OBJECT ABSTRACTION TO STREAMLINE EDGE-CLOUD-NATIVE APPLICATION DEVELOPMENT

Pawissanutt Lertpongrujikorn

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

December 2025

Lertpongrujikorn, Pawissanutt. *Object Abstraction To Streamline Edge-Cloud-Native Application Development.* Doctor of Philosophy (Computer Science and Engineering), December 2025, 181 pp., 8 tables, 51 figures, references, 127 titles.

Cloud computing has fundamentally transformed application development, yet a persistent gap remains between the serverless promise of simplified deployment and its practical realization. Current serverless platforms suffer from fundamental fragmentation: application logic resides in Function-as-a-Service (FaaS) runtimes, state management requires separate database services, and workflow orchestration demands additional coordination layers. This dissertation addresses this gap through empirical validation and technical innovation, establishing the Object-as-a-Service (OaaS) paradigm as a unified approach to cloud-native development. Grounded in empirical evidence from three complementary studies—an initial practitioner interview study (21 participants), a human study evaluating developer experience (39 participants), and NSF I-Corps National customer discovery (101 interviews across 86 organizations)—this work demonstrates that infrastructure complexity imposes substantial tax on developer productivity, with practitioners consistently prioritizing productivity, automation, and operational maintainability over cost optimization.

The dissertation makes five major contributions: (1) the OaaS paradigm unifies resource, state, and workflow management into a single object-oriented abstraction through the Oparaca prototype, demonstrating negligible performance overhead while achieving state-of-the-art scalability; (2) SLA-driven OaaS introduces declarative Non-functional Requirement (NFR) management, enabling developers to specify availability, throughput, consistency, and latency targets through high-level specifications; (3) OaaS-IoT with EdgeWeaver extends the paradigm across the edge-cloud continuum with SLA-driven placement and connectivity-aware invocation, achieving 31% faster task completion, 44.5% reduction in lines of code, and 10× fewer configuration requirements compared to traditional FaaS approaches; (4) commercialization validation establishes a staged pathway targeting technology SMEs and startups as primary early adopters; and (5) the empirical methodology demonstrates the critical importance of grounding technical research in validated practitioner needs. By con-

solidating fragmented abstractions, automating performance optimization, and extending seamlessly to the edge, OaaS establishes a foundation for cloud-native platforms that truly deliver on the promise of hiding infrastructure complexity and empowering developers to focus on innovation.

To my family,

whose love sustained me through the challenges

and celebrated with me in the triumphs.

ACKNOWLEDGMENTS

I express my deepest gratitude to my major professor, Dr. Mohsen Amini Salehi, for his unwavering support, insightful guidance, and patience throughout my doctoral journey. His expertise in distributed systems and cloud computing has been instrumental in shaping this research and my development as a researcher.

I am grateful to my committee members for their valuable feedback and constructive criticism, which significantly improved the quality of this work.

I would like to acknowledge the National Science Foundation for supporting this research and the I-Corps National program. I am particularly thankful to the 100+ industry professionals who participated in our empirical studies—their candid insights about cloud-native development challenges provided the empirical foundation that motivated and validated this research. I also extend my gratitude to the 39 college students who participated in the human study evaluating the OaaS platform; their dedication in completing tutorials, quizzes, and programming assignments provided crucial validation of the system's usability and developer experience.

I am grateful to Chameleon Cloud for providing the research infrastructure and to my collaborators, particularly Hai Duc Nguyen, for their significant contributions to this work.

I am deeply indebted to my family for their unconditional love and support. To my parents, who instilled in me a love of learning and perseverance—your sacrifices and encouragement made this achievement possible.

I am grateful to my labmates, colleagues, and friends for the community, perspective, and encouragement they provided throughout this journey.

Finally, I acknowledge the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM) for permission to reprint previously published material. Portions of this dissertation were published by IEEE and ACM and are used with permission.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The emergence of cloud technology has drastically transformed the application development process. With cloud infrastructure, provisioning can now be done in a few minutes instead of the weeks or months it used to take. Over the past decade, cloud services have replaced mundane tasks with software automation. The current state-of-the-art, serverless platform utilizes the function-as-a-service (FaaS) paradigm to enable developers to build applications by simply writing code in the form of a function and uploading it to the platform. The system then automates the process of building, deploying, and auto-scaling the application, making the overall development process more effortless and mitigating the burden for programmers and cloud solution architects. Major public cloud providers offer FaaS services (e.g., AWS Lambda [3], Google Cloud Function [29], Azure Function [78]), and several open-source platforms for on-premise FaaS deployments are emerging (e.g., OpenFaaS [42], Knative [46]). In the backend, the serverless platform hides the complexity of resource management and deploys the function seamlessly in a scalable manner. FaaS is proven to reduce development and operation costs via implementing scale-to-zero and charging the user in a truly pay-as-you-go manner. Thus, it aligns with modern software development paradigms, such as CI/CD and DevOps [16].

1.1. Empirical Motivation: Understanding Practitioner Challenges

The rapid advancement of cloud computing has reshaped modern business strategies by enabling scalability, agility, and resilience [50]. However, as industries increasingly shift toward autonomous, real-time, and localized data processing in highly distributed systems, there is growing demand for computing models that also support low-latency responsiveness, localized decision-making, and flexible deployment [44, 83]. These capabilities are vital for several business domains—such as healthcare, IoT, finance, and robotics—where fast and adaptive software development and deployment are critical.

Despite significant advancements in cloud-native technologies, a persistent gap remains between the challenges acknowledged in academic research and those experienced by practitioners in real-world business domains. Scholarly efforts often overlook critical constraints such as steep learning curves, fragmented toolchains, and limited technical capacity—especially prevalent among small enterprises and domain-specific organizations. These oversights are partly due to academic reliance on idealized assumptions, including underestimated cost implications, simplified system models, and minimal consideration of organizational limitations. This disconnect is further widened by the rapid evolution of cloud platforms and a lack of sustained collaboration between academic researchers and industry stakeholders. As a result, many proposed cloud-native frameworks, despite their conceptual promises, fall short when applied in operational environments.

To help bridge this research-practice divide and ground the motivation for this dissertation, we present empirical findings from interviews with domain-specific professionals directly engaged in the design, development, and adoption of cloud-native systems. These interviews aim to illuminate issues that are often underrepresented in academic discourse—specifically, the practical pain points experienced by practitioners, the expectations they hold for emerging technologies, and the business factors that shape decisions around cloud adoption. By centering these real-world perspectives, we seek to surface latent challenges that hinder the practical uptake of academic innovations and advocate for a more grounded research agenda—one that not only prioritizes technical novelty but also accounts for the operational, organizational, and economic realities of modern software development ecosystems.

1.1.1. Market Survey: Hypothesis and Methodology

Cloud computing emerged in the early 2000s alongside advancements in virtualization and web hosting technologies. Virtualization tools like VMware [113] enabled hardware abstraction, allowing multiple virtual machines to run on a single physical server, thus greatly improving resource utilization. By hiding infrastructure complexity, cloud platforms promised to reduce infrastructure management burdens and empower organizations, especially those with limited technical capacity, to develop and scale digital services more easily.

This vision led to widespread adoption of cloud-native technologies such as container orchestration systems [30] and serverless computing [3, 29].

Despite their potential, modern cloud platforms have evolved into a highly complex and fragmented landscape. In fact, major commercial cloud providers (e.g., Amazon AWS, Google Cloud, and Microsoft Azure) have collectively introduced over **500** new services since 2006 [21]. Many of these services demand deep technical expertise to configure, integrate, and operate effectively. Technologies like Kubernetes [30], Function-as-a-Service (FaaS) [3, 29], CI/CD pipelines [48], and microservices architectures compound steep learning curves and require developers to master a large and ever-growing set of tools and best practices. As a result, fundamental routine tasks—such as deploying an application or configuring its performance—now require a high level of expertise in cloud-native principles and platform-specific knowledge [69].

This escalating technical and conceptual complexity directly contradicts the cloud's foundational promise of streamlined IT and universal accessibility. For startups, small enterprises, and domain-focused organizations without dedicated technical teams, adopting cloud-native solutions can be prohibitively complicated. Organizations in these categories often face challenges such as talent shortages, prolonged development cycles, debugging and integration overhead, and constrained budgets.

This observation leads to the following hypothesis:

Hypothesis: *The current real-world practice of cloud-native application development, deployment, and maintenance is complicated, requiring significant time and specialized expertise, which increases the overall costs of cloud software production.*

This hypothesis stems from a thorough examination of existing cloud-native systems and industry reports. Through preliminary investigation, we identified a recurring pattern: domain-specific professionals often struggle with fragmented toolchains, complex deployment processes, and steep learning curves associated with distributed computing.

To verify this hypothesis, we conducted interviews with **21 developers and industry professionals** across multiple business domains, asking questions focused on the pain points they faced, their expectations for solutions, and the obstacles to implementing them. The interviews covered diverse sectors, including agriculture, chips, cybersecurity, education, finance, healthcare, industrial IoT, social networks, and technology consulting, with participants split between technical and non-technical roles.

1.1.2. Key Survey Findings: Pain Points

During interviews, participants described a wide range of challenges encountered when adopting and scaling cloud-based solutions. These challenges span performance bottlenecks, DevOps limitations, and infrastructure trade-offs. Based on their responses, we classified the reported issues into six distinct categories, summarized below:

Operational Maintainability (57% of participants): The most frequently reported pain poin is, often linked toa lack of dedicated technical personnel, limited cloud provider support, and the complexity of configuring and managing cloud services. This issue disproportionately affected non-technical organizations.

Development Complexity (48% of participants): Organizations reported significant difficulty in navigating fragmented toolchains and integrating multiple cloud services. Non-tech participants reported this issue at a rate **4× higher** than tech participants, indicating that technical expertise gaps substantially hinder cloud adoption.

Responsiveness (43% of participants): Performance and latency issues were common across both technical and non-technical domains, particularly for distributed and edge computing scenarios. Most participants reported that these issues remained partially unresolved.

Reliability and Availability (33% of participants): Concerns were particularly significant in organizations with IoT-centric and hybrid computing environments, especially those lacking dedicated DevOps or SRE teams. None reported full resolution of these issues, though some (57%) implemented partial mitigations.

Security and Privacy (38% of participants): Highly reported by organizations handling sensitive data (finance, healthcare, education). Some addressed concerns by adopting hybrid or private cloud solutions. Remote sites with unreliable network connectivity faced additional challenges, particularly for authentication systems that must function during disconnections.

Cost (24% of participants): Appeared to be of less immediate concern, likely because many technical interviewees were not directly involved in budgeting decisions. However, cost pressures were indirectly evident through the resource constraints mentioned in other categories.

Key Insight: *Most reported pain points have remained unresolved, especially for non-tech organizations lacking technical expertise and dedicated support teams. Across all categories, non-technical participants consistently reported higher rates of challenges than their technical counterparts.*

1.1.3. Key Survey Findings: Industry Expectations

Participants articulated clear expectations for what cloud platforms or underlying technologies must provide to address their current pain points and better support future application development and deployment. We categorized these expectations into five feature groups:

Availability & Reliability (43% total; 67% non-tech, 33% tech): High availability (minimum 99.9%), quick support contact, and well-defined disaster recovery. Non-technical organizations particularly emphasized this expectation, as they typically lack in-house technical support and depend heavily on platform reliability.

Security & Privacy (29% total; 67% non-tech, 33% tech): Strong and resilient security systems with data encryption, especially critical for sectors dealing with regulated or sensitive data.

Service Quality Assurance (33% total; 57% tech, 43% non-tech): High SLA offerings, rich configurability for performance optimization, system scalability, and capability. Tech participants prioritized this more highly, reflecting their reliance on underlying technology performance to deliver quality digital services.

Unified Maintainability (33% total; 57% tech, 43% non-tech): Support for hybrid cloud, automation (DevOps), and log aggregation. Tech interviewees emphasized streamlined deployment, updates, monitoring, and debugging to reduce operational overhead.

Programmability (38% total; 50% tech, 50% non-tech): Ease of development and user-friendly interfaces. While tech interviewees appreciated customization through APIs/SDKs, non-tech respondents focused on low-code capabilities that empower broader teams without deep technical skills.

Key Insights: *Tech participants prioritize QoS (availability, security, SLA) to deliver reliable digital services. Non-tech participants prioritize simplification (unified maintainability, programmability) due to limited technical expertise. The even distribution across categories indicates strong demand for comprehensive unified platforms.*

1.1.4. Discussion: Survey Implications for this Dissertation

Drawing from the empirical evidence, two major conclusions refine and expand our original hypothesis:

Complexity Despite Technology Advancements: Although cloud technologies have continued to evolve, the interviews clearly show that cloud users, especially those in non-technical domains, still struggle with the complexity of real-world adoption. Instead of simplifying the development and deployment process, many modern cloud paradigms introduce new layers of abstraction that require even more specialized knowledge and operational overhead. These findings validate the original hypothesis that current cloud-native practices remain complicated, but they go further: the complexity is not simply due to immature technology but is often embedded in the very design of modern cloud platforms. This insight underscores the pressing need for a unified and streamlined approach to cloud infrastructure that can mitigate fragmentation and lower the technical barrier to adoption.

Prioritizing Productivity and QoS over Cost and Migration: In contrast to common assumptions that practitioners primarily consider cost and migration concerns when adopting new technologies, our participants revealed a different reality. Across roles and domains, neither cost nor migration emerged as the primary barrier to adoption. Instead, their focus centered on expectations such as Quality of Service (QoS), simplified programmability, and improved accessibility. In fact, many participants expressed willingness to endure short-term migration challenges or higher upfront costs if the platform could offer meaningful improvements in service quality, operational simplicity, and development speed. This finding refines our original hypothesis by highlighting that while high cost is a consequence of complicated solution lifecycles, the root cause is deeper, stemming from productivity gaps and architectural complexity in today's cloud ecosystems.

Implications for Cloud Platform Design: These empirical findings directly motivate the technical contributions of this dissertation. The survey reveals that practitioners need:

- **Unified abstraction** to reduce fragmentation (operational maintainability, development complexity)
- **Quality-of-service guarantees** with intuitive interfaces (service quality assurance, availability/reliability)
- **Simplified programmability** that hides infrastructure complexity (programmability, ease of development)
- **Comprehensive platform support** across distributed environments (responsiveness, edge-cloud integration)

These practitioners needs point toward a fundamental rethinking of serverless platform design—one that unifies resource, state, and workflow management to reduce fragmentation; provides declarative QoS interfaces to guarantee service quality without deep expertise; and extends seamlessly across edge-cloud continua for distributed, responsive applications.

1.2. Technical Motivation: Limitations of Current FaaS Platforms

The empirical findings establish *what* practitioners need from cloud platforms. To understand *why* current solutions fall short and *how* to address these needs, we must examine the technical limitations of existing serverless platforms. This section focuses on Function-as-a-Service (FaaS)—the current state-of-the-art serverless paradigm—and identifies how its fundamental design constraints directly manifest as the pain points reported by practitioners.

While FaaS represents significant progress toward allowing developers to focus on business logic by automating infrastructure management and scaling, it remains constrained by three fundamental limitations that directly correlate with the practitioner pain points identified earlier:

Limitation 1: Fragmented State Management: The FaaS paradigm centers on *stateless functions*, deliberately excluding *state data* from its abstraction. In practice, most applications require persistent state—structured or unstructured—forcing developers to manually integrate external storage services (e.g., AWS S3 [5]). This fragmentation manifests as the *development complexity* pain point (48% of practitioners): developers must write additional code to manage data persistence, coordinate access across functions, and handle consistency. For complex workflows, developers resort to using external storage as communication channels between functions, further compounding the integration burden and directly contributing to the *operational maintainability* challenges (57% of practitioners).

Limitation 2: Lack of Non-functional Requirement (NFR) Control: FaaS platforms provide separate service abstractions for compute, databases, workflows, and messaging, preventing holistic application optimization. The platform operates with minimal knowledge of application semantics, while developers have limited ability to "hint" performance requirements or influence optimization decisions (e.g., data locality, caching policies). Non-functional requirements—such as Quality-of-Service (QoS) guarantees, consistency levels, and availability targets—cannot be declaratively specified within the FaaS abstraction. This architectural separation directly causes the *service quality assurance* concerns reported by 33% of practitioners—developers cannot declaratively specify performance targets, forcing them into trial-and-error tuning cycles that increase time-to-deployment and operational costs.

Limitation 3: Single-Cluster Deployment Model: When applications must serve users across geographic regions or at the edge, meeting QoS requirements within a single cluster becomes infeasible. Developers must manually provision multiple clusters, partition application state and logic, and coordinate distributed execution—tasks that require deep expertise in distributed systems. Edge deployments (Fog Computing [34]) compound this complexity with heterogeneous hardware, intermittent connectivity, and diverse protocol requirements.

This limitation directly explains the *responsiveness* pain point (43% of practitioners) and *reliability/availability* concerns in distributed environments, particularly for non-technical organizations lacking distributed systems expertise.

1.3. Research Problems and Dissertation Approach

The preceding analysis reveals a clear research agenda: current serverless platforms fail to address practitioner needs because of fundamental architectural constraints—fragmented state management, lack of NFR control, and single-cluster deployment models. These technical limitations directly cause the pain points experienced by practitioners. This dissertation addresses three interconnected research problems that systematically tackle these limitations:

- **Unified Serverless Abstraction:** How can we design a serverless paradigm that abstracts resource, state, and workflow management into a cohesive programming model, eliminating fragmentation and allowing developers to focus solely on business logic?
- **Declarative NFR/SLA Automation:** How can we enable developers to specify non-functional requirements (quality-of-service, consistency, availability) through intuitive, high-level metrics that drive automated platform optimization without requiring deep cloud expertise?
- **Distributed Deployment:** How can we extend NFR-aware serverless systems to seamlessly deploy and optimize applications across geo-distributed clusters and edge-cloud continua while maintaining unified abstractions?

To address these problems, we propose a three-stage research approach illustrated in Figure 1.1, progressively expanding serverless capabilities from unified abstraction (OaaS) to NFR-driven automation to federated edge-cloud deployment.

1.3.1. Stage 1: Object-as-a-Service (OaaS) — Unified Abstraction

To address the fragmented state management limitation and the first research problem, we propose **Object-as-a-Service (OaaS)** [70, 71]—a new serverless paradigm that borrows the notion of "object" from object-oriented programming (OOP) to unify resource,

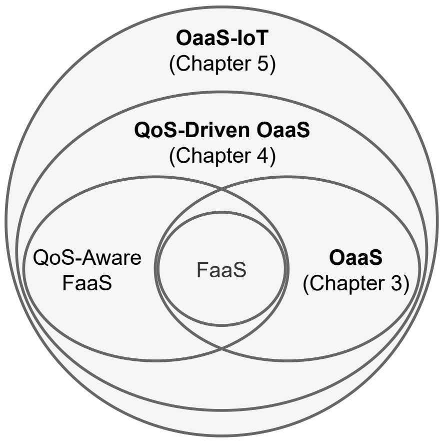
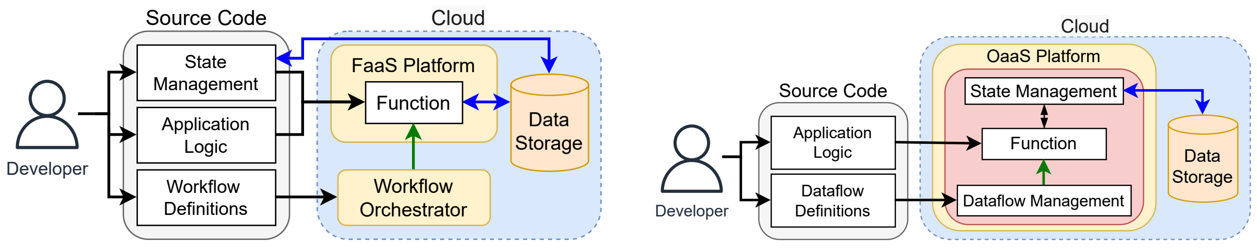


FIGURE 1.1. Three-stage evolution from FaaS to federated NFR-aware OaaS, progressively addressing fragmentation (Stage 1), NFR control (Stage 2), and distributed deployment (Stage 3).



