaS engines.

In the following experiments, Oparaca deploys and manages workloads using class runtime derived from the LTAG template. Thus, data access is automated via the invoker. In the Knative variants, however, these applications have to implement direct data access to storage or database manually.

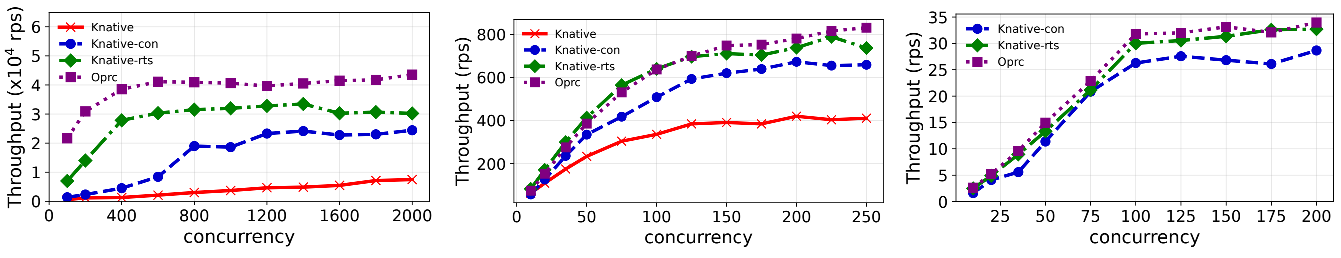
5.2 Experimental Results

5.2.1 Non-functional Requirement Enforcement. We validate the QoS enforcement capability of Oparaca by deploying applications mentioned in Section 5.1 using the LTAG class runtime template as described in Section 4.3.2.

Throughput. To validate Oparaca's throughput enforcement, we deployed the three applications with various target throughputs. Then, we configured the load generator to send the request at the same rate as the target throughput and measured the actual throughput on each system. The results are reported in Figure 9.

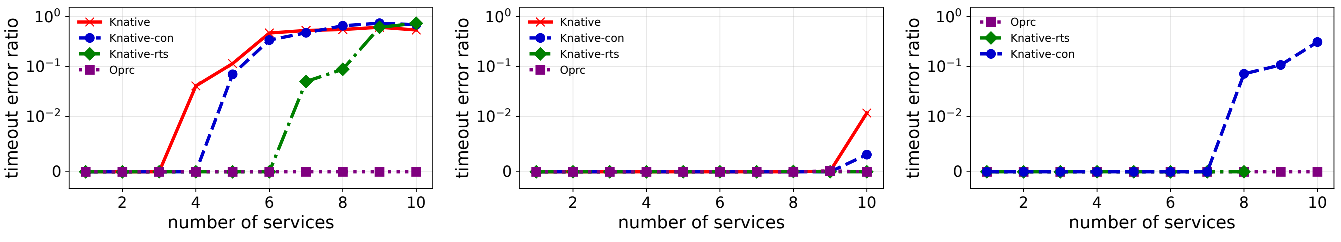
Overall, Oparaca can guarantee the throughput for all three applications. *Knative-rts* only meets low throughput targets and fails at higher ones due to over-provisioning. The other two *Knative* variances fail to meet the targets since they only rely on auto-scaling without the awareness of the target throughput. In the chatty workload, with the high request arrival rate, the internal queue cannot hold requests long enough to wait for the new pod to be spawned. Meanwhile, in the compute-intensive application, it takes longer for each request to be processed, making it easier to time out. Only the data-intensive application that *Knative-con* can meet the target throughput.

The results also demonstrate the complexity of FaaS configuration. Even when utilizing the same backend services (i.e., Knative), varying FaaS deployment configurations result in significantly different performance outcomes. Thus, manual adjustment of FaaS deployment, while daunting, is often required to achieve the desired throughput. In contrast, Oparaca simplifies and automates this process with its high-level interface.



(a) Chatty **(b) Data Intensive** **(c) Compute Intensive**

Figure 12: Achievable throughput under various request concurrency. Concurrency is defined as the number of clients that concurrently generate requests for the system.



(a) Chatty **(b) Data Intensive** **(c) Compute Intensive**

Figure 13: Error response ratio of different solutions upon deploying them with the different number of services.

Latency. We deployed all three applications over Operaca under the *locality* and *throughput* guarantee. We let the applications run under bursty loads by configuring the load generator to remain idle most of the time but occasionally create sudden bursts that send requests at a rate equal to the application throughput guarantee for a very short duration. We compare Oparaca against two baselines: (i) *Knative* with the data storage deployed at a separate data center from the FaaS deployment, representing a typical scenario of FaaS deployment [77], and (ii) *Ideal* where functions and data storage are deployed together on a dedicated machine with excessive resources, representing an ideal execution environment where the invocation execution latency depends solely on the application itself.