(A) Function as a Service (FaaS)   (B) Object as a Service (OaaS)

FIGURE 1.2. Architectural comparison: FaaS fragments state and workflow across external services, while OaaS unifies them within platform-managed objects.

state, and workflow management within the platform abstraction. This directly tackles the *development complexity* (48%) and *operational maintainability* (57%) pain points by eliminating the need for developers to manually integrate fragmented services.

As illustrated in Figure 1.2, OaaS fundamentally differs from FaaS by incorporating state management directly into the platform, making it transparent to developers. Where FaaS forces developers to coordinate separate storage services and manage data flow manually, OaaS provides:

- **Platform-Managed State:** State is segregated from application code and managed by the platform as part of the object abstraction, eliminating manual storage integration.

- **Built-in Workflow Orchestration:** Dataflow semantics are native to the programming model—developers declare data flow between functions rather than implementing communication channels.
- **Unified Optimization Opportunities:** By encapsulating compute, state, and workflow within objects, the platform can apply holistic optimizations (data locality, caching, replication) without developer intervention.

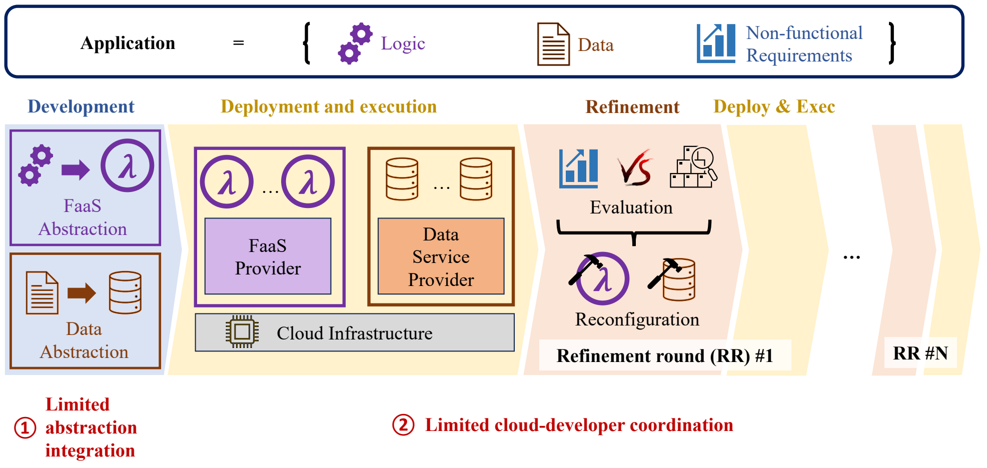
Developers interact with OaaS through familiar `class` and `function` declarations, expressing application logic in an object-oriented style while the platform handles resource provisioning, state persistence, fault tolerance, and inter-function data navigation. This unified abstraction directly addresses practitioner expectations for *simplified programmability* (38%) and *unified maintainability* (33%).

1.3.2. Stage 2: SLA-Driven OaaS — Declarative Performance Control

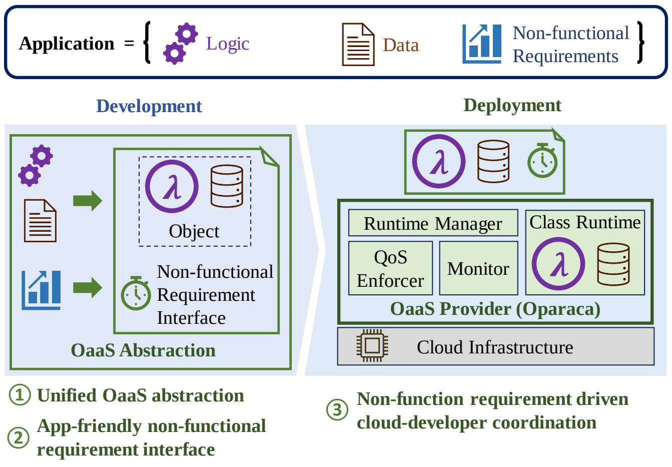
With OaaS providing unified abstraction of resources, state, and workflows, we can now address the second research problem: enabling declarative NFR control. The key insight is that by unifying the programming model, the platform gains complete semantic knowledge of the application—enabling automated optimization across compute, storage, and workflow without cross-domain constraints. This directly tackles the *service quality assurance* expectations (33%) and reduces the trial-and-error refinement cycles that plague FaaS development (Figure 1.3a).

NFR-Driven OaaS [69] extends the base paradigm with two key mechanisms:

- **Declarative NFR/SLA Interfaces:** Developers specify non-functional requirements—including performance targets (latency, throughput), consistency levels, and availability guarantees—alongside class definitions using intuitive, high-level metrics [18]. The platform interprets these requirements to guide optimization decisions—or proactively rejects infeasible specifications, eliminating costly deployment



(A) Current FaaS lifecycle: Trial-and-error refinement cycles



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) NFR-driven OaaS: Declarative requirements with automated refinement

Figure 1.3. Comparison of development lifecycles: FaaS requires repeated manual refinement to meet NFR targets, while NFR-driven OaaS automates optimization through declarative specifications, directly addressing *service quality assurance* expectations (33% of practitioners).

failures.

- **Dynamic Class Runtimes:** Rather than sharing a monolithic runtime across all classes (which creates conflicting optimization goals), the platform dynamically provisions dedicated *class runtimes* tailored to each class's NFR specifications. For example, cost-sensitive classes can disable expensive components (in-memory caches), while latency-critical classes receive aggressive caching and replication.

This approach creates *self-contained, portable objects* where behavior and performance are determined by declarative specifications rather than manual tuning. Developers benefit from a programming model focused on business logic (addressing *simplified programmability*, 38%), while the platform guarantees performance across deployments—enabling seamless migration between providers without code changes.

1.3.3. Stage 3: OaaS-IoT — Distributed Edge-Cloud Deployment

The third research problem extends NFR-driven OaaS to geo-distributed environments, specifically the Edge-Cloud continuum, where applications must span heterogeneous, intermittently connected computing tiers. This addresses the *responsiveness* pain point (43%) and *reliability/availability* concerns in distributed IoT deployments—from intelligent transportation and industrial automation to healthcare and smart cities. Current FaaS platforms require manual cluster management, state partitioning, and distributed coordination, creating prohibitive complexity for non-technical organizations.

The Edge-Cloud continuum introduces two fundamental challenges (Figure 1.4). First, *amplified development complexity* arises from infrastructure heterogeneity (k3s at edge vs. k8s in cloud), protocol diversity (MQTT, Kafka), and FaaS's lack of long-lived function support—essential for continuous IoT data processing. Component counts nearly double compared to cloud-only deployments [37, 94]. Second, *intermittent connectivity* between edge and cloud tiers creates network partitions where the CAP theorem forces unavoidable trade-offs between consistency and availability [22, 43]. Time-sensitive applications (industrial automation, healthcare) require availability during partitions, while others (financial transactions) demand consistency—yet current platforms lack mechanisms for declaratively specifying these application-specific requirements. Chapter 2 provides detailed technical analysis of these Edge-Cloud challenges.

To systematically address these challenges, we propose **OaaS-IoT**—an extension of the OaaS paradigm across the Edge-Cloud continuum. We realize OaaS-IoT through the

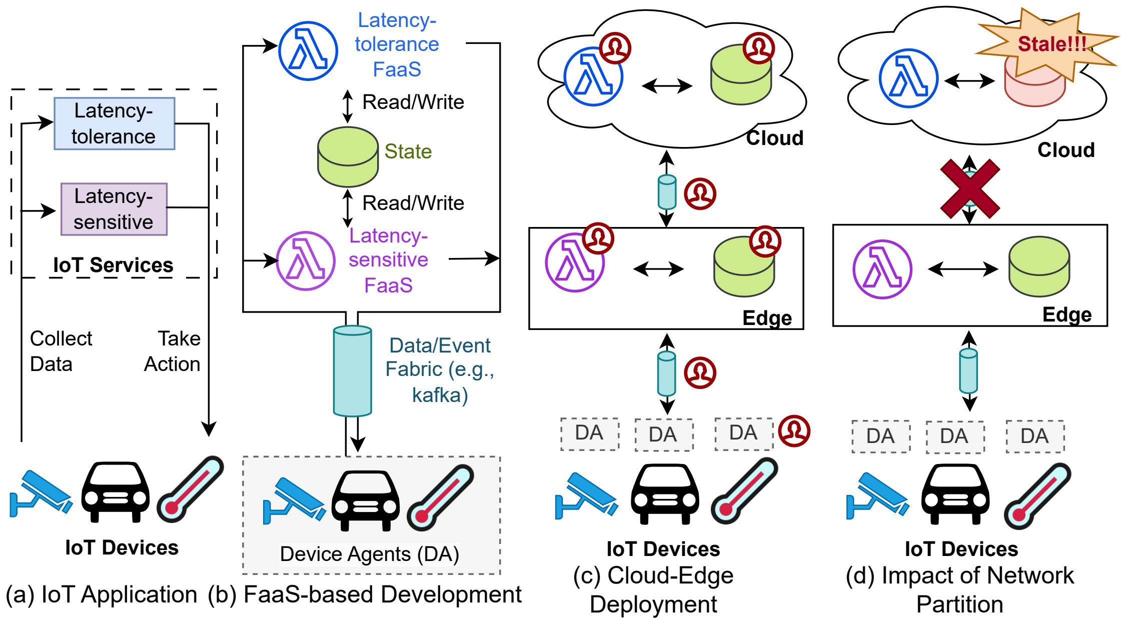


FIGURE 1.4. FaaS-based development and deployment challenges for IoT applications spanning edge-cloud continuum

EdgeWeaver platform. OaaS-IoT's key contributions are:

- **Distributed Objects:** Seamlessly span edge and cloud tiers, encapsulating application logic, state, IoT interactions, and communication patterns within unified class-based abstractions. Developers use the same object-oriented interface regardless of deployment location.
- **Declarative NFR/SLA Specification:** Extends NFR interfaces to include consistency levels, availability requirements under partitions, and geo-placement preferences. The EdgeWeaver platform interprets these specifications to guide function placement, state replication, and runtime adaptation.
- **Distributed Class Runtimes:** EdgeWeaver deploys class runtimes across Edge-Cloud tiers to hide infrastructure heterogeneity and enable location-transparent function calls. It leverages Raft and CRDTs for flexible consistency (eventual to strong), adaptive replication for availability during partitions, and rate-guarantee abstractions for performance targets.

This unified approach directly addresses all three practitioner pain points in distributed scenarios: *development complexity* through unified abstractions, *responsiveness* through edge placement, and *reliability/availability* through declarative SLA enforcement—all without requiring distributed systems expertise.

1.3.4. OaaS Positioning within the Cloud-Native Landscape

Having presented OaaS's three-stage evolution—from unified abstraction to declarative NFR control to edge-cloud deployment—we now situate these contributions within the complete cloud-native development landscape. Figure 1.5 organizes this landscape into five architectural layers, mapping where OaaS addresses critical gaps while honestly acknowledging limitations that remain for future research. Figure 1.5 reveals how the three-stage OaaS approach addresses challenges across five architectural layers:

Application Layer: OaaS's object-oriented abstractions and native dataflow (Chapter 3) address business logic complexity and workflow orchestration. End-to-end workflow guarantees across distributed components and AI-assisted code generation remain areas for future work.

Abstraction Layer: Unified object abstraction (Chapter 3) consolidates fragmented services (compute, state, workflow) and eliminates state management chaos through platform-managed persistence. Declarative SLA interfaces (Chapter 4) replace manual fine-tuning with intuitive specifications—developers declare outcomes (availability, throughput, consistency) rather than mechanisms (cache sizes, replica counts). Class versioning and live updates remain partially addressed.

Runtime Layer: Dynamic class runtimes (Chapter 4) enable SLA enforcement through automated resource allocation, while EdgeWeaver's distributed runtimes (Chapter 5) abstract consistency models and enable geo-distribution across edge-cloud tiers.

Operations Layer: Observability, debugging, and security represent cross-cutting concerns spanning all layers, remaining largely unaddressed in current OaaS implementations. These operational challenges manifest differently at each layer—from application-level workflow debugging to runtime SLA compliance monitoring to infrastructure security audit-

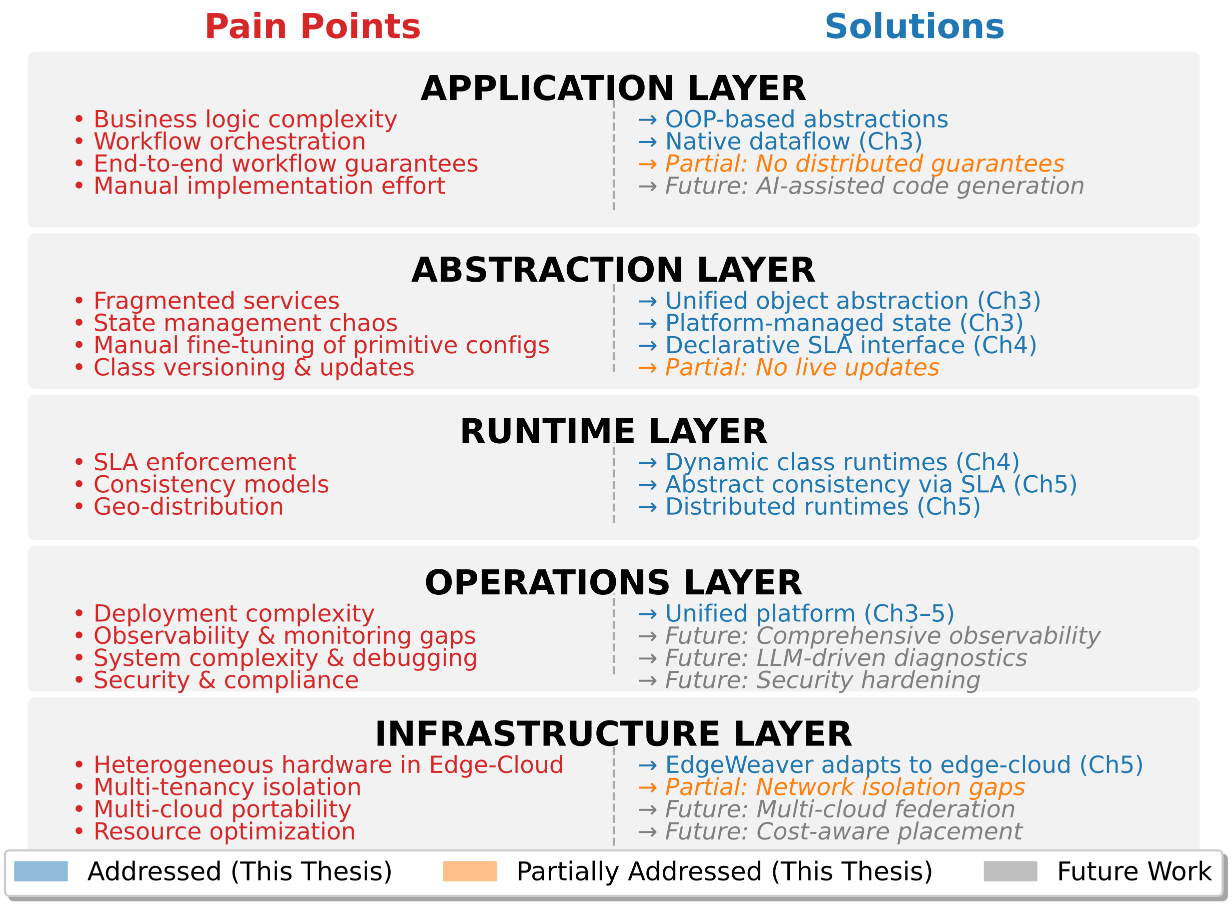

Pain Points
Solutions
APPLICATION LAYER
• Business logic complexity
• Workflow orchestration
• End-to-end workflow guarantees
• Manual implementation effort
→ OOP-based abstractions
→ Native dataflow (Ch3)
→ Partial: No distributed guarantees
→ Future: AI-assisted code generation
ABSTRACTION LAYER
• Fragmented services
• State management chaos
• Manual fine-tuning of primitive configs
• Class versioning & updates
→ Unified object abstraction (Ch3)
→ Platform-managed state (Ch3)
→ Declarative SLA interface (Ch4)
→ Partial: No live updates
RUNTIME LAYER
• SLA enforcement
• Consistency models
• Geo-distribution
→ Dynamic class runtimes (Ch4)
→ Abstract consistency via SLA (Ch5)
→ Distributed runtimes (Ch5)
OPERATIONS LAYER
• Deployment complexity
• Observability & monitoring gaps
• System complexity & debugging
• Security & compliance
→ Unified platform (Ch3–5)
→ Future: Comprehensive observability
→ Future: LLM-driven diagnostics
→ Future: Security hardening
INFRASTRUCTURE LAYER
• Heterogeneous hardware in Edge-Cloud
• Multi-tenancy isolation
• Multi-cloud portability
• Resource optimization
→ EdgeWeaver adapts to edge-cloud (Ch5)
→ Partial: Network isolation gaps
→ Future: Multi-cloud federation
→ Future: Cost-aware placement
Addressed (This Thesis)
Partially Addressed (This Thesis)
Future Work

FIGURE 1.5. Architecture landscape showing five layers of cloud-native development challenges. Pain points and solutions map thesis contributions, partial solutions, and future work, contextualizing where OaaS addresses critical gaps while acknowledging limitations for future research.

ing—yet require unified solutions for production readiness. Notably, *deployment complexity*—stemming from service fragmentation and manual low-level configuration—is mitigated by OaaS's unified platform (Chapters 3–5), which consolidates compute, state, and workflow into cohesive objects and replaces primitive-level tuning with declarative SLAs. End-to-end release automation and full operational integration remain future work.

Infrastructure Layer: EdgeWeaver (Chapter 5) demonstrates adaptation to heterogeneous hardware in edge-cloud environments through SLA-driven placement and connectivity-aware invocation. Multi-tenancy isolation, multi-cloud federation, and cost-aware placement optimization remain areas for future work.

This layered perspective reveals that while this dissertation makes substantial progress in the top three layers (Application, Abstraction, Runtime), significant opportunities remain for future research—particularly in operations tooling and infrastructure optimization. The honest assessment of partial coverage strengthens rather than diminishes the contribution by clearly delineating solved problems from open challenges, establishing concrete research directions addressed in Chapter 7.

1.4. Contributions

Grounded in empirical evidence from practitioner interviews that reveal persistent complexity in cloud-native development, this dissertation makes the following major contributions:

- **Empirical Validation of Practitioner Challenges:** We present findings from interviews with 21 industry professionals across diverse domains, identifying six categories of pain points (operational maintainability, development complexity, responsiveness, reliability/availability, security/privacy, and cost) and five categories of expectations (availability/reliability, security/privacy, service quality assurance, unified maintainability, and programmability). These findings validate that current cloud-native practices remain complicated and costly, with non-technical organizations facing disproportionately higher challenges (4× higher development complexity rates).
- **Object-as-a-Service (OaaS) Paradigm:** We propose a new serverless paradigm that abstracts resource, state, and workflow management into the notion of an object, directly addressing the fragmentation and development complexity pain points identified by practitioners. OaaS unifies the programming model by incorporating state management and workflow orchestration into the platform, enabling developers to focus solely on business logic.
- **SLA-Driven Serverless System:** We present an NFR-aware serverless system that exploits the advantages of OaaS to drive performance optimization with intu-

itive high-level non-functional requirement specifications. This contribution directly addresses practitioner expectations for service quality assurance (33% of participants) and simplified programmability (38% of participants) by providing declarative NFR interfaces and automated optimization without requiring deep cloud expertise.

- **OaaS-IoT Paradigm and EdgeWeaver Platform:** We propose OaaS-IoT, an extension of the OaaS paradigm to the Edge-Cloud continuum, and realize it through the EdgeWeaver platform. This architecture scales QoS-aware serverless systems across geo-distributed edge-cloud environments with a unified deployment interface. This contribution addresses the responsiveness pain point (43% of participants) and enables deployment across the computing continuum from cloud to edge.
- **Commercialization Validation:** We present insights from NSF I-Corps National customer discovery (101 interviews, 86 organizations) that corroborate and expand the initial survey findings, validating the real-world demand for simplified, unified cloud-native platforms and informing the path toward practical adoption of OaaS technologies.

1.5. Dissertation Organization

This dissertation is organized into seven chapters. Following this introduction, the remaining chapters are structured as follows:

- **Chapter 2: Background and Related Work**

  Presents technical background on serverless computing, stateful serverless systems, consistency models, and edge computing fundamentals. Synthesizes related work from multiple research domains including cloud-native platforms, QoS automation, and edge-cloud integration. This chapter provides the technical foundation necessary to understand the OaaS paradigm and its contributions.
- **Chapter 3: Object-as-a-Service (OaaS) Core**

  Proposes the Object-as-a-Service paradigm, presenting the architecture, invocation model, consistency mechanisms, fault tolerance, and dataflow abstractions. Evaluates the OaaS implementation against state-of-the-art serverless platforms, demonstrating performance improvements and reduced development complexity. This chapter directly addresses the unified abstraction and simplified programmability needs identified in the empirical motivation. The OaaS implementation is available at `https://github.com/hpcclab/OaaS`. The results of this chapter have been published in the following papers:

  – **P. Lertpongrujikorn** and M. Amini Salehi, "Object as a Service: Simplifying Cloud-Native Development through Serverless Object Abstraction," *IEEE Transactions on Computers*, accepted in Oct. 2025, In Press.
  – **P. Lertpongrujikorn** and M. Amini Salehi, "Object as a Service (OaaS): Enabling Object Abstraction in Serverless Clouds," *In Proceedings of the 16th IEEE International Conference on Cloud Computing (IEEE CLOUD '23)*, Chicago, IL, USA, 2023, pp. 238-248.

- **Chapter 4: SLA-Driven OaaS: Declarative Performance Control**

  Presents an NFR-driven serverless system that exploits OaaS to enable performance optimization through intuitive, high-level non-functional requirement specifications. Introduces declarative NFR interfaces, dynamic class runtimes, and automated optimization mechanisms. Demonstrates how declarative NFR specifications address the service quality assurance expectations identified by practitioners. The SLA-driven OaaS implementation is available at `https://github.com/hpcclab/OaaS`. The results of this chapter have been published in the following paper:

  – **P. Lertpongrujikorn**, H. D. Nguyen, and M. Amini Salehi, "Streamlining Cloud-Native Application Development and Deployment with Robust Encapsulation," *Proceedings of the ACM Symposium on Cloud Computing (SoCC '24)*, Redmond, WA, USA, 2024.

- **Chapter 5: OaaS-IoT: Extending Object as a Service to the Edge–Cloud Continuum**

  Proposes OaaS-IoT, an extension of the OaaS paradigm to geo-distributed edge-cloud environments, and its realization through the EdgeWeaver platform. Addresses edge placement strategies, connectivity-aware invocation, state synchronization across the computing continuum, and geo-location-aware performance optimization. This chapter directly tackles the responsiveness and distributed deployment challenges identified in the empirical study. The EdgeWeaver/OaaS-IoT implementation is available at `https://github.com/hpcclab/OaaS-IoT`. The results of this chapter have been submitted to the following venue:

  – **P. Lertpongrujikorn**, H. D. Nguyen, and M. Amini Salehi, "EdgeWeaver: Seamless Edge-Cloud Integration for Stateful Serverless Applications," *40th IEEE International Parallel and Distributed Processing Symposium (IPDPS '26)*, 2026 (under review).

- **Chapter 6: Exploration and Analysis of the OaaS Productization**
  Presents customer discovery findings from NSF I-Corps National Summer 2025 program (101 interviews, 86 organizations). Documents methodology, synthesizes industry-specific pain points and desired outcomes, and validates the real-world demand for OaaS-style unified platforms. Discusses value proposition, adoption risks, and paths to commercialization. This chapter corroborates and expands the initial empirical motivation with 5× more data points. The results of this chapter have been submitted to the following venue:

  – **P. Lertpongrujikorn**, H. D. Nguyen, J. Kwon, and M. Amini Salehi, "Exploring Challenges in Developing Edge-Cloud-Native Applications Across Multiple Business Domains," *Cloud-Network Convergence*, 2026 (under review).

- **Chapter 7: Conclusion and Future Directions**
  Summarizes the contributions of this dissertation, discusses lessons learned and limitations, and outlines future research directions, including LLM-driven self-evolving objects and multi-cloud object federation. Integrates insights from both empirical studies to propose a research agenda grounded in practitioner needs and operational realities.

CHAPTER 2

BACKGROUND AND LITERATURE STUDY

2.1. Towards the Next Generation of Cloud Computing Systems

As established in Chapter 1, empirical evidence from practitioners reveals persistent complexity in cloud-native development, with fragmented toolchains, inadequate NFR control, and limited support for distributed deployment identified as major barriers to adoption [69]. These practical challenges stem from fundamental architectural constraints in current serverless platforms—particularly the Function-as-a-Service (FaaS) paradigm [3, 78, 29]—which force developers to manually coordinate separate services for compute, state management, and workflow orchestration.

Cloud platforms are evolving beyond VM- and container-centric abstractions toward managed, intent-driven services that emphasize elasticity, programmability, and developer productivity. This evolution manifests as: (i) event-driven and dataflow-centric programming models [120]; (ii) deeper disaggregation of compute, storage, and control planes [2]; and (iii) automation that optimizes for application-level objectives (e.g., latency percentiles, availability, and cost) [111, 127]. The three-stage approach presented in this dissertation—OaaS core (Chapter 3), NFR-driven OaaS (Chapter 4), and OaaS-IoT with EdgeWeaver (Chapter 5)—builds on this trajectory by systematically addressing the practitioner pain points through unified abstraction, declarative NFR control, and seamless edge-cloud integration.

This chapter surveys the technical landscape that both motivates and contextualizes our contributions. We examine state-of-the-art research in stateful serverless computing, QoS-aware systems, and edge-cloud platforms to identify specific gaps that the OaaS paradigm addresses. Where Chapter 1 establishes *what* practitioners need and *why* current platforms fail, this chapter explores *how* existing research has attempted to solve similar problems and where opportunities remain for the unified, NFR-driven, and distributed approach proposed in this dissertation.

2.2. Serverless Computing and Function-as-a-Service (FaaS)

Serverless computing popularized the separation of application code from infrastructure management. In its prevalent *Function-as-a-Service* (FaaS) form, developers package functions that are triggered by events and scale on demand. Major providers offer managed FaaS (AWS Lambda [3], Azure Functions [78], Google Cloud Functions [29]) while open-source alternatives (OpenFaaS [42], OpenWhisk [45], Fission [96]) enable self-hosting. As compositions grow, managing triggers and intermediate data becomes complex, motivating the use of complementary *workflow orchestrators* such as AWS Step Functions [4] and Azure Durable Functions [23].

Beyond FaaS, Container-as-a-Service (CaaS) [54] exposes container-level control with serverless-like operations. Kubernetes [30] and Knative [46] bring autoscaling, scale-to-zero, and eventing to containerized services. While these offerings reduce operational burden, their *stateless-by-default* model complicates the design of applications that require affinity, low-latency state access, or cross-invocation consistency.

Empirical studies revealed key characteristics and constraints of production FaaS. Wang et al. measured cold starts, resource variability, and isolation overheads across providers [115], while Shahrad et al. characterized large-scale provider workloads and highlighted opportunities for optimization [101]. Subsequent work explored microVMs [2], function caching and prewarming [47, 38], and provider mechanisms such as provisioned concurrency [9] to mitigate tail latencies. Beyond control-flow orchestration, recent work advocates data-centric orchestration where the movement and placement of data guide function execution [120]. Platforms like Nightcore [61] and systems research on low-latency isolates aim to better support interactive microservices under the serverless model.

2.3. Stateful Serverless

Recent work extends serverless beyond statelessness to support *stateful* functions and objects (e.g., [60, 123, 76]). These approaches primarily aim to address the stateless nature of the FaaS model, where the burden of managing application data, access control, and function workflows is often shifted to the developer through separate cloud services. As summarized

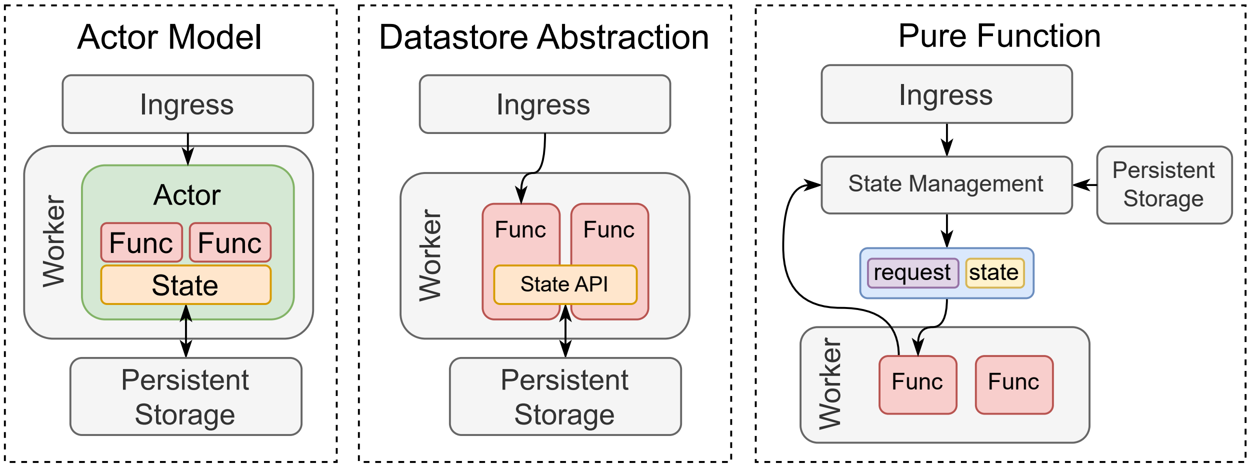


FIGURE 2.1. Comparison of three models for stateful serverless.

in Figure 2.1, three design patterns emerge based on state placement and access: (1) **actor model**, (2) **datastore abstraction**, and (3) **pure function**.

2.3.1. Actor Model.

In the actor model, the platform fetches the state from persistent storage and places (i.e., caches) it inside a worker instance, then dispatches the request to where the state resides to achieve data locality. This favors message-driven designs but can make maintainability difficult for bulky unstructured data, as the platform needs to balance each node's computing and storage aspects. The deployment granularity is an actor that contains multiple functions sharing the same state.

The foundational platform in this space is Orleans [24], which introduces "virtual actors". Its influence is evident in modern platforms such as Azure Entity Functions [79], which are part of Azure Durable Functions. While effective for certain use cases, this model can tightly couple state and compute within a single language and environment. The actor model approach has been popular in programming languages and object-oriented programming because it spurs asynchronous messaging across actors and lends itself to distributed deployments. In contrast, OaaS (Chapter 3) manages the object abstraction at the platform level, allowing functions implemented in different languages or runtimes to operate on an object, thereby offering greater flexibility.

2.3.2. Datastore Abstraction

The datastore abstraction is a hybrid approach where the platform provides an API for the function to access data on demand. Like pure functions, it relaxes the need for state and function co-location, but can utilize caching to preserve data locality. Several systems utilize this pattern. Cloudburst [108] uses a shared distributed key-value database, while Boki [60] provides API access to a distributed logging system. This log-based approach for consistency differs from OaaS's mechanisms (Chapter 3), which use a combination of fail-safe state transitions for unstructured data and localized locking to handle race conditions. Beldi [123], on the other hand, provides the database and transaction API to the state. While providing direct transactional APIs offers strong guarantees, OaaS abstracts these concerns from developers through its fail-safe versioning scheme and guarantees of exactly-once execution, aiming for a higher-level resilience model.

FAASM [103] optimizes the function-state interaction by using WebAssembly [49]. Although this enables multiple functions to share memory and achieve data locality, it requires compiling code to WebAssembly, which can limit library compatibility. Crucial [17] also explores shared-memory abstractions. OaaS's hybrid model avoids these limitations by combining the pure function and datastore abstraction approaches, offering broader language support.

2.3.3. Pure Function

In the pure function approach, state is placed on another system and transferred to the worker instance upon invocation, appearing as part of the function's input arguments. This disaggregates state management and computation for system design simplicity, but can compromise data locality. Apache Flink Stateful Function (StateFun) [11] is a representative solution based on this approach. OaaS (Chapter 3) combines the pure function approach for structured data with a lazy-fetch mechanism for unstructured data, practically employing both the pure function and datastore abstraction approaches.

2.3.4. Additional encapsulation patterns

Beyond the three primary models, additional patterns have emerged. Unified deployment units such as proclets [99] and recent actor surveys [107] further explore co-location of methods with state. Storage-resident execution such as Shredder [125] and Apiary [65] pushes computation into storage engines, inverting the traditional separation. OaaS (Chapter 3) spans these approaches: it provides an object API for encapsulation while retaining datastore and pure-function modes when appropriate, offering developers flexibility in choosing the right abstraction for their use case.

2.3.5. Process abstractions.

While OaaS raises the level of abstraction, a complementary line of work reduces abstraction to expose OS/process-like control for performance. Process-as-a-Service (PaaS) introduces a cloud process abstraction to unify elasticity with stateful execution [31]. For long-running services, Nu proposes migratable logical processes to achieve microsecond-scale resource fungibility [99]. These efforts align with OaaS in spirit—structuring control and state—but differ in the locus of programmability and who manages adaptation.

2.3.6. Serverless workflow and dataflow.

Beyond state management for individual functions, composing them into complex applications is another significant challenge. Commercial orchestrators such as AWS Step Functions [4] and Azure Durable Functions [23] can mitigate the burden of chaining events and managing control-flow; however, developers are often still required to navigate data between steps manually. The research community has proposed more integrated solutions. DataFlower [72] introduces a dataflow paradigm specifically for serverless workflow orchestration to optimize data movement between functions. Netherite [23] focuses on the efficient execution of these workflows through techniques like partitioned state management and collocated execution. While these systems significantly advance workflow performance and orchestration, OaaS (Chapter 3) approaches the problem from a different angle by integrating workflow directly into its core object abstraction. OaaS's dataflow functions allow developers to declaratively define a workflow as a directed acyclic graph (DAG) based on data dependencies rather than task dependencies. The platform transparently manages parallelism, data navigation, and state consistency, abstracting these complexities from the developer.

2.3.7. Consistency and fault tolerance.

A central theme across these systems, and a core focus of Chapter 3, is the consistency model offered for reads/writes across invocations and failures. Techniques range from at-least-once execution with idempotence, to transactional updates via logging and versioning (e.g., Beldi [123]), to object-centric consistency with conflict resolution via CRDTs and virtual actors [24, 102]. The design of OaaS adopts dataflow-aware state management and structured recovery to balance availability and correctness without imposing monolithic transactions.

2.4. QoS-Aware Serverless Computing

While serverless abstracts infrastructure, developers still need to meet *non-functional* requirements such as tail latency, availability, throughput, and cost. Emerging systems expose interfaces to declare QoS intents and automate the choice of runtime configurations (e.g., memory size, concurrency, placement, batching) and control policies (e.g., timeouts, retries, prewarming). Chapter 4 formalizes this as a contract between the developer and the platform and introduces a dynamic class runtime that materializes objects according to the declared QoS. An optimization loop closes the gap between observed and target metrics with minimal developer intervention, harmonizing with the consistency and fault-tolerance mechanisms in Chapter 3.

2.4.1. Non-functional requirements enforcement and prior art.

A large body of work targets latency and isolation symptoms of today's platforms. Cold-start mitigation spans ahead-of-time preparation and sandbox reuse (e.g., Catalyzer [38], Nightcore [61], and recent analyses [39]); lightweight isolation via microVMs improves performance and multi-tenancy [2]. Providers expose pre-allocation (e.g., provisioned concurrency [9]) to trade cost for predictability, while real-time serverless extends the API with rate guarantees [89]. Beyond single functions, Sequoia [111] proposes QoS-aware scheduling, Aquatope [127] reasons about uncertainty at workflow scope, and Astrea [59] automates analytic job configuration. Data-centric orchestration [120] further reduces coordination overhead by aligning placement and movement with data dependencies. These efforts motivate OaaS's declarative QoS interface and adaptive runtime.

2.5. Serverless Computing in Edge–Cloud Settings

Edge-to-cloud applications introduce intermittent connectivity, constrained resources, and locality-sensitive state access. Chapter 5 (EdgeWeaver) specializes OaaS for IoT by adding connectivity-aware invocation, edge affinity/placement policies, and state synchronization across edge and cloud. The design must reconcile eventual disconnections and updates with bounded staleness or conflict-resolution strategies while preserving the "serverless" developer experience. This positions objects as a natural unit for mobility, caching, and coordination across the continuum.

Research on FaaS across the continuum considers resource provisioning and placement [12, 114], QoS-aware function delivery [119], and consistency models adapted to geo-distribution [102]. Edge-oriented orchestration emphasizes data proximity and network variability; our EdgeWeaver design builds on these insights to enable connectivity-aware execution and reconciliation.

Two fundamental challenge threads recur in edge–cloud IoT deployments. First, *complexity and heterogeneity*: edge FaaS often runs on constrained platforms and differs from cloud-side stacks, leading to fragmented designs and duplicated concerns across tiers. Diverse devices and middleware add to configuration and lifecycle overheads, especially across multiple sites [90, 25, 13]. Second, *intermittent connectivity and consistency*: partitions and variable links are common at the edge; systems must balance consistency and availability in light of CAP [22, 67], choosing application-appropriate reconciliation or bounded staleness strategies without blocking urgent actions [89]. The following subsections examine these challenges in detail, providing the technical foundation for EdgeWeaver's design (Chapter 5).

2.5.1. Amplified Development Complexity in Edge-Cloud Deployments

The “function” abstraction in FaaS focuses solely on application logic, leaving data management and communication to developers. This results in fragmented implementations requiring complex interactions among FaaS invocations, databases, and orchestrators. Edge deployments compound this fragmentation: developers must manually integrate not only cloud services but also IoT devices, edge servers, communication protocols (MQTT, Kafka), and heterogeneous FaaS implementations (k3s at edge vs. k8s in cloud) [37].

Critically, FaaS’s stateless nature *lacks built-in support for long-lived functions*—essential for IoT scenarios involving frequent or continuous data processing [94]. Traditional FaaS platforms are designed around ephemeral, event-triggered executions that complete within seconds or minutes. However, IoT applications often require persistent connections to device streams, continuous monitoring of sensor data, or long-running analytics pipelines. This mismatch forces developers to orchestrate intricate dataflows across many short-lived functions and shared-state services, dramatically increasing system complexity and introducing coordination overhead.

Furthermore, large-scale IoT solutions span diverse technologies, application models, and communication protocols [90, 36]. While IoT middleware provides abstraction layers (e.g., device agents managed by fleet managers) for easier administration [82], these layers add management overhead. Even with a single edge node, the number of components developers must implement and manage nearly doubles compared to cloud-only architectures. With multiple edge sites, this complexity escalates as each node may differ in software stacks (different versions of Kubernetes or container runtimes), capacity (CPU, memory, storage constraints), network conditions (bandwidth, latency, jitter), and policies (security requirements, data sovereignty rules) [25, 13]. Developers must account for these variations when designing, testing, and deploying applications, leading to fragile configurations that break when assumptions about the underlying infrastructure change.

2.5.2. CAP Theorem and Application-Specific Trade-offs

The Edge-Cloud continuum often suffers from *intermittent connectivity* between tiers, caused by network congestion, physical obstructions (buildings, terrain), power limitations, or fluctuating bandwidth [43]. Lost or delayed connections (*network partitions*) disrupt data flow, causing the cloud's view of application state to become stale while edge nodes continue processing locally. In time-sensitive applications like industrial automation or healthcare, these inconsistencies have serious consequences: unsynchronized production line data may result in incorrect machine behavior, and outdated patient information could lead to delayed or dangerous clinical decisions [1]. The CAP theorem [22, 67] formally shows that distributed systems cannot simultaneously guarantee *consistency*, *availability*, and *partition tolerance* under network partitions. Edge-Cloud IoT deployments must balance these trade-offs based on application-specific QoS needs: time-sensitive applications (industrial automation, emergency response) prioritize availability, accepting temporary inconsistency, while financial transactions demand strong consistency even if operations are delayed.