Figure 10 shows the average execution time of the three applications across different deployments. The latency is normalized to the case of the ideal deployment. Knative is the worst among approaches, with the latency can be as high as 60× the ideal. The reason is two-fold. First, Knative needs external storage to keep the application data, but the actual data location is hidden under the storage abstraction, causing significant data transmission latency. Oparaca does not have this limitation as it encapsulates the data and invocations under a unified object abstraction, enabling locality enforcement, i.e., *Oparaca (Local)*, that allows invocations to execute at the same machine with their data, significantly reducing the latency by 1.5× (Chatty) to 4× (Data Intensive). Second, Knative scales resources allocated to FaaS functions based on concurrency. That makes invocations suffer from cold-start under bursty loads. Applications can workaround this issue with Oparaca via throughput guarantee, enforcing the cloud to execute invocations without cold-start up to a certain rate, i.e., *Oparaca (Local + Warm)*. This configuration further reduces the latency by 1.7× (Compute Intensive) to 46.5× (Chatty)! Enforcing these two non-functional requirements together allows applications deployed over Oparaca to minimize their invocation overhead (as low as 7% of execution time), achieving invocation execution latency that is very close to the ideal execution.

Availability. Next, we validate Oparaca's availability enforcement. We have created a failure emulator that injects failures by deleting the platform container according to a predefined Mean Time Between Failures (MTBF). Whenever a failure is injected, Kubernetes automatically recovers the container. The emulator then waits for MTBF, which is also supplemented by a random value from a normal distribution, before introducing the next failure. The emulator carries out these operations on each container individually. To select the MTBF, we use the reference MTBF of the Intel server boards [37] that have around 50K hours on average. To speed up the experiment process, we scaled this number down by a million, setting the MTBF to 180s, which makes each container only operate for 94.36% of the time. We then use 94.36% as the resource stability (P) to configure Oparaca. We deploy the application according to the different target availability, generate the load to test the actual application availability with a rate of 200 requests per second for 1.5 hours, and measure the ratio of the requests being processed unsuccessfully.

The results of this experiment are reported in Figure 11. When availability enforcement is on, Oparara deploys classes and objects with replications, significantly reducing the failure rate to meet the availability targets. The actual failed request ratio is slightly lower than each predefined target because Oparaca adds just enough replicas to meet the target, minimizing availability enforcement overhead. Notably, increasing the availability from 99% to an exceptional rate of 99.999% (1000× better) incurs only 2.5× extra resource cost. This is a 50× improvement versus the current industry standard that necessitates an SLA on availability of 99.95% [2] with only 1.67× cost increment.

Takeaway: *Unlike traditional FaaS deployments, Oparaca can automatically reconfigure to enforce various non-functional requirements for different classes of applications, eliminating the need for manual refinement.*

5.2.2 *Efficiency of Oparaca.*

In this subsection, we examine Oparaca efficiency, running various experiments on a fixed quantity of resources to see how well the implementation handles various workloads under different operation scenarios.

Function Invocation Efficiency. To evaluate Oparaca invocation efficiency, we compare its maximum throughput with Knative variants; all are under limited resources. The throughput measurement takes multiple runs with an increasing number of clients (i.e., concurrency). We measure the mean throughput achieved in each run and report them in Figure 12.

In general, the throughput becomes steady after increasing the concurrency to a certain level. Oparaca provides a higher throughput compared to other baselines, especially for the chatty workload (Figure 12a) because Oparaca relies on the internal in-memory distributed hash table (DHT) to store the object data; thereby, it speeds up the data access and reduces the database operation. For the chatty workload, *Knative-con* and *Knative* yield significantly lower throughput compared to *Knative-rts*. This is because this workload performs little computation compared to its network I/O operation, which makes the Knative auto-scaler inaccurately adapt the acquired resources to the workload.

For the data-intensive workload (Figure 12b), *Knative* performs poorly because the auto-scaler cannot accurately adjust acquired resources to the increasing workload without per-container concurrency declaration. In contrast, by only declaring per-container concurrency, *Knative-con* can perform with a little less performance than *Knative-rts*.

For the compute-intensive workload (Figure 12c), because it is computationally intensive and the invocation rate is also less than the other workloads, all of the solutions can provide similar performance. Only *Knative* cannot be used for this workload because without controlling the concurrency, each function container has to handle more concurrent invocations than it can. As a result, they fail to handle requests continually. Oparaca can perform slightly better than the others because it eliminates the need to fetch and deserialize the record (i.e., metadata) from the database on each function container.

Throughput Enforcement Efficiency. Our primary objective in this experiment is to examine the resource efficiency of Oparaca against other baselines and ensure its throughput is not attained with the cost of lavishly allocating resources. The other objective is to investigate Oparaca's behavior in the face of services with different throughput expectations. To that end, we configure multiple services of the same type, each with its own target throughput. To achieve this, we started by testing on a single service and gradually increasing the number of services to ten. We set the target throughput of each replicated service to be 1/10th of the maximum throughput we found in the previous experiment. We chose these numbers so that the target throughput is not too low and scaling remains relevant. The experiment is performed by generating invocation requests to each service, with the request rate capped to the target throughput, and then measuring the ratio of the number of timeout errors to the total number of requests.

As shown in Figures 13, overall, Oparaca outperforms other baselines for almost all workloads. For the chatty workload (Figure 13a), Oparaca can handle all of the requests with zero error rate because of its ability to readjust its allocated resources and its internal DHT structure. *Knative-rts* also performs well at the beginning; however, after 6 services, the external document database starts to slow down, leading to a sharp increase in the error rate. The poorer performance of *Knative* and *Knative-con* is mainly because their independent scaling of services and lack of awareness of performance objectives lead to resource contention among co-existing services.

For the data-intensive workload (Figure 13b), all baselines are capable of handling requests up to 9 services. Nonetheless, for 10 services, only *Knative-rts* and Oparaca remain error-free. *Knative-con* and *Knative* still suffer from the resource contention. Similarly, for compute-intensive workload (Figure 13c), *Knative-rts* and *Knative-con* only have enough resources to meet the target throughput up to 8 and 7 services without any error, respectively. Oparaca, however, can handle all of the requests for up to 10 services.

Takeaway: *Being cognizant of performance objectives is crucial for Oparaca to deliver competitive efficiency for both the user and the system across different applications while also offering a high-level abstraction to the user.*

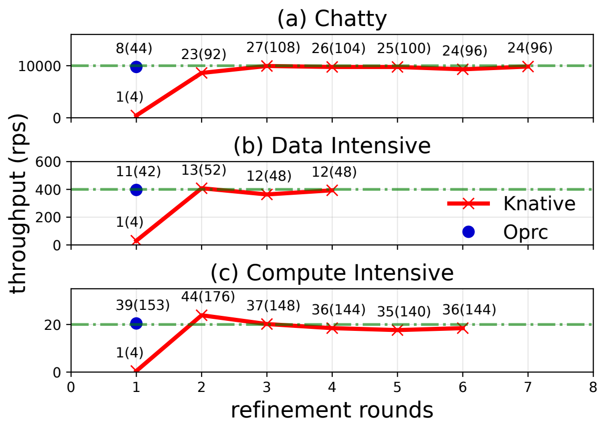


Figure 14: Rounds of refinement for Knative to enforce the target throughput (green lines) versus Oparaca. Data points are annotated by `#pods (#cores)`, including invoker pods.

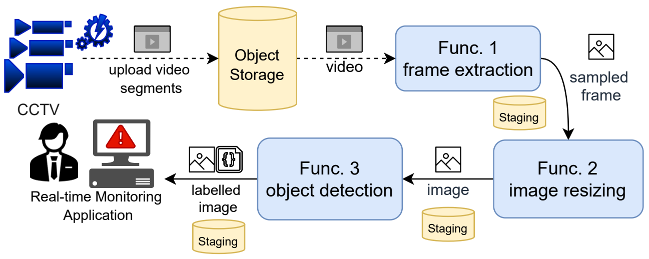


Figure 15: The case study of developing video and image processing for a real-time monitoring system

5.2.3 Deployment Productivity Using Oparaca. To show the productivity improvement of Serverless application deployment, we present the experiment on the refinement steps using Knative on three application deployments with the requirement to enforce the throughput of 10k, 400, and 20 requests per second for chatty, data-intensive, and compute-intensive, respectively. The manual refinement strategy consists of three phases. First, we want to find the number of pods that roughly provide throughput that is equal to our objective. We deploy the application with a single pod and then perform load testing to find the throughput. Then, we scale it up using the formula below and repeat this process until the throughput matches the objective.

$$pods_{next} = \frac{throughput_{target}}{throughput_{current}} \times pods_{current}$$

The second phase reduces the pods until they cannot satisfy the target. The last phase increases the container-level concurrency but reduces the number of pods to improve utilization.

As shown in Figure 14, the manual refinement method needs at least 4 rounds to find the optimal number of pods to meet the target throughput, while we only need to give the Oparaca the number, and it will automatically adjust the deployment when we feed the load. Furthermore, Oparaca improves application performance while reducing the required resource allocation to meet the target throughput. For IO-intensive workloads focused on structured data like the chatty workload, Oparaca reduces resource usage from 100 cores to 44 cores. This is because OaaS unlocks cross-domain optimization—in this case, data locality—to speed up invocation execution time, quickly freeing up FaaS pods for higher concurrency and significantly reducing resource requirements compared to Knative. Even for the compute-intensive application, where locality is not an issue, Oparaca automatic refinement still achieves the throughput target at a comparable cost (153 cores) versus Knative (144 cores, only 6% higher), which requires much more effort in manual tuning (6 rounds of refinement versus one).