Current platforms lack mechanisms for developers to declaratively specify these NFR trade-offs. For example, an intelligent transportation system detecting an accident should immediately dispatch an ambulance rather than waiting for cloud acknowledgment [89], yet developers must implement custom distributed protocols (two-phase commit, vector clocks, operational transformation) from scratch, requiring deep expertise in distributed systems. EdgeWeaver addresses these challenges by providing connectivity-aware execution, object-centric state synchronization, and policy-driven placement that allow developers to declaratively specify consistency and availability requirements without implementing low-level coordination protocols.

2.6. Positioning of this Dissertation

The literature reviewed in this chapter reveals both the promise and limitations of current serverless computing paradigms. While FaaS has democratized cloud application deployment, its stateless-by-default model burdens developers with state management complexities. Stateful serverless systems have emerged to address this gap, yet they often impose trade-offs: actor models tightly couple state and compute, pure function approaches sacrifice data locality, and datastore abstractions require explicit state navigation. Furthermore, existing systems typically focus on either functional correctness *or* performance optimization, rarely integrating both concerns within a unified abstraction. Finally, the extension of serverless to edge–cloud environments remains fragmented, with most solutions addressing edge deployment as an afterthought rather than a first-class design consideration.

This dissertation addresses these limitations by advancing serverless computing along three complementary dimensions. First, we introduce *Object as a Service* (OaaS), a paradigm that unifies stateful computing, dataflow orchestration, and fault tolerance within a single object-oriented abstraction (Chapter 3). Unlike prior actor-based or pure-function approaches, OaaS manages object abstractions at the platform level, supporting both structured and unstructured data through fail-safe state transitions and exactly-once execution semantics. The platform transparently handles consistency, recovery, and dataflow dependencies, freeing developers from low-level coordination concerns.

Second, we extend OaaS with a declarative interface for non-functional requirements, enabling developers to specify quality-of-service (QoS) objectives—such as latency bounds, throughput targets, and cost constraints—as first-class contracts (Chapter 4). A dynamic class runtime materializes objects according to these declarations, continuously optimizing resource configurations and execution strategies through feedback-driven adaptation. This approach harmonizes functional and non-functional concerns within the same abstraction, avoiding the fragmentation between application logic and performance tuning prevalent in current systems.

Third, we specialize OaaS for the edge–cloud continuum through OaaS-IoT and its realization in the EdgeWeaver platform (Chapter 5). Rather than treating edge deployment as a variant of cloud execution, EdgeWeaver provides connectivity-aware invocation, edge-affinity placement policies, and object-centric state synchronization that explicitly account for intermittent connectivity, resource constraints, and geo-distribution. This positions objects as natural units for mobility, caching, and coordination across heterogeneous edge and cloud tiers.

Together, these contributions establish a cohesive framework that raises the level of abstraction in serverless computing while maintaining the flexibility, performance, and reliability required by modern distributed applications. The following chapters detail the design, implementation, and evaluation of each component, demonstrating how this integrated approach addresses the challenges identified in the literature while opening new directions for cloud application development.

2.7. Summary

This chapter surveyed serverless computing research across stateful systems, QoS-aware platforms, and edge-cloud deployments, revealing fundamental limitations that motivate the Object-as-a-Service paradigm. While stateful serverless approaches—actor models, pure functions, and datastore abstractions—advance beyond stateless FaaS, they fragment state management, workflow orchestration, and quality-of-service control across competing abstractions. QoS-aware systems target specific symptoms like cold starts and provisioned concurrency but lack integrated mechanisms for declarative NFR specifications and automated adaptation. Edge-cloud extensions introduce additional complexity from intermittent connectivity and resource heterogeneity, yet current platforms lack declarative mechanisms for application-specific consistency-availability trade-offs mandated by the CAP theorem.

This dissertation addresses these gaps through three integrated contributions: OaaS (Chapter 3) unifies stateful computing, dataflow orchestration, and fault tolerance within a platform-managed object abstraction; declarative NFR interfaces (Chapter 4) enable developers to specify QoS objectives with dynamic runtime optimization; and EdgeWeaver (Chapter 5) specializes OaaS for edge-cloud continuum through connectivity-aware execution and object-centric state synchronization. Together, these contributions establish a cohesive framework advancing serverless computing beyond current fragmentation while preserving developer simplicity.

CHAPTER 3

OBJECT AS A SERVICE (OAAS): ENABLING OBJECT ABSTRACTION IN SERVERLESS CLOUDS[1]

3.1. Overview

The empirical findings resulted from our market survey, presented in Chapter 1, revealed that development complexity and operational maintainability constitute the most pressing challenges in cloud-native adoption, particularly for organizations with limited technical expertise. Current serverless platforms, despite their promise of simplified deployment, suffer from fundamental fragmentation: application logic resides in Function-as-a-Service (FaaS) runtimes, state management requires separate database services, and workflow orchestration demands additional coordination layers. This fragmentation forces developers to master multiple abstractions, manually integrate disparate services, and navigate complex inter-service communication patterns—directly contradicting the serverless vision of infrastructure abstraction.

This chapter introduces Object as a Service (OaaS), a unified paradigm that consolidates resource, state, and workflow management into a single object-oriented abstraction. By borrowing established object-oriented programming (OOP) concepts—classes, methods, attributes, inheritance, and polymorphism—OaaS enables developers to define entire applications through familiar constructs while the platform automatically handles deployment, scaling, and state management. We present Oparaca, an open-source prototype implementation that demonstrates how this unified abstraction streamlines cloud-native development without sacrificing performance or scalability. The chapter is organized as follows: Section 3.2 defines the OaaS paradigm and its conceptual model; Section 3.3 describes the

[1]This chapter is based on and includes material from the following publications: (1) P. Lertpongrujikorn and M. Amini Salehi, "Object as a Service: Simplifying Cloud-Native Development through Serverless Object Abstraction," *IEEE Transactions on Computers*, accepted Oct. 2025, In Press; and (2) P. Lertpongrujikorn and M. Amini Salehi, "Object as a Service (OaaS): Enabling Object Abstraction in Serverless Clouds," in *Proceedings of the 16th IEEE International Conference on Cloud Computing (IEEE CLOUD '23)*, Chicago, IL, USA, 2023, pp. 238–248. Reprinted and adapted here with permission from IEEE.

Oparaca architecture, including class deployment, function invocation, and data management strategies; Section 3.4 discusses security, multi-tenancy, and cold start considerations; and Section 3.5 evaluates Oparaca through experiments demonstrating that OaaS imposes negligible overhead while significantly simplifying development workflows.

3.2. Object as a Service (OaaS) Paradigm

3.2.1. Conceptual Modeling of OaaS

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

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

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

Fourth, OaaS enables higher-level abstractions by allowing cloud objects to be nested, where a high-level object references lower-level objects. Functions can use these references to fetch inputs or invoke dataflow functions (called *macro functions*) that chain operations across lower-level objects. Unlike traditional FaaS workflows [4], macro functions determine execution flow based on dataflow rather than task (i.e., function call) dependency. Developers only define the data flow, while OaaS manages parallelism, data navigation, and state

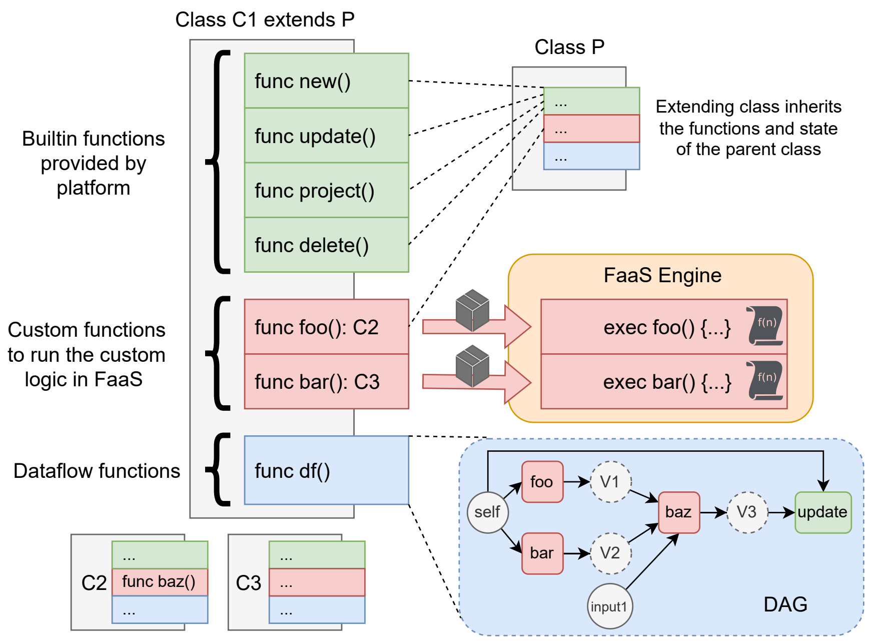


FIGURE 3.1. Different types of functions supported by OaaS.

consistency transparently.

3.2.2. Developing Classes in OaaS

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

As shown in Figure 3.1, OaaS supports three function types. First, *built-in* functions that are provided by the platform. These functions could be the standard functions such as `CRUD` (create, read, update, and delete), which are the common data manipulation operations. The platform manages the execution of these functions without intervention from the developer. Second, *custom* (a.k.a. *task*) functions that are developed by developers (OaaS users) to provide their business logic. To handle the invocation of these functions, OaaS employs existing FaaS engines in its underlying layers to exploit their auto-scaling and scale-to-zero capabilities. Third, *dataflow* (macro) functions are defined as a DAG representing the chain of invocations to objects.

As an example of package definition, Listing 3.1 represents a declaration example for a package that includes one class called `video` that has a state named `mp4` (Line 6), *built-in* function named `new` (Line 9), and *custom* function named `transcode` (Line 1). The state `mp4` refers to video data that is unstructured data. The class has a public *custom* function called `transcode`. The definitions of the *custom* function are declared in Lines 15—17. The `type` of a function (Line 16) can be a `task` (or a `macro`, as noted earlier). This function creates another object instance of type `video` as an output. Line 17 declares the container image URI for executing function code.

Listing 3.1. An example simplified script that declares `multimedia` package with a `video` class, and a `transcode` function for it in the `YAML` format.

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

3.3. Oparaca: A Platform for the OaaS Paradigm

3.3.1. Design Goals

The Oparaca platform is designed with its foundational goal of providing object abstraction with two additional design goals: *backward compatibility* and *extensibility*. *first*, while OaaS simplifies cloud-native application development, it is not always a replacement for FaaS; thus, Oparaca supports stateless FaaS and direct data access to storage systems. *Second*, for extensibility, Oparaca decouples the control plane from the execution plane, allowing the execution plane to operate independently via standardized APIs. This platform-

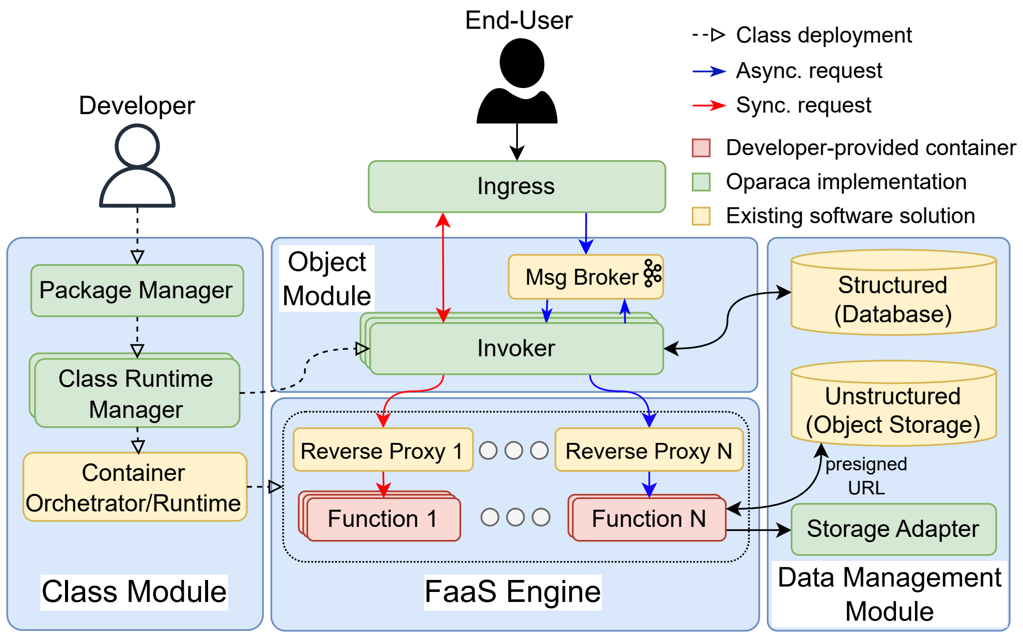


FIGURE 3.2. A bird-eye view of the Oparaca architecture. Dashed lines show actions of the developer defining classes and objects, and solid lines show actions using objects and invoking functions.

agnostic design accommodates various execution planes optimized for specific use cases, such as latency-constrained function calls [105] or access to hardware accelerators [118].

3.3.2. Overview of the Oparaca Architecture

The Oparaca platform is designed based on multiple self-contained microservices that communicate within a serverless system. Figure 3.2 provides a birds-eye view of the Oparaca architecture that is composed of five modules:

- **Class Module** serves as the interface for developers to create and manage classes and their functions.
- **Object Module** serves as the cornerstone of Oparaca that has two main objectives: (a) providing the "object access interface" for the user application to access an object(s); and (b) offering the object abstraction while transparently handling function invocation and state manipulation.
- **FaaS Engine** is the underlying execution engine of Oparaca, which can be any existing

FaaS system (e.g., Knative).

- **Data Management Module** is to manage object data persistence via employing database (e.g., document database) and object storage (e.g., S3-compatible storage). To bind these storages to the functions, the Invoker abstracts data access for structured data, while the Storage Adapter is employed to handle access to unstructured data in the object storage
- **Ingress Module** whose purpose is to provide a single end-point for the user application.

Details of these modules, their interactions, and how they fulfill the consistency and fault-tolerance objectives (described in Section 3.1) are elaborated in the following subsections.

3.3.3. Class Module

To define a new class and its functions in Oparaca, the developer defines them as a package definition and registers it to the *Package Manager*, shown in Figure 3.2. Upon successful package validation by the Package Manager, the *Class Runtime Manager* (termed *CRM* for brevity) performs the class registration process that includes two operations:

(**a**) Informing the Object Module about the new/updated class. Upon receiving a class registration, the Object Module creates a handler instance to be prepared for handling object invocation. We elaborate on this process in Section 3.3.4.

(**b**) Registering the custom functions of the new class in the FaaS engine for future invocation. Recall that we aim to make Oparaca agnostic from the underlying FaaS engine. We design Oparaca to host a dedicated CRM for each FaaS engine. Accordingly, a new FaaS engine can be integrated into Oparaca by simply plugging its dedicated CRM into the system. When a function registration event occurs, the corresponding CRM processes this event by translating the function configuration into the specific format for that engine (e.g., Knative) and forwards it. Consequently, the underlying FaaS Engine creates the actual function runtime to be invoked by the Object Module.

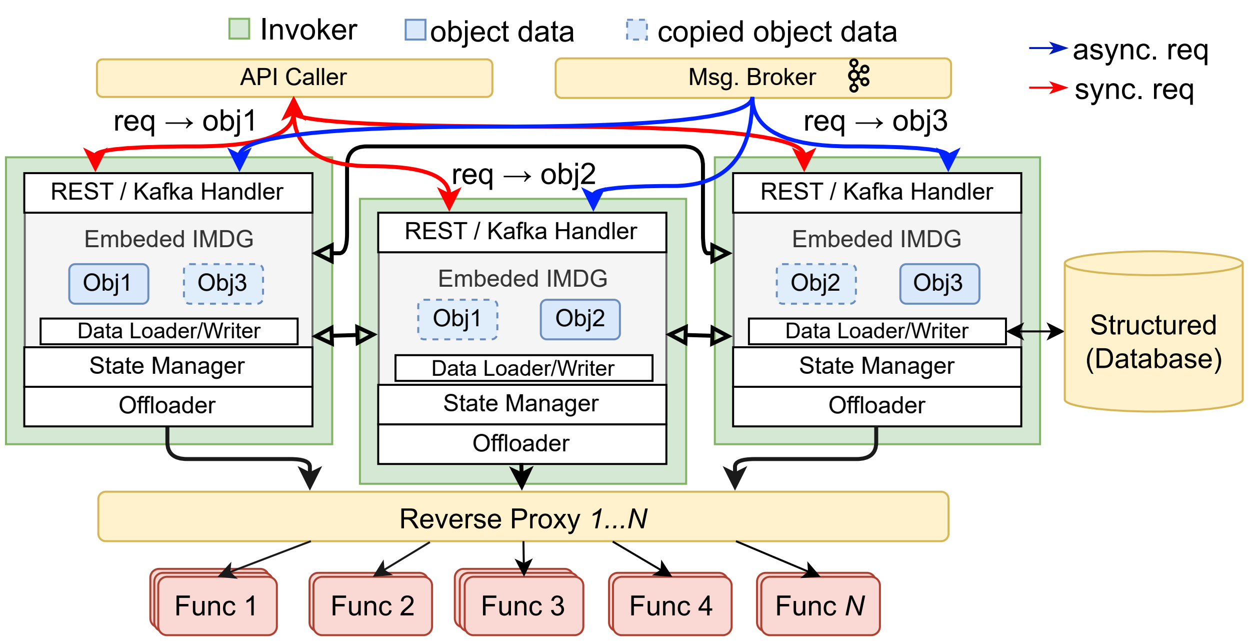


FIGURE 3.3. The cluster of Invokers replicates and distributes object data across the cluster via IMDG with consistent hashing. The invoker offloads the invocation to a corresponding FaaS function.

3.3.4. Object Module and FaaS Engine

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

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

In the Oparaca architecture (Figure 3.2), the mechanism for handling invocation and state management is managed by the *Invoker* component. In particular, to offload the object invocation to the FaaS engine, Invoker bundles the request and the related structured object data as a "*task*", as described in the next part, and passes it to the associated FaaS engine for execution.

Task Generation in the Invoker. Upon receiving a function call, the Invoker bundles the invocation request and associated object data into the task and offloads it to be executed on the FaaS engine. To further reduce the data transfer overhead of providing the object abstraction in the task generation process, we design Invokers to maintain the object data (i.e., state and metadata) in a distributed hash table [51], thereby reducing the cost of data transfer in a scalable manner. As shown in Figure 3.3, we equip each Invoker instance with an embedded in-memory data grid (IMDG) [122]. IMDG partitions the entire data space into multiple segments and distributes them across Invoker instances. The Invoker with IMDG determines the segment for a given object by consistent hashing of the object ID and assigns the object data to the selected segment. Similarly, to retrieve the object data, IMDG determines the owner of the data and then fetches it from the owner of the segment in one hop.

Unstructured Data Accessing. To minimize the overhead of accessing unstructured data, Oparaca allows function code to access the unstructured data on-demand and directly through a *presigned URL* and *redirection* mechanism. The presigned URL is the specific HTTP URL that includes the digital signature in query parameters to grant permission for anyone with this URL to access the specific data without the secret token. When a function needs to access the unstructured data, it sends an HTTP request to the storage adapter to receive the redirection response that points to the presigned URL of specific state data. Then, the function code can fetch the content directly from object storage via the given presigned URL. In addition to minimizing the overhead, using the presigned URL is important in protecting the function container from unauthorized access to other objects' data by analyzing their URL patterns.

Task Completion. After the FaaS engine completes the task, it sends the task completion data to the Invoker to update the state. If the function reports a failed task, the state remains unchanged. Otherwise, the Invoker updates the object data in IMDG and then writes it to the persistent database immediately or asynchronously. If an invocation involves both structured and unstructured states, we use pure and datastore techniques together, which can potentially lead to "state inconsistency challenges". We address this challenge in section 3.3.6.

Synchronous and Asynchronous Invocation. As mentioned in Section 3.3.1 and shown in Figure 3.3, we designed Oparaca to offer synchronous and asynchronous function invocations. In synchronous mode, the function is executed immediately upon invocation and returns the result to the caller. Meanwhile, in asynchronous mode, the invocation ID is provided to the caller as a reference so they can check the invocation result later. The request is placed into the message broker to be reliably processed at a later time. To accommodate both modes, the Invoker utilizes the handler instance to accept the invocation request for either the REST API (synchronous) or the message broker (asynchronous). Subsequently, the handler instance forwards the request to be processed in the same way by the other part of the Invoker.

3.3.5. Ingress Module

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

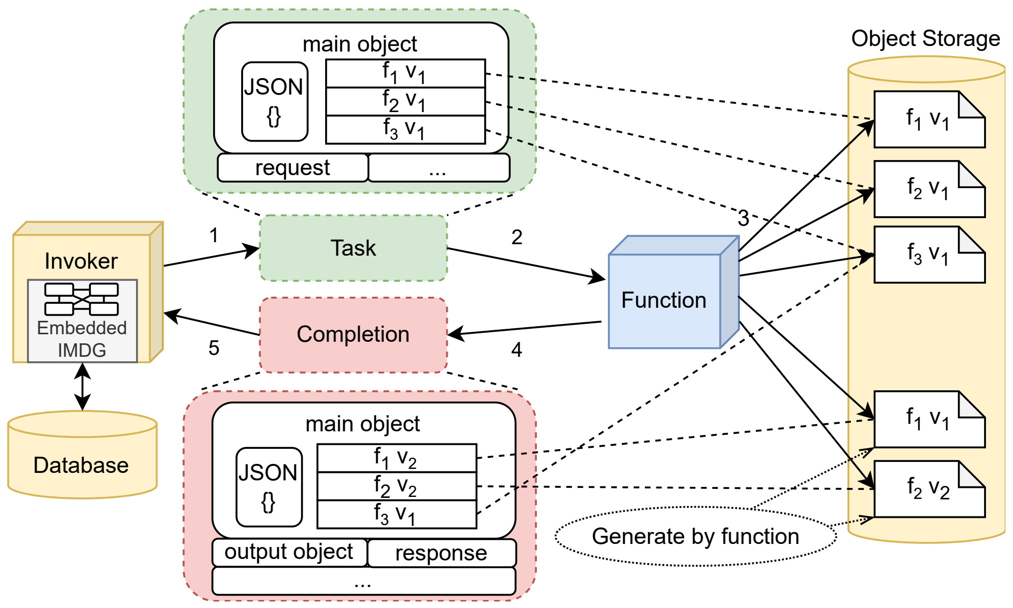


FIGURE 3.4. The process of offloading invocation task into the function runtime. Invoker bundles the request input and object state into a task and offloads it to the function to be executed. With *fail-safe state transition*, when the function needs to update the file in object storage, it creates a new file and updates the corresponding version ID via the returning completion message.

to access the data in its memory.

3.3.6. Resilience Measures of Oparaca

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

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

Maintaining data consistency across two data storages implies guaranteeing both storages are either successfully updated or fail for the same invocation. Otherwise (i.e., if only one of them succeeds), it leads to data inconsistency. To overcome this problem, we develop the *fail-safe state transition* mechanism that disregards the data update in the object storage if Invoker fails to update the structured part of the object data in the structured database. For that purpose, the mechanism uses a two-phase versioning scheme to keep track of the unstructured data. As shown in Figure 3.4, in the first phase, the mechanism creates a version ID for each file (unstructured data) and keeps them as structured data (metadata of object data) to track the current version of the file. In the second phase, which occurs upon function completion, Invoker changes all version IDs associated with the updated files (unstructured data) and then writes them to IMDG and the structured database.

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

Resilience against race condition. Race conditions in Oparaca can occur when multiple invocations modify the same object data simultaneously, resulting in potential data inconsistency. One way to prevent this issue is by using database transactions; however, this method lacks abstraction as it allows direct function code access to the database and is tightly dependent on the type of database. An alternative approach to avoiding race conditions is the cluster-wide pessimistic locking mechanism to synchronize the locking state for all invokers. Nevertheless, this approach necessitates additional network communication to coordinate the locking state, which can lead to scalability issues. Alternatively, we develop an improved version of this mechanism, called "localized locking," which relies on consistent hashing to direct the invocation request to the invoker that owns the primary copy of the targeted object data. Each invoker will only need to lock the object locally without additional network communication, making it more scalable than the cluster-wide version. Additionally, our localized locking approach guarantees that requests to the same object are executed in the arrival order, which is necessary in certain use cases where order matters, such as seat reservations. This is difficult to achieve with cluster-wide locking.

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

To achieve the *exactly-once* guarantee, we have to prevent three sources of the problem that are: (a) losing messages, (b) duplicating messages, and (c) processing messages more than once. Message brokers with stable storage (e.g., Kafka [66]) have features that can be leveraged to address these problems. To solve the first problem, upon failure occurrence, the Invoker can detect and reprocess the incomplete request using an offset number that is automatically generated by the message broker. The offset number is the auto-incremental number based on the message's arrival order and can be used to track the message's position in the queue. The second problem of producing duplicated request messages can be resolved using the message broker's "idempotent producer" feature.

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

3.3.7. Dataflow Abstraction in Oparaca

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

Upon calling the dataflow function, one of the Invokers takes on the role of orchestrator, similar to the orchestrator pattern [98] in microservices. It breaks down the dataflow into multiple lower-level invocations and forwards them based on the topological order of DAG. Using consistent hashing, the invoker can determine the address of the target object and send the request directly to another Invoker that holds the target object. When each step is completed, the orchestrator keeps track of the intermediate dataflow state to transparently operate the data exchange between invocation steps. With the orchestrator pattern, the dataflow control logic is centralized into a single invoker, simplifying the management, monitoring, and error-handling implementation.

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

3.4. Discussion of other Concerns

3.4.1. Security

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

3.4.2. Multi-tenancy

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

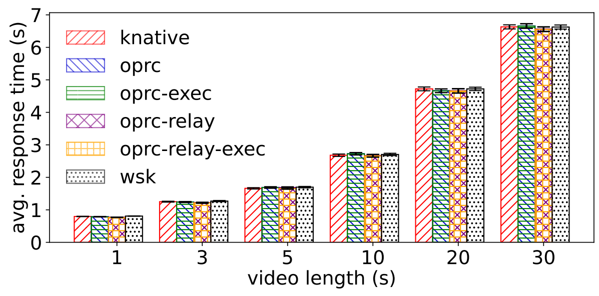
3.4.3. Cold Start

The developer functions and the Oparaca components can benefit from scale-to-zero to reduce the cost when there is no usage. However, this has the side effect of causing more cold starts. Since Oparaca components are shared across functions, we can effectively keep it warm to eliminate the additional cold start impact. In such a case, the cold start performance entirely depends on the underlying serverless execution engine.

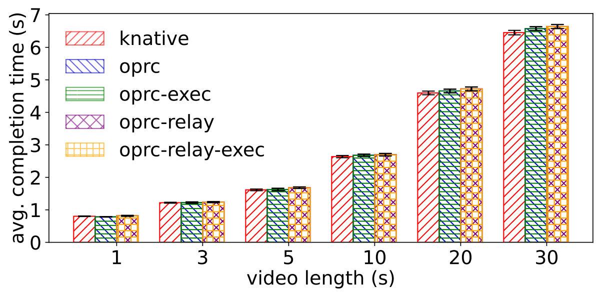
3.5. Performance Evaluation

3.5.1. Experimental Setup

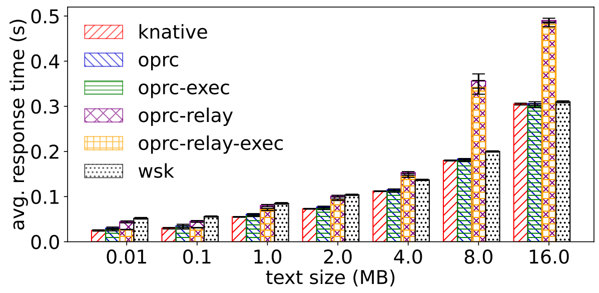
We deploy the Oparaca platform on 4 machines of Chameleon Cloud [64], each with 2 sockets of 24-Core Intel(R) Xeon(R) Gold 6240R CPU processors that collectively have



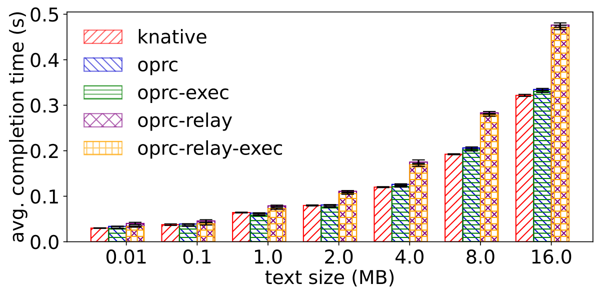
(A) Video transcoding (sync)



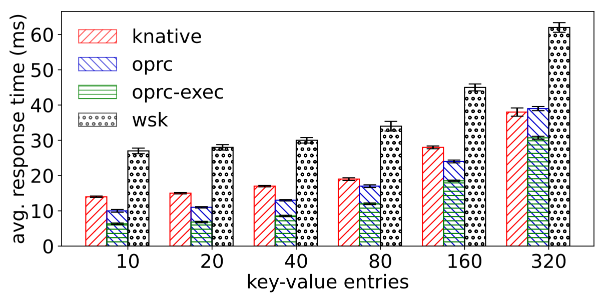
(B) Video transcoding (async)



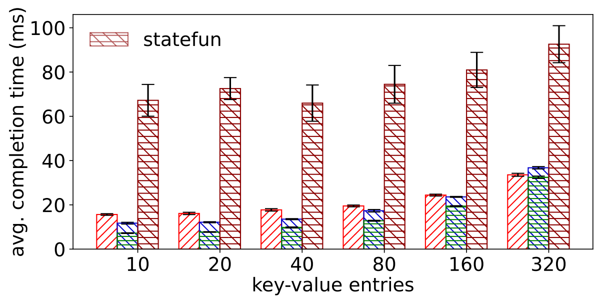
(C) Text concatenation (sync)



(D) Text concatenation (async)



(E) JSON update (sync)



(F) JSON update (async)

FIGURE 3.5. The average execution time of functions for objects with various state sizes in synchronous and asynchronous invocations. Two versions of Oparaca are examined: the full version and the version without URL-redirection (*oprc-relay*). We also capture the time used by the internal Knative in both Oparaca versions and show them with the suffix -*exec* and plot them in the same bar as their Oparaca version.

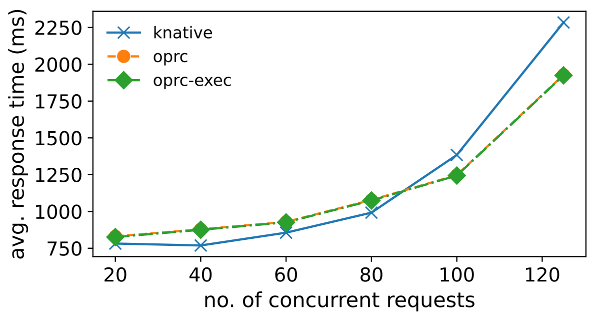
192 cores, 768 GB memory, and SSD SATA storage. We set up the Kubernetes cluster, which includes 15 VMs with 16 vCPUs and 32 GB of memory. We made another 2 VMs for the S3-compatible storage (Minio [56]) for unstructured data and ArangoDB ([55]) for structured data. Oparaca is implemented using Java with Infinispan [57] for IMDG. The source code is available at `https://github.com/hpcclab/OaaS`.

Baselines. We configure Apache Flink Stateful Function (StateFun) [11], OpenWhisk [45], and Knative [46] to serve as the baselines. Unlike Oparaca and OpenWhisk, which focus on API calls and event handling, StateFun is an open-source stateful serverless system focusing on stream processing. Because StateFun does not manage the function worker instances out of the box, we configure Knative to complement it. OpenWhisk and Knative are popular open-source stateless FaaS platforms that we use to represent the state management done by the developer.

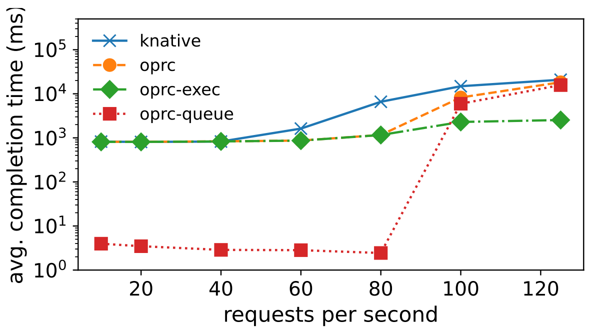
We used Gatling[32] for load generation and implemented three applications to serve as the workload. First is the video transcoding function, which utilizes FFmpeg [121], a CPU-intensive application. The second is a text concatenation function that concatenates the content of a text file (state) with an input string. This function represents a highly IO-intensive workload. Third is the JSON update function, which uses only structured data in JSON and is used to insert key-value pairs into the JSON state data randomly. The remaining workload characteristics are specific to each experiment and are explained in the respective sections. All three functions are implemented in the Python language.

3.5.2. Analyzing the Imposed Overhead of Oparaca

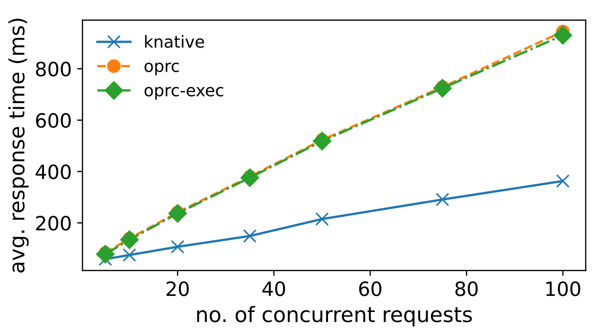
The abstractions provided by Oparaca are not free of charge and introduce some time overhead to the applications using these abstractions. In this experiment, our aim is to measure this overhead and see how the efficient design of Oparaca can mitigate this overhead. The latency of a function call is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



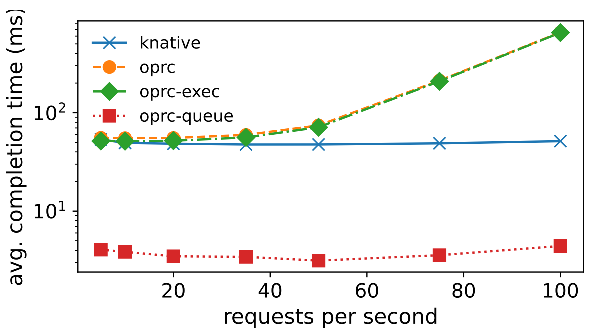
(A) Video transcoding function (sync)



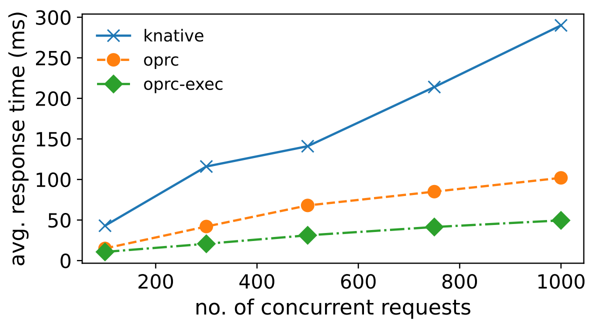
(B) Video transcoding function (async)



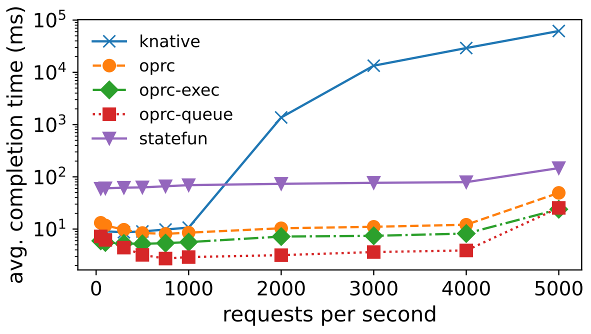
(C) Text concatenation function (sync)



(D) Text concatenation function (async)



(E) JSON update function (sync)



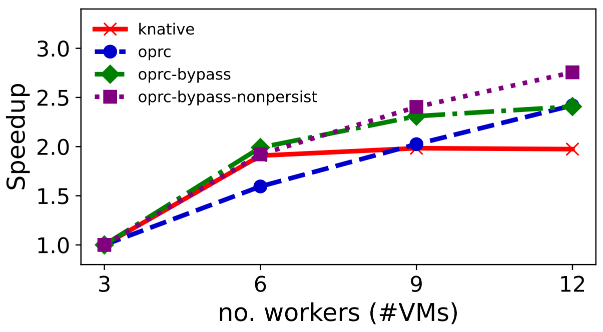
(F) JSON update function (async)

FIGURE 3.6. The average completion time of functions upon varying the rate of incoming requests in synchronous and asynchronous invocations. *oprc-queue* is the queuing time that requests stay within the message queue

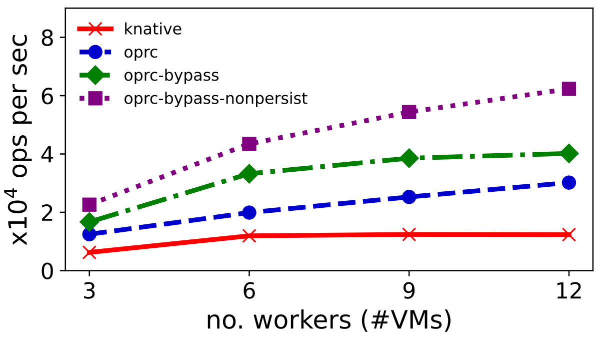
The impact of changing the state size. As shown in Figure 3.5, to generate objects with various state sizes, we increased the input video length from 1—30 seconds. To remove the impact of video content on the result, the longer videos were generated by concatenating the same 1-second video. Similarly, the text files are from 0.01—16 MB. For the JSON object, the key and value sizes are 10 and 40 bytes, respectively, and the number of key-value pairs varies from 10—320 pairs. To concentrate only on the overhead of data access and avoid other sources of overheads, we configure Gatling to assign only one task at a time and set it to repeat this operation 100 times. To analyze the improvements offered by the URL redirection, we examine two versions of Oparaca: the full version (expressed as *oprc*) and without URL redirection (expressed as *oprc-relay*). The horizontal axes represent different state sizes for video, text, and JSON, respectively, and the vertical axes represent the average response/completion time (latency).

In Figure 3.5, the average task execution time increases for larger state sizes. For the video transcoding function, all of the platforms perform with similar latency, which is expected because of the compute-intensive nature of the video transcoding that dominates the completion time. In a text concatenation function, however, Knative performs slightly better than Oparaca because of the overhead of unstructured state access by the redirection of the presigned URL. However, if we compare Oparaca with another version that uses a relay mechanism to provide the state abstraction, it performs much lower than its alternative with an average of 30% lower response time. Lastly, we can see all the described trends happen similarly for synchronous and asynchronous request types.

In the JSON update function (Figures 3.5e and 3.5f), Oparaca can perform with lower latency than Knative because the function does not need to fetch the object data from the database because of the pure function semantic. Nevertheless, Knative can catch Oparaca by increasing the key-value entries to 320. The reason is that the gain from eliminating the database connection is surpassed by the overhead of moving the data to the function code for larger records. OpenWhisk and Knative have the same pattern because both of them are FaaS, but OpenWhisk performs significantly worse. In Figure 3.5f, the Statefun shares the same pattern with Oparaca with the consistent gap because it also relies on local storage to keep the function state without the need to fetch the data from the database. We also observe that Statefun performance degraded compared to our initial results [70]. This is because the storage hardware being used for the experiment has a lower through, which impacts the performance of Statefun.



(A) Speedup results by horizontal scaling



(B) Throughput results from horizontal scaling

FIGURE 3.7. Evaluating the scalability of the OaaS platform against other baselines.

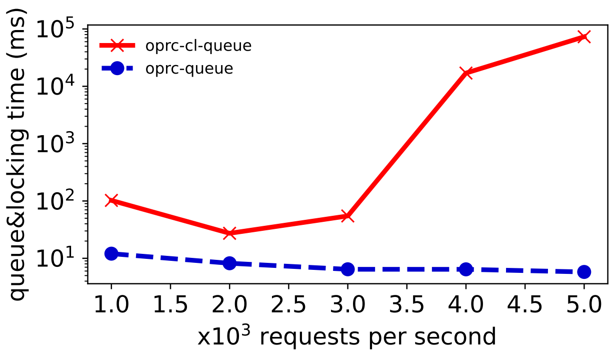


FIGURE 3.8. Evaluating the performance of localized locking compared to cluster-wide locking

The impact of concurrent function invocations. on the Oparaca overhead is shown in Figures 3.6. In synchronous invocation, we increase the number of concurrent invocations of the same function (horizontal axes), whereas, for asynchronous invocation, concurrency depends on the system implementation which cannot be forced directly; thereby, we use the request arrival rate to increase the concurrency of invocations. To remove the impact of any randomness, We disabled the auto-scaling and limited the number of worker instances to 6. We also exclude OpenWhisk from this section because the Python runtime in OpenWhisk does not support container-level concurrency.