Takeaway: *Oparaca's OaaS abstraction improves deployment productivity and performance enforcement effectiveness.*

5.2.4 Development Productivity Using Oparaca. In this part, we provide two cloud application developments representing common cloud applications at different scales, non-functional requirements, and complexities. We will deploy these applications using the OaaS paradigm and recommended FaaS deployment practices to demonstrate how OaaS can make the development of cloud-native serverless applications more productive.

Case Study # 1. Real-time Monitoring System. Figure 15 shows a CCTV system uploading video segments to object storage, waiting to be processed by a workflow of function that includes `extractFrame()` that splits a video segment into multiple frames; `resizeImg()` whose job is to resize the image frame to be usable by the next function in the pipeline; and `detectObject()` is in charge of performing the object detection on an image and generating label in the `JSON` format. These functions must persist their output data so that the following function in the workflow can consume it. Because the entire workflow is latency sensitive, the execution rate of the whole workflow (i.e., throughput) has to be guaranteed. Developers can calculate the throughput by the number of cameras and the object detection frequency.

FaaS implementation. The developer must repeat the following steps for each function deployment: (i) Configuring cloud-based object storage, database and maintaining the credential access token for the functions to use. (ii) Implementing the functions' business logic. (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 must connect them as a workflow via a function orchestrator service (e.g., AWS Step Functions [4]). Finally, upon arrival of a new video segment, the event triggers the workflow to put the result into the database, waiting to be processed by the monitoring system. To ensure the target throughput, developers have to go through multiple rounds of testing and refinement to get the final configuration for each function.

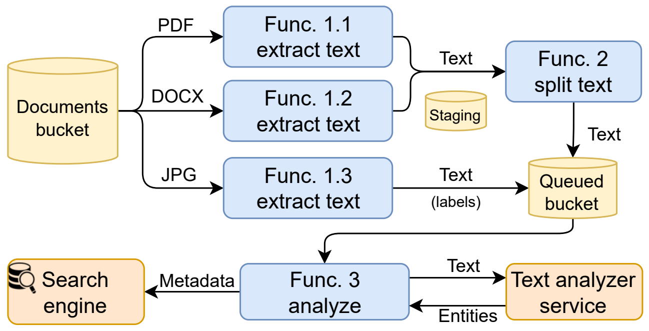
OaaS implementation. The developer defines three classes:

- **`Video`** class with `extractFrame()` function that produces `LabeledImage` as the output, and `wfDetectObject(freq)` workflow function that has a detection frequency as the input. This class also has `video` file as an unstructured state.
- **`Image`** class contains `resize` function and `image` file as an unstructured state (see Listing 1).
- **`LabeledImage`** class inherits from the `Image` class and has its own `objectDetection()` function and `labels` data (state) in JSON format (see Listing 1).

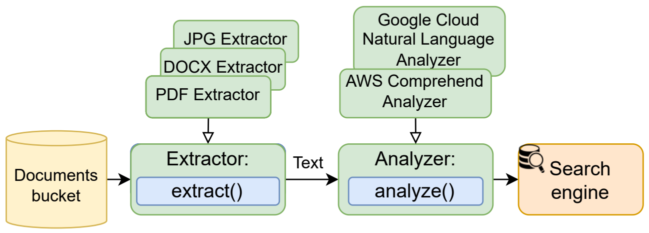
Upon uploading a new video to the Oparaca platform by the CCTV system, it creates a "video" object and invokes `video.wfDetectObject(freq)` that outputs a `LabeledImage` object that is consumed by the real-time monitoring application. We note that, in developing the class functions, the developer does not need to be involved in the data locating and authentication steps. To ensure the application performance, developers only need to declare the target throughput within the class definition (see example in Listing 1); then, the Oparaca can transparently create the suitable class runtimes and their configuration.

Case Study # 2. Searchable Document Repository. Retrieving and processing at scale the vast repositories of valuable documents, images, and media from enterprise customers is a common practice in the cloud [72, 92]. In this case study, we first present how the application is deployed with traditional FaaS on the cloud, the limitations of this approach, and how to resolve them with OaaS/Oparaca.

FaaS implementation. Figure 16 shows the serverless workflow to analyze the document in various formats and update the metadata to the search engine recommended by AWS [12]. Upon the document uploads to the document bucket (object storage), the storage triggers the event to invoke `extractText()` based on the type of the document. If the document is in `PDF` or `DOCX` format, the function extracts the text and sends the text to be split by the next function `splitText()`. The



(a) FaaS-based [12]



(b) OaaS-based

Figure 16: The searchable enterprise document repository implemented based on FaaS and OaaS paradigm.

result will be put into the `Queued bucket`. Alternatively, if the document is in `JPG` format, the `extractText()` function analyzes the image to get labels and puts them in the `Queued bucket`. In the next step, the `analyze()` function loads text from the `Queued bucket` to analyze it via the external text analyzer service (e.g., AWS Comprehend) and then saves the metadata result to the search engine.