For the transcoding function (Figures 3.6a and 3.6b), at the low concurrency levels (< 80 invocations), Oparaca has average response times higher than Knative, but for the higher concurrency levels, the response time of Knative grows faster than Oparaca due to computing resource limitations. Oparaca doesn't need to fetch video file metadata, giving it an edge at high concurrency. In the concatenation function (Figures 3.6c and 3.6d), however, this phenomenon does not happen. The difference is that text concatenation is IO-intensive and desires high network bandwidth. The overhead of unstructured data access overwhelms the performance gain from eliminating structured data fetching.

For the JSON update function (Figures 3.6e and 3.6f), Oparaca can effectively reduce the latency by eliminating the need to fetch from the database. In Figure 3.6f, because Statefun also shares this invocation scheme and, therefore, offers less completion time than Knative. However, since it relies on local storage to keep the state, while Oparaca uses the memory, Statefun's completion time is higher than Oparaca's.

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

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

3.5.3. Scalability of the Oparaca Platform

To study the scalability, we scale out the Kubernetes workers from 3—12 VMs, each with 16 vCPU cores (in total 48—192 vCPUs). We measured throughput and speedup metrics, focusing on the JSON update function, which does not rely on slow object storage, which becomes the bottleneck of this experiment. We measure the throughput by continually increasing the concurrency until the throughput stops growing (Figure 3.7b). We assume three VMs as the base speedup=1, and the speedup of other cluster size is calculated with respect to the base value. Moreover, we add two other versions of Oparaca: first is *oprc-bypass* that uses a standard Kubernetes deployment as its underlying function execution instead of Knative; Second is *oprc-bypass-nonpersist* that does not persist the object data to the database to measure if Oparaca is not bottleneck by the database write operation.

According to Figure 3.7a, the speedup of Knative plateaus after reaching 6 VMs. We realized that this plateau is attributed to the database write operation throughput bottleneck. Conversely, Oparaca exhibits the potential for higher speedup enhancement due to its reliance on the distributed in-memory hash table to consolidate data for batch write operations. This approach can boost maximum throughput by up to 3.27× when comparing *oprc-bypass* with *knative*.

Figure 3.7b shows that *oprc-bypass* yields a higher throughput over the baseline Oparaca. This is because Oparaca sends task data through the Knative internal proxy to offload the task to Knative. While this setup allows for scale-to-zero functionality, bypassing these components leads to even higher throughput. Furthermore, by disabling the database writing operation, which is the bottleneck, *oprc-bypass-nonpersist* can achieve even higher throughput. Although there isn't linear scalability due to the limitations of the database write performance, Oparaca significantly improves maximum throughput compared to traditional FaaS systems.

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

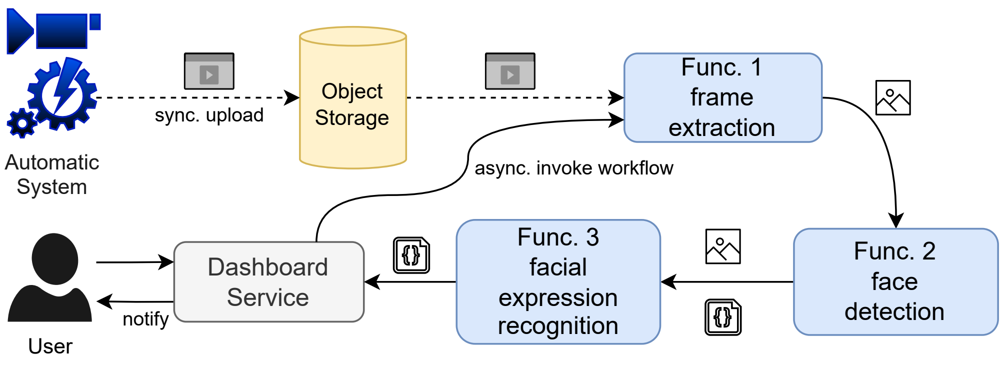


FIGURE 3.9. The use case of developing a face expression recognition workflow for an input video.

3.5.4. Performance of Localized Locking

To analyze effectiveness of localized locking, we created a variation of Oparaca, called *oprc-cl*, that has cluster-wide locking and evaluated it using a cluster of 12 Invokers while increasing the request arrival rate to measure the locking overhead. To generate requests involving the locking mechanism, we created multiple requests targeting the same object. From Figure 3.8, the overhead of localized locking remains mostly constant, while the overhead of cluster-wide locking rises for higher request rates. The cluster-wide version does not exhibit this behavior, as the network communication overhead limits the throughput and hinders high invocation rates.

3.5.5. Case Study: Development Efficiency Using OaaS

In this part, we provide a real-world use case of object development using OaaS and its FaaS counterpart and then demonstrate how OaaS makes the development process of cloud-native serverless applications easier and faster.

Case Study # 1. expression detection system.. This case study is a video processing workflow that performs face detection and facial expression recognition. Figure 3.9 shows the automatic system uploads the video file to the object storage to be processed by the workflow of functions. The workflow includes: `Func_1` to split the video into multiple image segments; `Func_2` to detect the face on each sample image frame; and `Func_3` to perform facial expression recognition on the detected face image and generate a `JSON` format label.

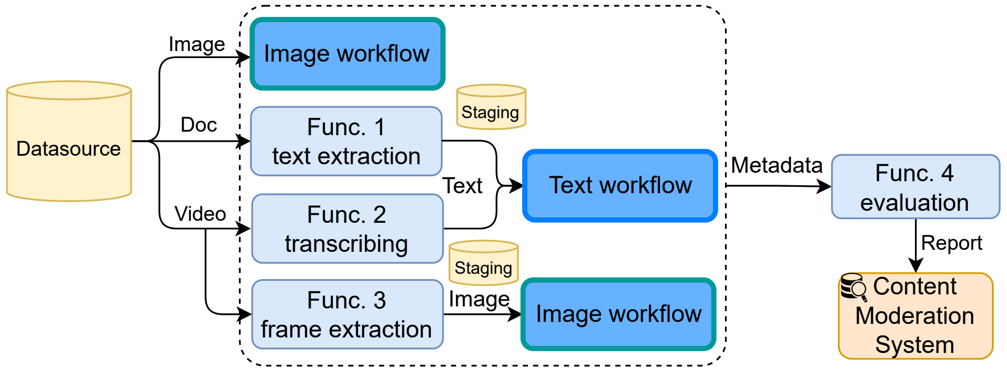
These functions must persist their output object so that the next function in the workflow can consume it.

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

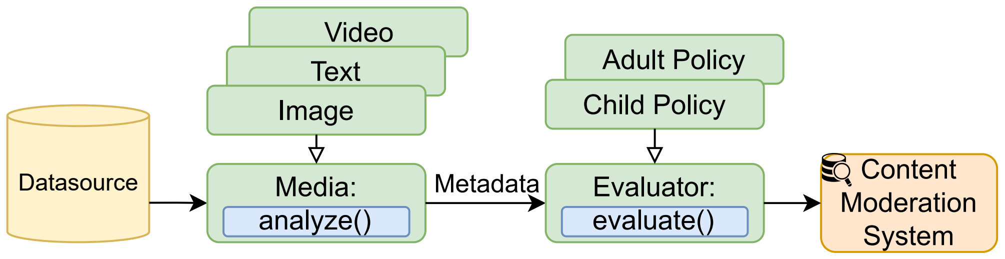
OaaS implementation. The developer defines three classes, namely `Video`, `Image`, and `Expression`, in the form of the three following classes: (a) `Video` class with `frame_extract()` functions; and a macro function, `df_exp_recog(detect_interval)`, that includes the whole dataflow of function calls,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

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

FaaS implementation. To simplify complex multimedia processing workflows [10], we abstract the workflow to extract the metadata from the image files as `Image workflow`, and the workflow to extract metadata from the text files as `Text workflow` as shown in Figure 3.10a. Before using both workflows, the content must be pre-processed to extract



(A) FaaS-based



(B) OaaS-based

FIGURE 3.10. The automatic content moderation system.

raw data via using the pertinent FaaS functions: (a) `text extraction` to extract text from the document. (b) `transcribing` to extract text from the video. (c) `frame extraction` to sample image frame from the video. After feeding the data into both workflows to extract the metadata, we use the `evaluation` function to generate a report to the content moderation system.

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

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

- **Workflow Construct.** We encapsulate related FaaS functions and states into classes representing two key functionalities: `Media` to extract the metadata and `Evaluator` to evaluate metadata and report to the content moderation system. The two classes form the critical path of the application processing pipeline, as shown in Figure 3.10b.
- **Object Encapsulation.** We use inheritance and polymorphism to enhance software reuse by encapsulating FaaS functions and states in classes derived from two base classes. This approach hides the need for storage services behind object abstraction and allows their implementation to be managed in the cloud. It simplifies development, as developers only build the processing pipeline once in the base classes. They can then focus on specific functionalities in the derived classes, avoiding **redundant** implementation when adding new content types or evaluator services.

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

3.6. Summary

In this research, we presented the OaaS paradigm, which aims to simplify state and workflow management for cloud-native applications. Our prototype, Oparaca, supports both structured and unstructured data types, ensuring consistency through fail-safe state transitions. It also provides secure and low-overhead management for unstructured data using presigned URLs and redirection mechanisms. For structured data, it employs the pure function scheme to transparently manage application data to the developer code and using DHT and consistent hashing to scalably cache the object data and improve the data locality. To make the Oparaca fault-tolerant, we developed the *exactly-once* and *localized locking* schemes. To support cloud-native workflow, Oparaca enables declarative dataflow abstraction that hides the concurrency and synchronization concerns from the developer's perspective. The evaluation results demonstrate that Oparaca streamlines cloud-native programming and is ideal for use cases that require persisting the state or defining a workflow. Oparaca offers scalability with negligible overhead, particularly for compute-intensive tasks. With the core OaaS abstraction and Oparaca architecture in place, Chapter 4 builds on these foundations with declarative non-functional requirement management, and Chapter 5 extends OaaS across the edge–cloud continuum.

CHAPTER 4

SLA-DRIVEN OAAS: DECLARATIVE PERFORMANCE CONTROL FOR CLOUD-NATIVE APPLICATIONS[1]

4.1. Overview

While Chapter 3 demonstrated how the Object as a Service paradigm unifies application logic and state management, a critical limitation remains: *developers still lack systematic control over non-functional requirements such as performance, availability, and consistency.* The empirical studies in Chapter 1 revealed that practitioners prioritize service quality assurance and unified maintainability, expecting platforms to provide high SLA offerings and rich configurability for performance optimization without requiring deep infrastructure expertise. Current serverless platforms force developers into iterative trial-and-error cycles—deploying applications, measuring performance, manually adjusting configurations, and redeploying—a process that contradicts both productivity goals and the serverless promise of automated resource management.

This chapter extends OaaS with a declarative Non-functional Requirement (NFR) interface that enables developers to specify desired outcomes—availability targets, throughput requirements, consistency guarantees, and latency constraints—through high-level, measurable specifications. The platform then automatically configures and adapts deployments to meet these requirements, eliminating manual tuning while establishing a symbiotic relationship between developers and cloud providers. We present the enhanced Oparaca system that realizes NFR-driven OaaS, demonstrating how declarative specifications can be systematically translated into runtime enforcement mechanisms. The chapter is organized as follows: Section 4.2 presents the design rationale for unifying abstraction and NFR control; Section 4.3 describes the Oparaca architecture for NFR enforcement, including SLA

[1]This chapter is based on: P. Lertpongrujikorn, H. D. Nguyen, and M. Amini Salehi, “Streamlining Cloud-Native Application Development and Deployment with Robust Encapsulation,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC ’24)*, Redmond, WA, USA, 2024. Reprinted and adapted here with permission from ACM.

specification interfaces, class runtime management, and automated optimization strategies; and Section 4.4 evaluates NFR enforcement demonstrating that Oparaca automatically satisfies declared throughput, latency, and availability requirements while reducing deployment complexity and eliminating manual tuning cycles.

4.2. Design Rationale: Unifying Abstraction and NFR Control in OaaS

To establish an agile and cost-efficient application delivery, the limitations of current FaaS platforms presented in Chapter 1 (specifically, fragmented state management and lack of NFR control) 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.

4.2.1. 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

Name	Value Type	Unit	Definition
NFR Specifications			
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 be 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 4.1. Potential Non-functional requirements and constraints. Those with bold font are currently supported by Oparaca.

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.

4.2.2. 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 NFR specifications (e.g., availability and throughput) 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 three-fold:

- *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 functionality, meet its NFR 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.

4.2.3. Simplified, Refinement-Free Deployment

Based on the ideas above, as shown in Figure 1.3b, 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 NFR specifications 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 4.3) 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.

4.3. Oparaca: Realizing the NFR-Driven OaaS

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

4.3.1. 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 NFRs 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`.

4.3.2. 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.

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),

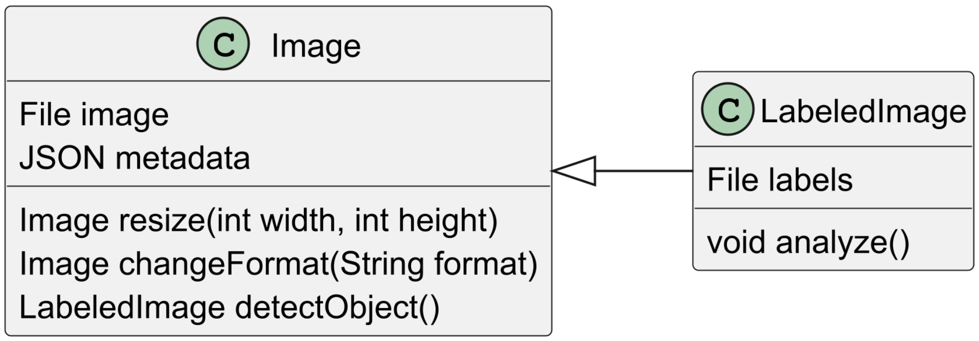


Image
File image
JSON metadata
Image resize(int width, int height)
Image changeFormat(String format)
LabeledImage detectObject()
LabeledImage
File labels
void analyze()

FIGURE 4.1. Class diagram for the image processing example. The developer can translate the class diagram directly to cloud deployment in Listing 4.1 through OaaS abstraction.

respectively. For structured state data, Oparaca allows the developer to keep the data as a JSON-based document, similar to the document database [26]. For unstructured data, however, object storage is employed to store them. We model each method as a serverless function[2]. Oparaca shares object states among methods of the same object following the OOP encapsulation principles.

In Oparaca, application NFRs 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 4.1 shows the list of NFR specifications 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 4.1 shows the class diagram of an example application providing image processing functionalities, such as resizing and changing the format. A developer can translate 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