The FaaS implementation has two main drawbacks. First, developers must explicitly manage application state and data using separate storage services, which increases complexity and makes it difficult to configure non-functional requirements as in the previous case study. Second, functionalities may require numerous and heterogeneous FaaS deployments—for example, needing separate extraction functions for each document type, where some (like `PDF` and `DOCX`) require staging and others (like `JPG`) do not. 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 text analyzer services instead of one).

OaaS implementation. To demonstrate the feasibility of OaaS in production, we transform the given FaaS-based solution into OaaS with minimal effort to resolve the previously mentioned drawbacks. The transformation involves three steps.

- **Workflow Construction.** We encapsulate related FaaS functions, states, and key data into objects representing two key functionalities: **`Extractor`** to extract text from the document repository and **`Analyzer`** to analyze the extracted text. The two classes form the critical path of the application processing pipeline, as shown in Figure 16b.

- **Object Encapsulation.** We apply inheritance and polymorphism to promote software reuse by wrapping corresponding FaaS functions and states into classes derived from the two base classes. This approach hides the need for storage services behind the object abstraction and outsources their implementation to the cloud. It also simplifies development, as developers only need to construct the processing pipeline once in the base class definitions and then focus on implementing functionalities for specific cases with their derived classes, avoiding repetitive pipeline construction and implementation whenever a new document type or analyzer service is added.
- **Integration of Non-Functional Requirements.** Developers integrate appropriate non-functional requirements into the corresponding objects to meet application needs for performance, availability, and cost. With Oparaca, non-functionality requirement enforcement, as shown in previous experiments, is achieved without any additional refinement effort from the developers.

Takeaway: *Oparaca accelerates development by abstracting low-level infrastructure concerns and automating runtime configurations through a high-level interface.*

6 RELATED WORKS

6.1 Compute-Data Encapsulation

Combining data and compute abstraction is an active research direction to deploy stateful applications with FaaS productively. We can classify studies on this front based on how the function can access the data.

Unified compute-data abstraction. Many serverless platforms are designed to combine one or more functions and state data into unified deployment units such as "actor" [86] or proclets [79]. Functions and state data are co-located when executed so that functions can access the state data in local memory. Azure Entity Functions [66] that is based on the concept of virtual actor, Orleans [16]. Kalix [38] uses CRDT [83] to replicate the state among functions. Similar to OaaS, our prior works [53–55] and Nubes [63] also rely on the object-oriented concepts to encapsulate the function and data into unified deployments.

Datastore abstraction. The serverless platform provides a datastore API to the function for storing the state. Cloudburst [87] offers stateful functions using a shared distributed key-value database. FAASM [84] allows the function access to the shared memory via WASM. Crucial [10] allows a function to access the shared data via the DSO layer (distributed hash table). Boki [42] enables stateful functions by providing API access to the distributed logging system. Beldi [95], on the other hand, provides the database and transaction API to the state. YuanRong [20] offers a unified interface for the function to access the external database. Shredder [97] and Apiary [49] enable the function to be executed within storage/database service in a stored-procedure manner. Kalix [38] and Apache Flink Stateful Function (StateFun) [7] proactively package the state within the invocation request payload and expect the modified state to be returned as part of the response payload.

Existing works, despite their diversity, focus mainly on data and compute encapsulation to enhance programmability and productivity, often neglecting non-functional requirements like performance and availability. OaaS fills this gap by introducing the new non-functional requirements interface and enforcing them by leveraging enriched information from the encapsulation with state-of-the-art solutions, as presented below.

6.2 Non-functional Requirements Enforcement

There is a rich body of research has been carried out to improve serverless execution latency. Most of them address the well-known cold-start problem [26, 82, 91], which applications cannot easily resolve on their own. Noticeable approaches include mitigating cold-start penalty [25, 43, 59] and sandbox recycling [32, 82]. Other efforts in the area focus on strengthening performance isolation [1, 62] and proper resource allocation [13, 14] to keep invocation executing at the desired speed. Commercial cloud providers let applications manually configure for throughput through pre-allocation [6], but this can be costly if the actual FaaS resource demand does not meet load estimation. Real-time Serverless [73] resolves the problem by allowing applications to dynamically scale to actual use under a predefined guaranteed invocation rate.