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

Listing 4.1. 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
```

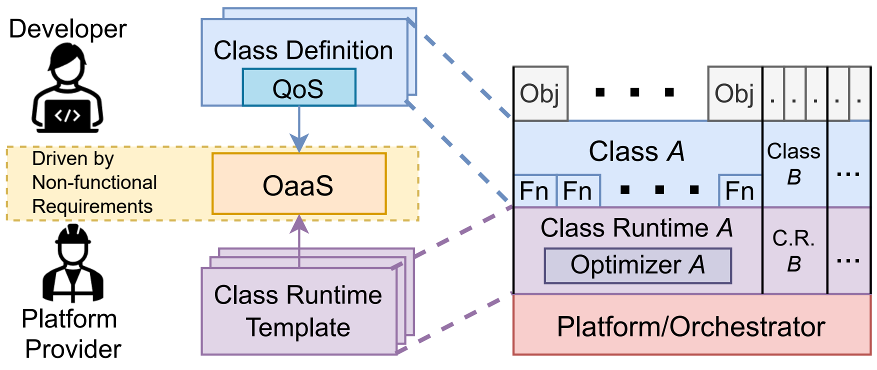


Figure 4.2. Realizing objects with class runtime and template: OaaS maintains templates customized for various deployment scenarios. For a specific class, Oparaca uses one of its predefined templates to create a class runtime to manage the deployed classes optimally.

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 4.1.

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`. Also, note that the `detectObject` function should inherit the class throughput requirements (150 rps); however, since its own NFR specification also includes throughput, Oparaca overrides its throughput (50 rps) instead.

4.3.3. Object Realization

Class Runtime and Template. Oparaca uses *class runtime* to deploy and manage objects derived from user-defined classes (Figure 4.2). 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.

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).

Class Runtime Example. Figure 4.3 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

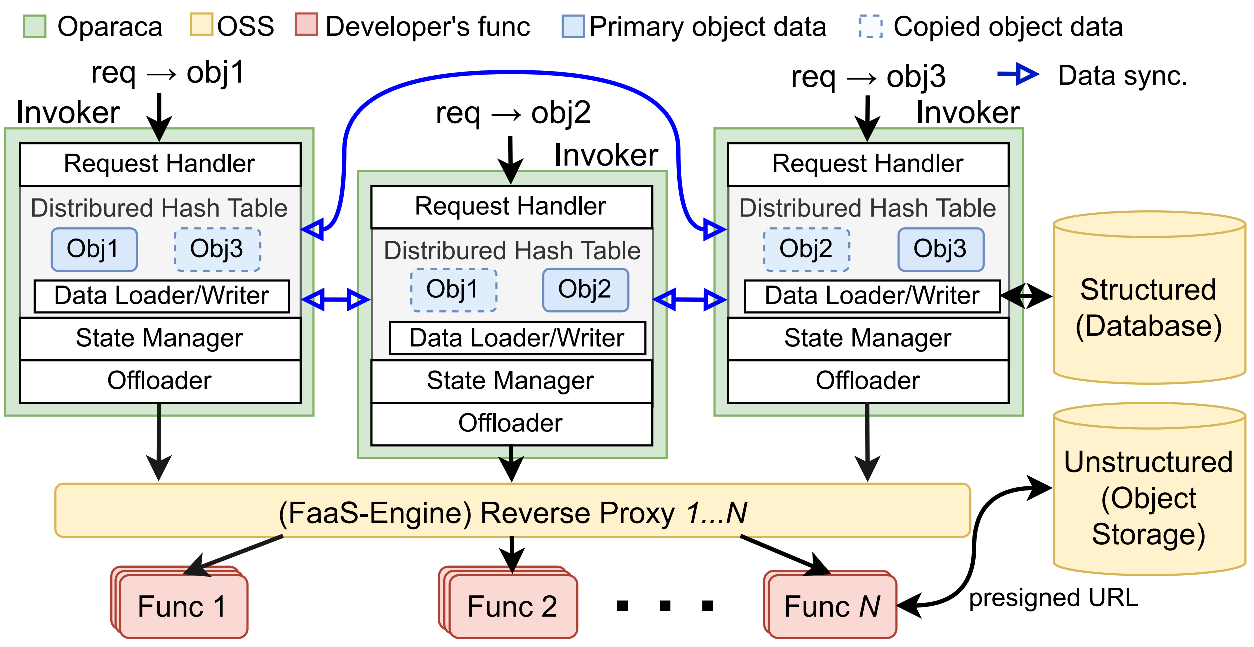


FIGURE 4.3. LTAG (Latency, Throughput, and Availability Guarantee): An example of a class runtime template designed for enforcing class latency, throughput, and availability requirements (OSS: Open-source software).

the updated attributes as the output to the invoker. The invoker maintains an internal in-memory distributed hash table (DHT) [51] 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 [89]. 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 [89, 87] 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 enforcement aims to minimize end-to-end latency in a best-effort manner [62, 68, 126, 124, 74], giving no guarantee to construct/realize non-functional requirements. Besides, other efforts try to keep latency within a specific target deadline [110, 12, 112, 84, 19], but this is extremely difficult from the cloud provider's perspective due to the highly dynamic and unpredictable nature of invocation logic [101, 63, 41], data size [19, 40, 86], and communication requirements [120]. 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 [77] as follows.

$$A = 1 - (1 - P)^N \tag{1}$$

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.

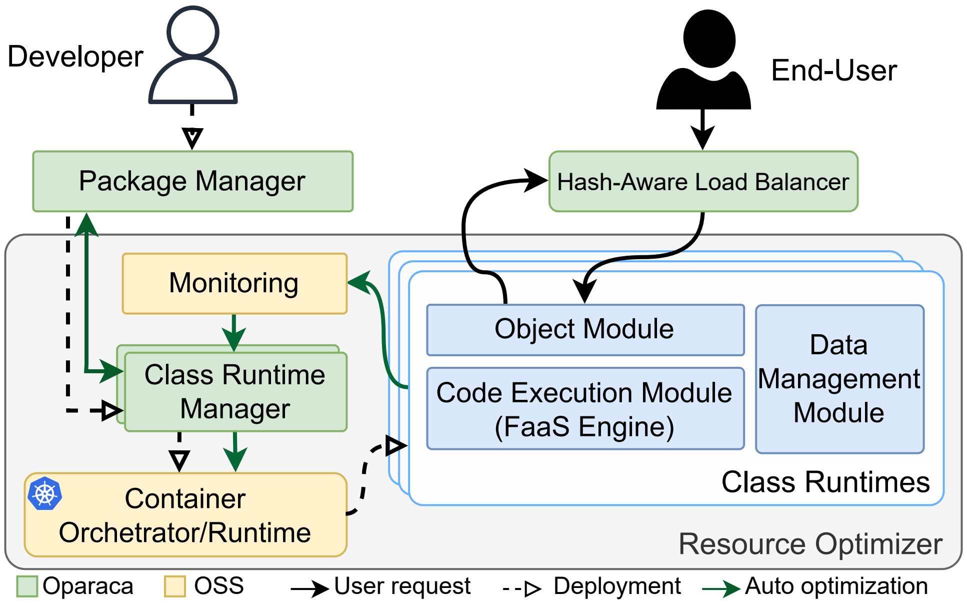


FIGURE 4.4. A bird-eye view of Oparaca's NFR-driven architecture

4.3.4. Oparaca Architecture

Oparaca's architecture, shown in Figure 4.4, 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., NFR specifications and constraints). The extracted information is then forwarded to the *class runtime manager* to find an appropriate 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.

4.4. Performance Evaluation

In this section, we seek to learn the performance of Oparaca in the following aspects: non-functional requirement enforcement (Section 4.4.2), implementation efficiency (Section 4.4.3), deployment productivity (Section 4.4.4), and development productivity (Section 4.4.5).

4.4.1. Experimental Setup

We prepare the experimental environment on 4 machines on Chameleon Cloud [64], 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 [109]) for deploying applications. The last machine generates load using Gatling [32]. Regarding data management, we use Minio [56] (S3-compatible storage) for unstructured data and ArangoDB [55] (document database) for structured data.

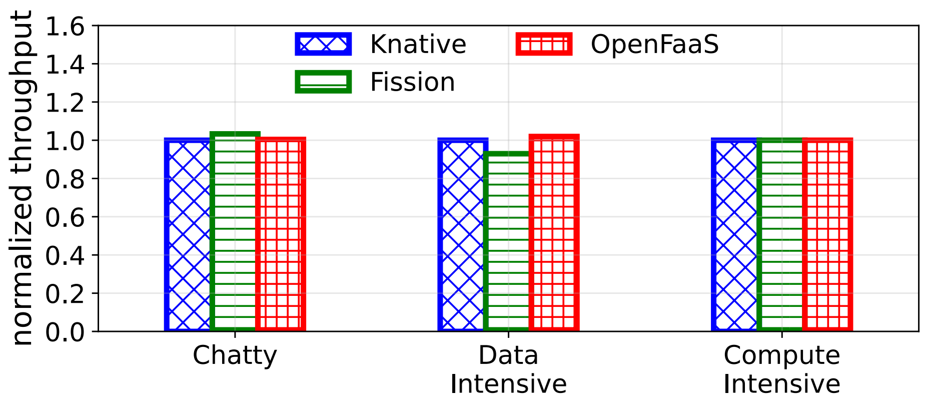
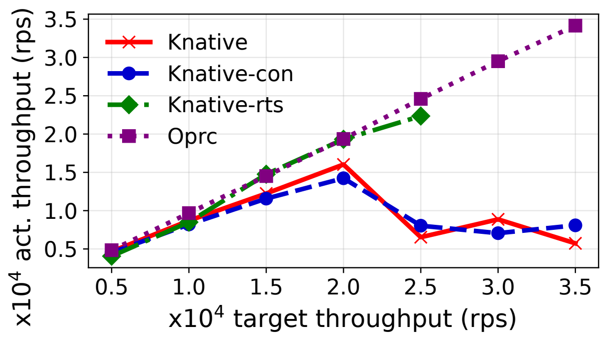


FIGURE 4.5. Oparaca does not significantly differ in throughput performance across the FaaS engines.

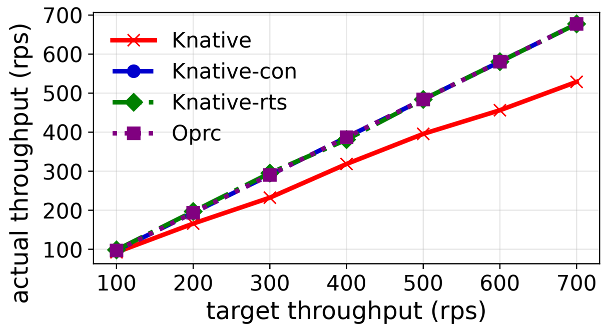
Workloads. To make sure our evaluation is comprehensive, we consider the following three classes of applications that exhibit different behaviors:

- *Chatty*: characterized by frequent small communications that impose significant overhead on network transmission [80]. As a representative workload for the application class, we utilize *JSON randomization* [75], 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 [53]. We use an *image resizing* workload [106, 15], 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 [27] and HPC [88] applications). To represent this class, we use *video transcoding* [85, 117], which involves changing the resolution of a video file stored in object storage.

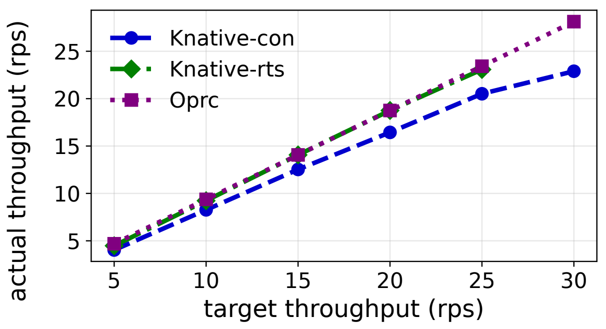
Approaches. To ensure generality, we integrated Oparaca with various FaaS engines—Knative [46], Fission [96], and OpenFaaS [42], all backed by Kubernetes—to host object functions. Figure 4.5 shows the maximum throughput achieved by workloads mentioned above when deployed over Oparaca using these different 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



(A) Chatty



(B) Data Intensive



(C) Compute Intensive

FIGURE 4.6. 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.

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 [89] to enforce throughput

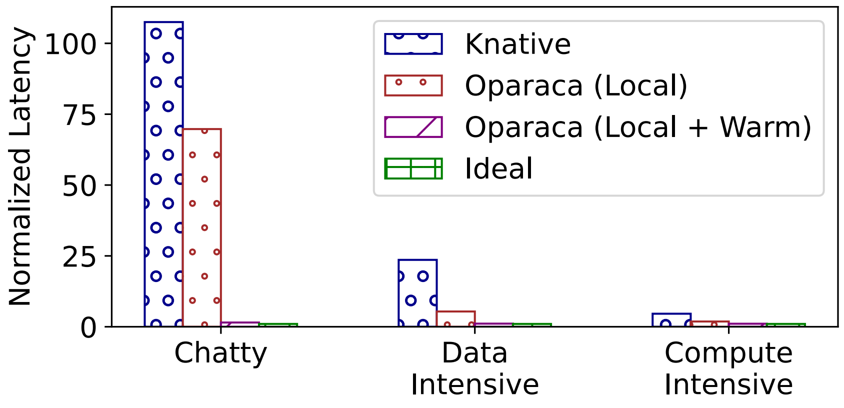


FIGURE 4.7. Oparaca can exploit data locality to provide various latency guarantees.

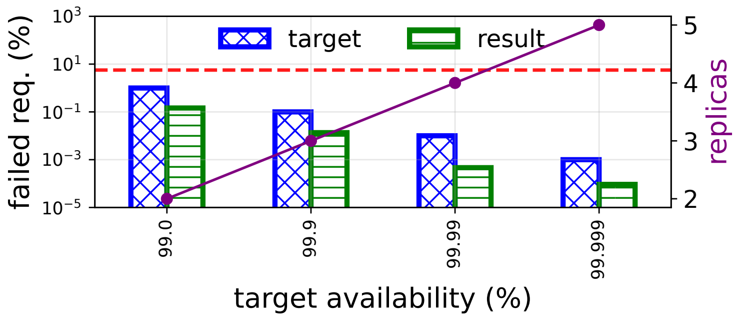
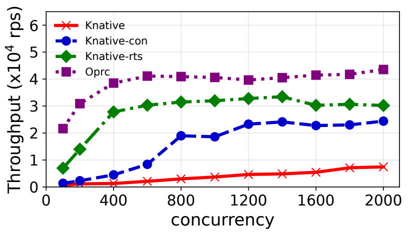


FIGURE 4.8. Successful invocation rate at different availability targets with availability enforcement. Resource stability (P) is 94.36% (red line).

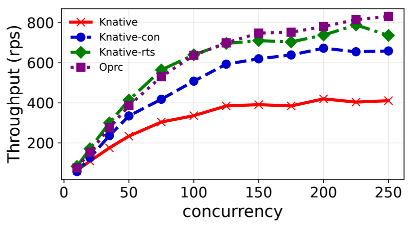
guarantee.

- *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.

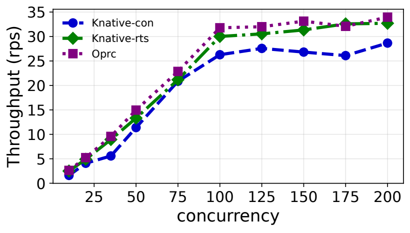
Beyond ensuring a fair comparison, we choose Knative as a FaaS baseline because it offers a rich set of configuration options to capture diverse deployment scenarios often unsupported by other engines. Additionally, varying Knative settings demonstrate how current FaaS implementations address non-functional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



(A) Chatty

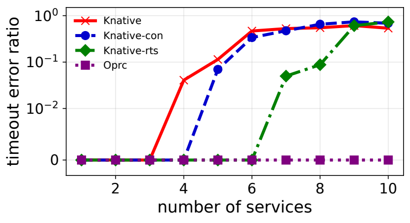


(B) Data Intensive

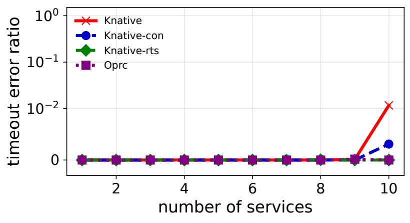


(C) Compute Intensive

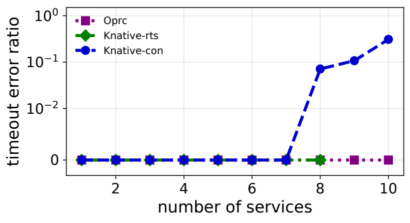
FIGURE 4.9. Achievable throughput under various request concurrency. Concurrency is defined as the number of clients that concurrently generate requests for the system.



(A) Chatty



(B) Data Intensive



(C) Compute Intensive

FIGURE 4.10. Error response ratio of different solutions upon deploying them with the different number of services.

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.

4.4.2. Non-functional Requirement Enforcement

We validate the NFR enforcement capability of Oparaca by deploying applications mentioned in Section 4.4.1 using the LTAG class runtime template as described in Section 4.3.3.

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 4.6.

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.

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 [7], 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 4.7 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 [58] 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 4.8. 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% [6] 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.*

4.4.3. 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 4.9.

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 4.9a) 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 4.9b), *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 4.9c), 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 4.10, overall, Oparaca outperforms other baselines for almost all workloads. For the chatty workload (Figure 4.10a), 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 4.10b), 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 4.10c), *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.*

4.4.4. 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

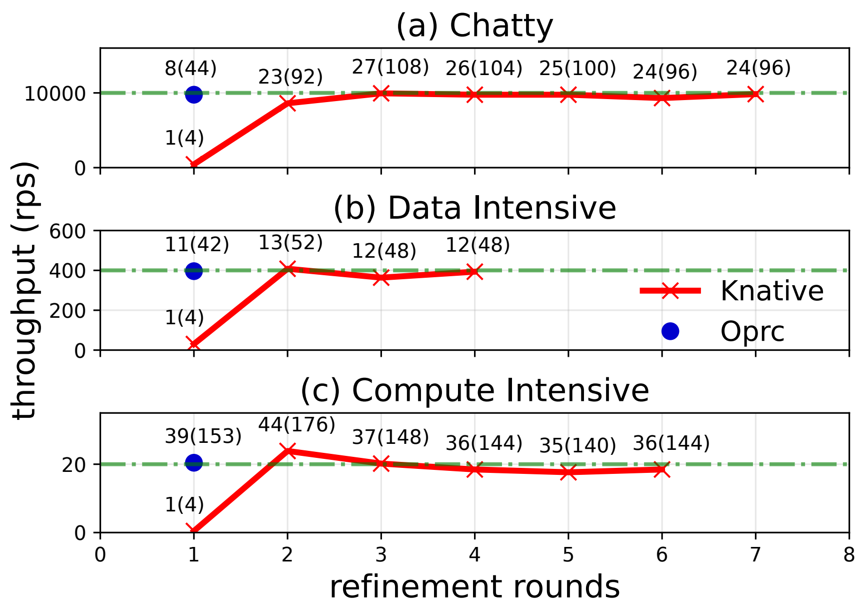


FIGURE 4.11. Rounds of refinement for Knative to enforce the target throughput (green lines) versus Oparaca. Data points are annotated by `#pods(#cores)`, including invoker pods.

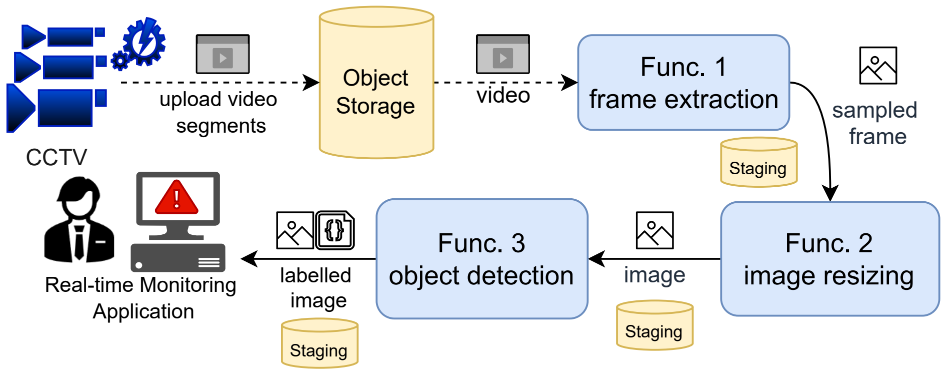


FIGURE 4.12. The case study of developing video and image processing for a real-time monitoring system

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 4.11, 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.*

4.4.5. 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.

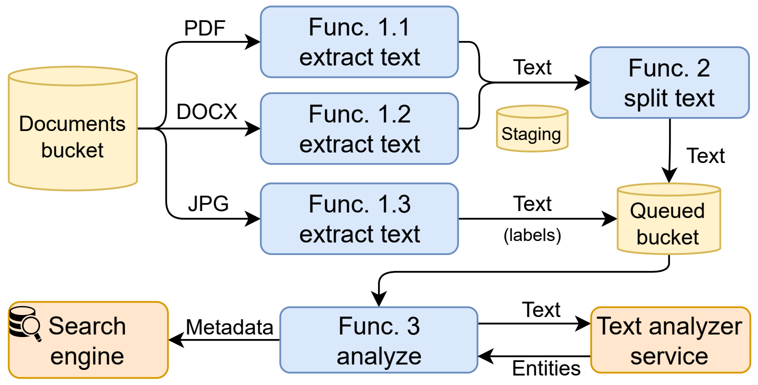
Case Study # 1. Real-time Monitoring System.. Figure 4.12 shows a CCTV system uploading video segments to object storage, waiting to be processed by a workflow of function that includes `extractFrame()` that splits a video segment into multiple frames; `resizeImg()` whose job is to resize the image frame 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.

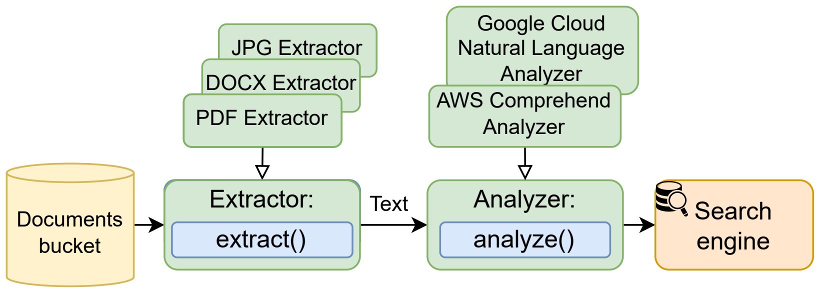
OaaS implementation. The developer defines three classes:

- `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 4.1).
- `LabeledImage` class inherits from the `Image` class and has its own `objectDetection()` function and `labels` data (state) in JSON format (see Listing 4.1).

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



(A) FaaS-based [8]



(B) OaaS-based

FIGURE 4.13. The searchable enterprise document repository implemented based on FaaS and OaaS paradigm.

authentication steps. To ensure the application performance, developers only need to declare the target throughput within the class definition (see example in Listing 4.1); 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 [88, 116]. 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 4.13 shows the serverless workflow to analyze the document in various formats and update the metadata to the search engine recommended by AWS [8]. 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 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 4.13b.
- **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.*

4.5. Summary

This chapter presented declarative Non-functional Requirement (NFR) management as a critical capability of the Object-as-a-Service paradigm. By encapsulating application logic, data, and performance specifications into unified class definitions, OaaS enables developers to express desired outcomes—throughput guarantees, latency targets, availability requirements—through intuitive declarations rather than low-level configurations. The Oparaca platform demonstrates this approach through the LTAG class runtime template, which automatically translates NFR specifications into optimized deployments. Evaluation across diverse workloads confirms that Oparaca enforces declared NFRs with comparable or superior resource efficiency versus state-of-the-art approaches while dramatically reducing deployment complexity and eliminating manual refinement cycles. Case studies further illustrate how OaaS abstractions simplify real-world development by hiding infrastructure complexity behind object-oriented interfaces, allowing developers to focus on business logic.

While this chapter focused on centralized cloud deployments, modern applications increasingly demand distributed execution across edge and cloud environments. IoT applications, mobile computing, and latency-sensitive services require deployment flexibility beyond single data centers. However, the edge introduces new challenges: resource heterogeneity, intermittent connectivity, and geographic distribution complicate NFR enforcement strategies designed for centralized infrastructure. Chapter 5 addresses these challenges by extending OaaS across the edge-cloud continuum, introducing EdgeWeaver—an architecture that adapts declarative NFR management to distributed, geo-dispersed deployments.

CHAPTER 5

OAAS-IOT: EXTENDING OBJECT AS A SERVICE TO THE EDGE–CLOUD CONTINUUM[1]

5.1. Overview

Chapters 3 and 4 established the OaaS paradigm and SLA-driven deployment within centralized cloud environments. However, the empirical findings in Chapter 1 revealed that 43% of practitioners face responsiveness challenges, particularly in distributed and edge computing scenarios where low-latency, localized decision-making is critical for IoT, healthcare, finance, and robotics applications. Modern applications increasingly demand deployment across the edge-cloud continuum—spanning resource-constrained edge devices, regional edge datacenters, and centralized cloud infrastructure—yet current serverless platforms lack abstractions that gracefully span these heterogeneous, intermittently connected tiers.

This chapter extends OaaS to the edge-cloud continuum through OaaS-IoT, a paradigm that maintains the unified object abstraction while introducing SLA-driven placement, connectivity-aware invocation, and graceful degradation mechanisms tailored for geo-distributed, resource-heterogeneous environments. We present EdgeWeaver, a platform that realizes OaaS-IoT by automatically deploying objects across edge and cloud tiers based on declarative SLA specifications, managing state consistency despite intermittent connectivity, and adapting execution strategies to network conditions. The chapter is organized as follows: Section 5.2 presents the OaaS-IoT abstraction and its extensions for the edge-cloud continuum; Section 5.3 describes the EdgeWeaver architecture, including comprehensive object abstraction, SLA-driven deployment across tiers, and connectivity-aware enforcement mechanisms; and Section 5.5 evaluates EdgeWeaver through case studies and human studies demonstrating 44.5% reduction in lines of code, 10× fewer configuration requirements, and

[1]This chapter is based on: P. Lertpongrujikorn, H. D. Nguyen, and M. Amini Salehi, "EdgeWeaver: Seamless Edge–Cloud Integration for Stateful Serverless Applications," in *Proceedings of the 40th IEEE International Parallel and Distributed Processing Symposium (IPDPS '26)*, 2026 (under review). The version included here is the author's dissertation version and may differ from a final published version, if accepted.

31% faster development completion time compared to traditional FaaS-based approaches while maintaining robust SLA enforcement across the continuum.

5.2. OaaS-IoT: Abstraction for the Continuum

To overcome the limitations of existing approaches in addressing the complexity and connectivity challenges of IoT deployment across the Edge–Cloud continuum, we present OaaS-IoT, an extension of the OaaS paradigm that brings unified object abstraction to heterogeneous edge–cloud environments. We realize OaaS-IoT through the EdgeWeaver platform, which provides a *unified abstraction* that *decouples* the application from the underlying infrastructure, reducing the effort required to develop, adapt, and maintain IoT services in heterogeneous environments.

5.2.1. Comprehensive Object Abstraction

Building on the OaaS abstraction introduced in Chapter 3, OaaS-IoT applies the same object-oriented model across the edge–cloud continuum. Applications are composed of *distributed objects* whose *attributes* capture IoT state and whose *methods* (serverless functions) encapsulate logic; objects communicate via method invocations. QoS constraints are declared as SLAs (Table 5.1) and enforced by the platform. Unlike the cloud-only setting, objects may span intermittently connected tiers and heterogeneous devices, so the abstraction emphasizes location awareness (locality), resilience to disconnection, and smooth integration with device endpoints.

Developers reuse familiar OOP constructs (abstract classes, inheritance, polymorphism) to build modular IoT services while staying decoupled from infrastructure details. The same class definitions can be deployed at the edge or in the cloud without per-environment configuration, preserving portability while EdgeWeaver (the OaaS-IoT implementation) manages placement and communication.

5.2.2. Declarative SLA-Driven Deployment

Extending the declarative NFR management of Chapter 4, OaaS-IoT abstracts deployment complexity through SLA specifications attached to classes, attributes, and meth-

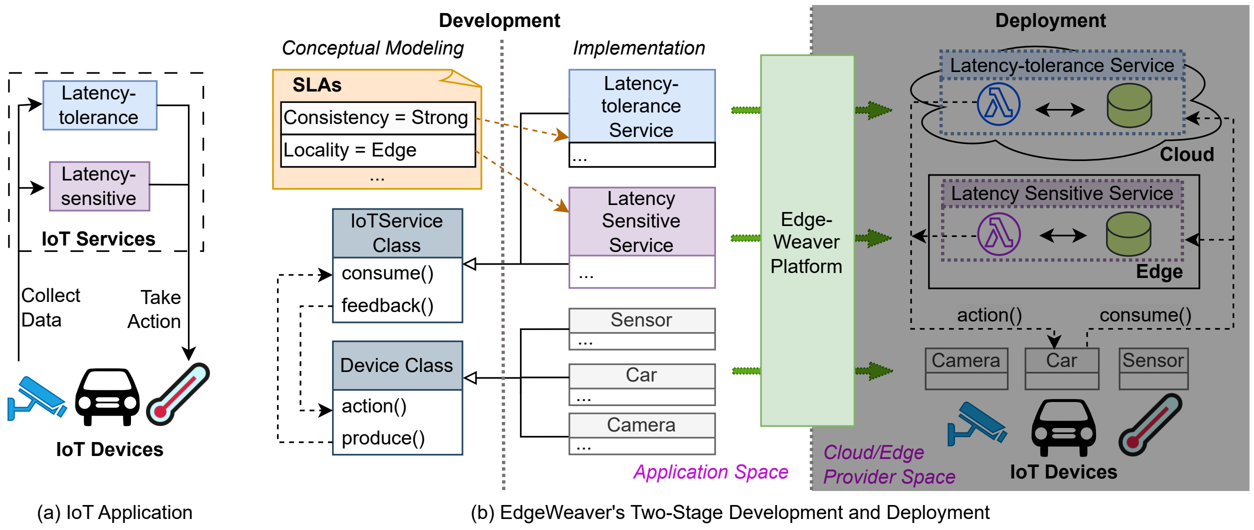


FIGURE 5.1. Overview of EdgeWeaver (EdgeWeaver implements the OaaS-IoT paradigm) to resolve Edge-Cloud challenges for IoT applications

ods. SLAs drive automated placement and configuration across edge and cloud, so applications meet their QoS targets without manual tuning. With a unified view of logic and state, EdgeWeaver co-locates functions with their data to minimize latency and transfer overhead, and adapts to changing conditions and heterogeneous resources to sustain SLA-compliant execution across the continuum.

5.2.3. Application Development and Deployment

As illustrated in Figure 5.1, OaaS-IoT establishes a novel two-stage abstraction for developing and deploying IoT applications across Edge-Cloud continuum:

Conceptual Modeling: Developers begin by modeling essential components and workflows using OaaS-IoT's unified abstraction. For example, heterogeneous IoT devices are modeled by a `Device` class, defining their basic operations (e.g., produce data and take action) with other IoT services, which are also modeled as separate classes. This high-level blueprint abstracts away the complexities of the underlying infrastructure, offering a clear and unified design framework.

Implementation: Developers extend these conceptual classes to capture the specifics of actual IoT devices and services, associating them with SLAs. The enriched class definitions are then submitted to the EdgeWeaver platform (the OaaS-IoT implementation). The platform extracts the embedded logic, state, and communication patterns to instantiate concrete Edge-Cloud components (e.g., FaaS functions, databases, and event pipelines) for deployment.

Deployment is fully automated within the provider ecosystem, with the declared SLAs driving each component placement, scheduling, and management. For example, as shown in Figure 5.1b, the latency-sensitive service is implemented with a "`locality=Edge`" SLA. During deployment, EdgeWeaver places its corresponding FaaS functions and state at the edge nodes to reduce data access and device communication latency. Conversely, a latency-tolerant service that requires strong consistency (e.g., linearization) is deployed on the cloud, where robust infrastructure can meet its SLA demands. With this SLA-driven deployment, EdgeWeaver ensures diverse application QoS requirements are met across Edge-Cloud continuum without any tuning effort from the developers, thereby, mitigating their deployment effort.

In sum, OaaS-IoT overcomes the limitations of existing approaches by providing a unified framework that simplifies IoT application development and deployment by abstracting both functional and non-functional aspects, ensuring seamless operation across intermittent Edge–Cloud environments.

5.3. EdgeWeaver: Realizing OaaS-IoT

5.3.1. Design Goals and Architecture

We design the EdgeWeaver platform, following the architecture shown in Figure 5.2, to realize the OaaS-IoT paradigm and meet three key requirements:

a. Comprehensiveness. To reduce development complexity and provide a unified view of IoT applications, we leverage object-oriented programming (OOP) concepts to define abstractions that capture both functional and non-functional aspects of application logic without exposing developers to underlying infrastructure details. We implement the *Object Abstraction*, which provides APIs and tools for modeling applications as classes, along with

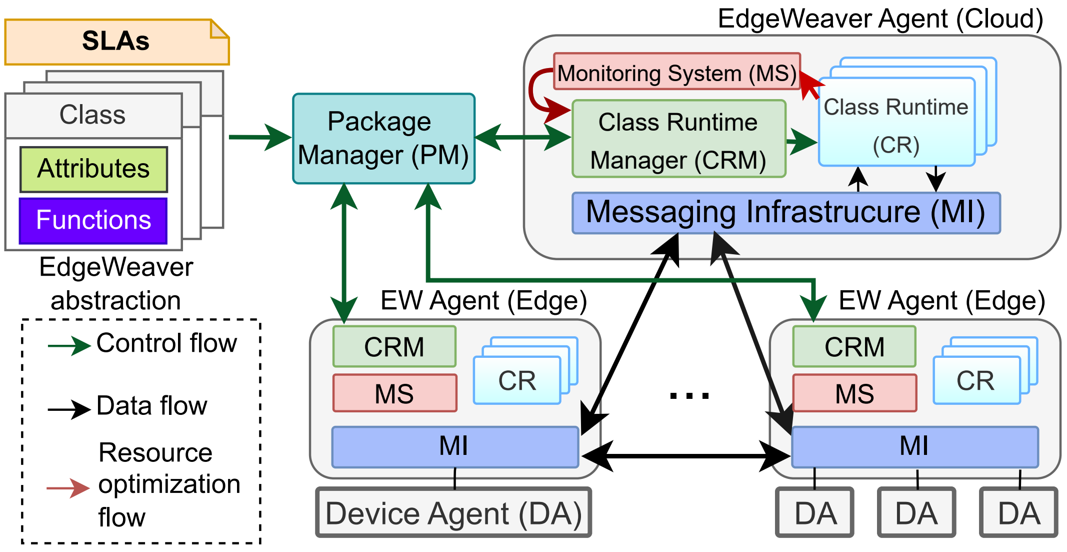


FIGURE 5.2. EdgeWeaver Architecture. The cloud-based Package Manager (PM) coordinates deployment across edge and cloud tiers, each hosting EdgeWeaver Agents with Class Runtimes (CR) for object execution, Class Runtime Managers (CRM) for lifecycle management, Monitoring Systems (MS) for SLA metrics, and Messaging Infrastructure (MI) for communication. Device Agents (DA) integrate IoT devices. Control flows (green), data flows (black), and resource optimization flows (red) enable SLA-compliant execution across the continuum.

SLAs to specify requirements for performance, availability, and consistency. Additionally, we introduce lightweight *Device Agents* (DA) that run on IoT devices, exposing platform-specific APIs to integrate these devices into the EdgeWeaver environment. Together, these components create a unified abstraction layer that supports full lifecycle of application design and development.

b. Adaptability. To handle the dynamics of execution environments (e.g, network failures), EdgeWeaver enables automated, SLA-driven adaptation. For that purpose, each tier across Edge–Cloud hosts an *EdgeWeaver agent*, which includes a *Monitoring System* (MS in Figure 5.2) that collects SLA-related metrics and a *Class Runtime Manager (CRM)* that adjusts deployments in real-time. This forms a localized control loop that continuously adapts to workload fluctuations, resource availability, and network conditions without manual tuning.

c. Applicability. To ensure broad adoption and scalable operation across heterogeneous infrastructures, EdgeWeaver uses a modular architecture. Objects are deployed via *Class Runtimes (CR)*, configured by the Class Runtime Manager (CRM) according to both SLA specifications and capabilities of the hosting datacenter. Each EdgeWeaver agent includes a *Messaging Infrastructure (MI)* that abstracts inter-object communication into a topic-based protocol-agnostic model. At the global level, *Package Manager* coordinates deployment and synchronization across tiers, enabling platform-wide consistency and scalable, hybrid Edge-Cloud deployments.

By fulfilling these design requirements, the EdgeWeaver architecture delivers its original vision: simplifying application development; enabling flexible and automated deployment; and supporting robust, SLA-compliant execution in heterogeneous and intermittently connected environments. In the remainder of this section, we describe how the architectural components interact to support end-to-end IoT application conceptualization, development, and deployment—demonstrating how EdgeWeaver meets its design goals in practice.

5.3.2. Object Abstraction Realization

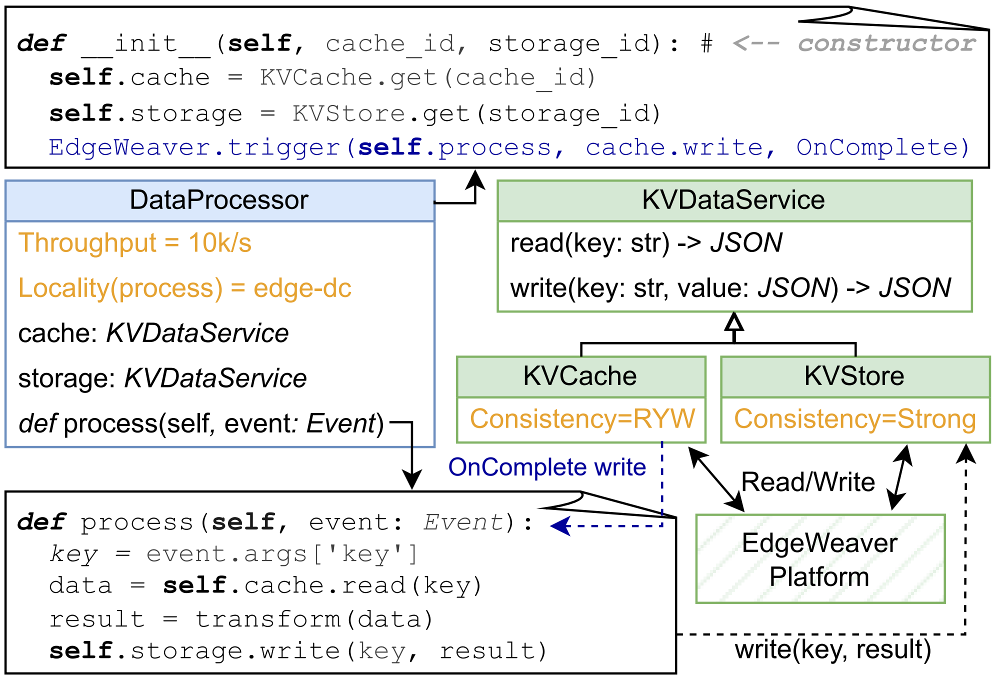
At its core, an OaaS-IoT application is structured around classes that define the blueprint for independently executable objects. Each class encapsulates attributes (representing state or data) and methods (implemented as serverless functions), following object-oriented programming principles. Upon deployment, EdgeWeaver automatically instantiates and manages these objects using distributed *Class Runtimes*, as described earlier. The abstraction also supports inheritance and polymorphism, allowing developers to create extensible, reusable components, promoting modular design and reducing code duplication. EdgeWeaver allows developers to associate SLAs with classes, methods, or attributes to specify QoS requirements, including locality, throughput, availability, and consistency (see Table 5.1). It also provides a high-level API (Table 5.2) for inter-object communication,

SLA	**Value Type**	**Unit**	**Definition**
Throughput	Integer	RPS	Minimum number of invocations guaranteed to be executed per second
Locality	Preferred datacenters	N/A	Preferred data centers that will be used for deployment
Availability	Real	%	The percentage of time an object/function must be available for service
Consistency	- Read your Write (RYW)	N/A	Ensure a client's next read includes its most recent write
	- Bounded Staleness (Δ)	sec	Read can lag behind the latest write, but only within Δ seconds
	- Strong	N/A	Read always reflects the latest write

TABLE 5.1. SLAs supported by OaaS-IoT

Categories	**API**	**Explanation**
Object APIs	`CLASS.create()`	Create a new object of class `CLASS` and return its ID
	`CLASS.get(ID)`	Retrieve an object of class `CLASS` by ID
	`CLASS.delete(ID)`	Delete an object of class `CLASS` by ID
Attribute APIs	`commit(obj, attr)`	Write local changes of `attr` in `obj` to storage
	`refresh(obj, attr)`	Read the latest value of `attr` in `obj` from storage
Function APIs	`trigger(func, src, e)`	Trigger `func` when event `e` occurs on `src`. Events: `OnComplete` or `OnFailure` if `src` is a function; `OnCreate`, `OnUpdate`, or `OnDelete` if `src` is an attribute
	`suppress(func, src, e)`	Disable trigger on `func` from `src` on event `e`

TABLE 5.2. EdgeWeaver's API