Enforcing the non-functional requirements becomes more complicated as applications evolve and become bigger. Many dedicated studies are addressing different aspects of the problem. Sequoia [89] proposes a new QoS function scheduling and allocation framework. Real-time Serverless [71, 73] extends the FaaS model to enable performance engineering through configurable guaranteed invocation rates. Aquatope [99] proposes a QoS-and-uncertainty-aware resource scheduler for end-to-end serverless workflows. Astrea [41] proposes an autonomous system that configures and orchestrates serverless analytic jobs. Pheromone [94] replaces the traditional invocation-based workflow orchestration with a data-centric approach to enable locality exploitation across workflow execution.

Despite their significant benefits, the mentioned approaches address only limited aspects of FaaS applications. Furthermore, most rely on best-effort methods due to limited abstraction integration and cloud-developer coordination in current FaaS implementations and programming models (as presented in Section 2). This limits their practicality, as real-world applications often have multiple objectives and constraints [72, 73]. In contrast, OaaS's non-functional requirements API enables the enforcement of multiple objectives only through declaration with minimal refinement effort, allowing for simple and reliable application deployment.

7 CONCLUSION

In this paper, we introduced the Object-as-a-Service (OaaS) paradigm that offers a new cloud service abstraction that borrows principles of object-oriented programming to encapsulate application logic, data, and non-functional requirements into a unified deployment package. The approach not only greatly simplifies native-cloud application development, but also enables requirements-driven cloud-developer coordination that opens the gate for many performance optimization opportunities. Moreover, OaaS relieves developers from the complexity of application fine-tuning to meet the desired QoS and deployment constraints. We also developed a prototype OaaS platform called Oparaca and evaluated it across various real-world applications and scenarios. The evaluation shows that Oparaca can enforce various application QoS with comparable resource efficiency versus other cutting-edge approaches while significantly reducing the time and effort required for cloud-native application deployment and development.

In the future, we plan to enhance Oparaca to support additional non-functional requirements (e.g., those listed in Table 1). We will also expand its object configuration to give developers more flexibility in choosing data storage, execution, and orchestration implementations. This work serves as a starting point for several promising research directions. For instance, can Oparaca be extended across multiple data centers to leverage its high-level abstractions and non-functional requirement enforcement for addressing challenges in distributed systems, such as resilience and heterogeneity?

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their constructive feedback; and Chameleon Cloud for providing resources. This project is supported by National Science Foundation (NSF) through CNS CAREER Award# 2419588.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[2] Amazon. 2022. AWS Lambda Service Level Agreement. https://aws.amazon.com/lambda/sla. Online; Accessed on 15 July 2024.

[3] Amazon. 2024. Amazon Relational Database Services. https://aws.amazon.com/rds/. Online; Accessed on 1 Apr. 2024.

[4] Amazon. 2024. AWS Step Functions | Serverless Microservice Orchestration. https://aws.amazon.com/step-functions. Accessed on 23 Jul. 2022.

[5] Amazon. 2024. Cloud Object Storage | Amazon S3 – Amazon Web Services. https://aws.amazon.com/s3/. Online; Accessed on 12 Nov. 2023.

[6] Amazon. 2024. Configuring provisioned concurrency for a function. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html. Online; Accessed on 7 July 2024.

[7] Apache. 2024. Apache Flink Stateful Functions. https://nightlies.apache.org/flink/flink-statefun-docs-stable. Online; Accessed on 15 Jul. 2024.

[8] Onur Ascigil, Argyrios G Tasiopoulos, Truong Khoa Phan, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. 2021. Resource provisioning and allocation in function-as-a-service edge-clouds. *IEEE Transactions on Services Computing* 15, 4 (2021), 2410–2424.

[9] David Balla, Markosz Maliosz, and Csaba Simon. 2021. Estimating function completion time distribution in open source FaaS. In *Proceedings of the 10th IEEE International Conference on Cloud Networking (CloudNet)*. IEEE, 65–71.

[10] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, 41–54.

[11] Salman A Baset, Long Wang, and Chunqiang Tang. 2012. Towards an understanding of oversubscription in cloud. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*.

[12] James Beswick. 2021. Creating a searchable enterprise document repository. https://aws.amazon.com/blogs/compute/creating-a-searchable-enterprise-document-repository/. Online; Accessed on 12 Oct 2024.

[13] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*. 257–272.

[14] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the 18th European Conference on Computer Systems*. 381–397.

[15] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1591–1604.

[16] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 1–14.

[17] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing 2019*. 13–24.

[18] Inês Carvalho, Filipe Sá, and Jorge Bernardino. 2022. NoSQL Document Databases Assessment: Couchbase, CouchDB, and MongoDB. In *Proceedings of the 11th International Conference on Data Science,*

Technology and Applications - Volume 1: DATA,. INSTICC, SciTePress, 557–564. https://doi.org/10.5220/0011352700003269

[19] Dheeraj Chahal, Ravi Ojha, Manju Ramesh, and Rekha Singhal. 2020. Migrating large deep learning models to serverless architecture. In *Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 111–116.

[20] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, et al. 2024. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 843–859.

[21] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *Proceedings of the 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 897–907.

[22] Gatling Corp. 2024. Gatling - Professional Load Testing Tool. https://gatling.io/. Online; Accessed on 31 Mar. 2024.

[23] Chavit Denninnart, Thanawat Chanikaphon, and Mohsen Amini Salehi. 2023. Efficiency in the serverless cloud paradigm: A survey on the reusing and approximation aspects. *Software: Practice and Experience* 53, 10 (2023), 1853–1886.

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

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[26] Ana Ebrahimi, Mostafa Ghobaei-Arani, and Hadi Saboohi. 2024. Cold Start Latency Mitigation Mechanisms in Serverless Computing: Taxonomy, Review, and Future Directions. *Journal of Systems Architecture* (2024), 103115.

[27] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the costs of serverless workflows. In *Proceedings of the 11th ACM/SPEC international conference on performance engineering*. 265–276.

[28] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2021. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4152–4166.

[29] Alex Ellis. [n. d.]. OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/. Online; Accessed on 10 Oct. 2024.

[30] Brad Everman, Narmadha Rajendran, Xiaomin Li, and Ziliang Zong. 2021. Improving the cost efficiency of large-scale cloud systems running hybrid workloads-A case study of Alibaba cluster traces. *Sustainable Computing: Informatics and Systems* 30 (2021), 100528.

[31] Cloud Native Foundation. 2024. Knative. https://knative.dev/. Online; Accessed on 31 Mar. 2024.

[32] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.

[33] Samuel Ginzburg and Michael J Freedman. 2020. Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms. In *Proceedings of the 6th International Workshop on Serverless Computing*. 43–48.

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

[35] M Reza HoseinyFarahabady, Javid Taheri, Albert Y Zomaya, and Zahir Tari. 2021. Data-intensive workload consolidation in serverless (Lambda/FaaS) platforms. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*. IEEE, 1–8.

[36] ArangoDB Inc. 2024. ArangoDB. https://www.arangodb.com. Online; Accessed on 31 Mar. 2024.

[37] Intel Inc. 2013. MTBF Data for Intel Server Board S1200RP Family. https://www.intel.com/content/www/us/en/support/articles/000007550/server-products.html. Online; Accessed on 15 July 2024.

[38] Lightbend Inc. 2024. High performance microservices and APIs | Kalix.io. https://www.kalix.io. Online; Accessed on 31 Mar. 2024.

[39] MinIO Inc. 2024. MinIO | High Performance, Kubernetes Native Object Storage. https://min.io/. Online; Accessed on 31 Mar. 2024.

[40] SUSE Inc. 2024. RKE2. https://docs.rke2.io. Online; Accessed on 15 July 2024.

[41] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2022. Astrea: Auto-serverless analytics towards cost-efficiency and qos-awareness. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3833–3849.

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

[43] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[44] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM Conference 2023*. 406–419.

[45] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[46] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the ACM Symposium on Cloud Computing 2023*. 443–458.

[47] 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. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association.

[48] Stefan Kehrer, Dominik Zietlow, Jochen Scheffold, and Wolfgang Blochinger. 2021. Self-tuning serverless task farming using proactive elasticity control. *Cluster Computing* 24 (2021), 799–817.

[49] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, et al. 2022. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *arXiv preprint arXiv:2208.13068* (2022).

[50] Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Karim El Tal, and Stefan Tai. 2019. An evaluation of faas platforms as a foundation for serverless big data processing. In *Proceedings of the 12th IEEE/ACM international conference on utility and cloud computing*. 1–9.

[51] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, et al. 2021. Prediction-Based power oversubscription in cloud platforms. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 21)*. 473–487.

[52] Zhengyu Lei, Xiao Shi, Cunchi Lv, Xiaobing Yu, and Xiaofang Zhao. 2023. Chitu: accelerating serverless workflows with asynchronous state replication pipelines. In *Proceedings of the ACM symposium on cloud computing 2023*. 597–610.

[53] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. 2023. Object as a service (OaaS): Enabling object abstraction in serverless clouds. In *Proceedings of the 16th International Conference on Cloud Computing (CLOUD'23)*. IEEE, 238–248.

[54] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. 2024. Object as a Service: Simplifying Cloud-Native Development through Serverless Object Abstraction. *arXiv preprint arXiv:2408.04898* (2024).

[55] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. 2024. Tutorial: Object as a Service (OaaS) Serverless Cloud Computing Paradigm. In *Proceedings of the 44th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 5–8.

[56] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proceedings of the ACM Symposium on Cloud Computing 2023*. 32–47.

[57] Changyuan Lin and Hamzeh Khazaei. 2020. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 615–632.

[58] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. 2022. Fine-grained performance and cost modeling and optimization for faas applications. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2022), 180–194.

[59] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–29.

[60] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *Proceedings of the 6th IEEE international conference on cloud engineering (IC2E)*. IEEE, 159–169.

[61] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. {SONIC}: Application-aware data passing for chained serverless applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

[62] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 218–233.

[63] Kinga Anna Marek, Luca De Martini, and Alessandro Margara. 2023. Nubes: Object-Oriented Programming for Stateful Serverless Functions. In *Proceedings of the 9th International Workshop on Serverless Computing*. 30–35.

[64] Bakhta Meroufel and Ghalem Belalem. 2013. Managing data replication and placement based on availability. *AASRI Procedia* 5 (2013), 147–155.

[65] Microsoft. 2024. Chatty I/O antipattern. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/chatty-io/. Online; Accessed on 4 July 2024.

[66] Microsoft. 2024. Durable entities - Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities. Online; Accessed on 31 Mar. 2024.

[67] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. 2023. Parrotfish: Parametric regression for optimizing serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing 2023*. 177–192.

[68] Wilmer Moina-Rivera, Miguel Garcia-Pineda, Jose M Claver, and Juan Gutiérrez-Aguado. 2023. Event-driven serverless pipelines for video coding and quality metrics. *Journal of Grid Computing* 21, 2 (2023), 20.

[69] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data 2020*. 115–130.

[70] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the 16th European Conference on Computer Systems*. 228–244.

[71] Hai Duc Nguyen and Andrew A Chien. 2023. Storm-RTS: Stream Processing with Stable Performance for Multi-cloud and Cloud-edge. In *Proceedings of the 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 45–57.

[72] Hai Duc Nguyen, Zhifei Yang, and Andrew A Chien. 2020. Motivating high performance serverless workloads. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*. 25–32.

[73] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A Chien. 2019. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*. 1–6.

[74] Manish Pandey and Young Woo Kwon. 2023. Optimizing Memory Allocation in a Serverless Architecture through Function Scheduling. In *Proceedings of the 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. IEEE, 275–277.

[75] Fission Project. 2024. Fission. https://fission.io. https://fission.io Online; Accessed on 10 Oct. 2024.

[76] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX symposium on networked systems design and implementation (NSDI 19)*. 193–206.

[77] Resilience in AWS Lambda 2024. Resilience in AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/security-resilience.html. Online; Accessed on 14 Oct 2024.

[78] Ran Ribenzaft. 2024. What AWS Lambda's Performance Stats Reveal. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/. Online; Accessed on 6 July 2024.

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

[80] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, et al. 2020. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 497–511.

[81] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: a survey of opportunities, challenges, and applications. *Comput. Surveys* 54, 11s (2022), 1–32.

[82] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild:

Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[83] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

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

[85] Khondokar Solaiman and Muhammad Abdullah Adnan. 2020. WLEC: A not so cold architecture to mitigate cold start problem in serverless computing. In *Proceedings of the 8th IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 144–153.

[86] Jonas Spenger, Paris Carbone, and Philipp Haller. 2024. A Survey of Actor-Like Programming Models for Serverless Computing. In *Active Object Languages: Current Research Trends*. Springer, 123–146.

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

[88] Mark Szalay, Peter Matray, and Laszlo Toka. 2022. Real-time faas: Towards a latency bounded serverless cloud. *IEEE Transactions on Cloud Computing* 11, 2 (2022), 1636–1650.

[89] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 311–327.

[90] Aastik Verma, Anurag Satpathy, Sajal K Das, and Sourav Kanti Addya. 2024. LEASE: Leveraging Energy-Awareness in Serverless Edge for Latency-Sensitive IoT Services. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events 2024 (PerCom Workshops)*. IEEE, 302–307.

[91] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[92] Na Wang, Junsong Fu, Bharat K Bhargava, and Jiwen Zeng. 2018. Efficient retrieval over documents encrypted by attributes in cloud computing. *IEEE Transactions on Information Forensics and Security* 13, 10 (2018), 2653–2667.

[93] Shangrui Wu, Chavit Denninnart, Xiangbo Li, Yang Wang, and Mohsen Amini Salehi. 2020. Descriptive and predictive analysis of aggregating functions in serverless clouds: The case of video streaming. In *Proceedings of the 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 19–26.

[94] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1489–1504.

[95] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[96] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus:{NIMBLE} task scheduling for serverless analytics. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 653–669.

[97] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing 2019*. 1–12.

[98] Xuan Zhang, Hongjun Gu, Guopeng Li, Xin He, and Haisheng Tan. 2023. Online Function Caching in Serverless Edge Computing. In *Proceedings of the 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2295–2302.

[99] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1–14.