```
def __init__(self, cache_id, storage_id): # <-- constructor
  self.cache = KVCache.get(cache_id)
  self.storage = KVStore.get(storage_id)
  EdgeWeaver.trigger(self.process, cache.write, OnComplete)
```

```
def process(self, event: Event):
  key = event.args['key']
  data = self.cache.read(key)
  result = transform(data)
  self.storage.write(key, result)
```

FIGURE 5.3. Modeling and implementing a simple IoT processing service with EdgeWeaver

event-driven triggers, and system interactions—all without handling low-level details like networking protocols or deployment scripts.

Figure 5.3 illustrates how the abstraction supports comprehensive, infrastructure-independent application development. This example has an IoT data processing service, implemented by the `DataProcessor` class that consumes data from a short-term cache, processes it, and writes it to a long-term store. Since both expose a key-value interface, developers define a base class (`KVDataService`) and extend it into `KVCache` and `KVStore` through inheritance. EdgeWeaver manages all object instantiations and data handling internally. Developers simply attach SLA annotations, for example, `Consistency=Strong` on `KVStore` to ensure strict linearizability without manually configuring consensus protocols (e.g., Raft [91]). The `DataProcessor` interacts with these services via class attributes, which are automatically injected by the platform at object creation. Developers can also register event-driven triggers (e.g., executing a function upon adding data to the cache). Furthermore, performance requirements can be specified either globally (e.g., `throughput=10k/s`) or

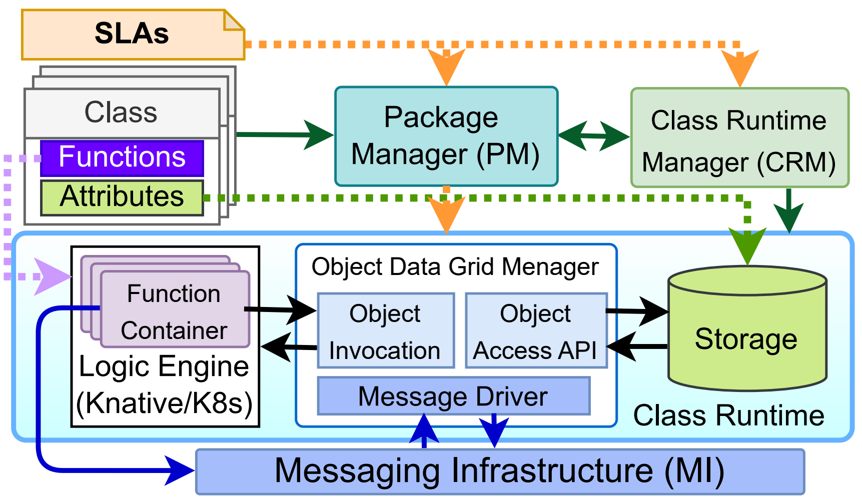


FIGURE 5.4. Class Runtime Internal Architecture. Each Class Runtime (CR) contains three core components: the Logic Engine executes class methods in isolated function containers (via Knative/K8s); the Storage System manages object attributes with appropriate backends based on data types and consistency requirements; and the Object Data Grid Manager (ODGM) orchestrates all operations through its Object Invocation module (routes method calls), Object Access API (enforces consistency protocols), and Message Driver (handles cross-datacenter communication).

per method (e.g., `Locality(process)=edge-dc`). All are managed entirely by EdgeWeaver; without any custom orchestration, messaging setup, or deployment scripting.

5.3.3. Class Deployment

Upon development completion, developers submit their applications to EdgeWeaver as a collection of class definitions annotated with SLAs. Figure 5.4 illustrates how EdgeWeaver transforms this submission into concrete deployments. First, the deployment package, including class definitions and SLAs, is submitted to the Package Manager. Based on the list of data centers the application is authorized to access, the Package Manager forwards the relevant class definitions to the corresponding EdgeWeaver agents operating in those target data centers. Each destination EdgeWeaver agent invokes its Class Runtime Manager to process the submission and instantiate a dedicated Class Runtime for each class. These class

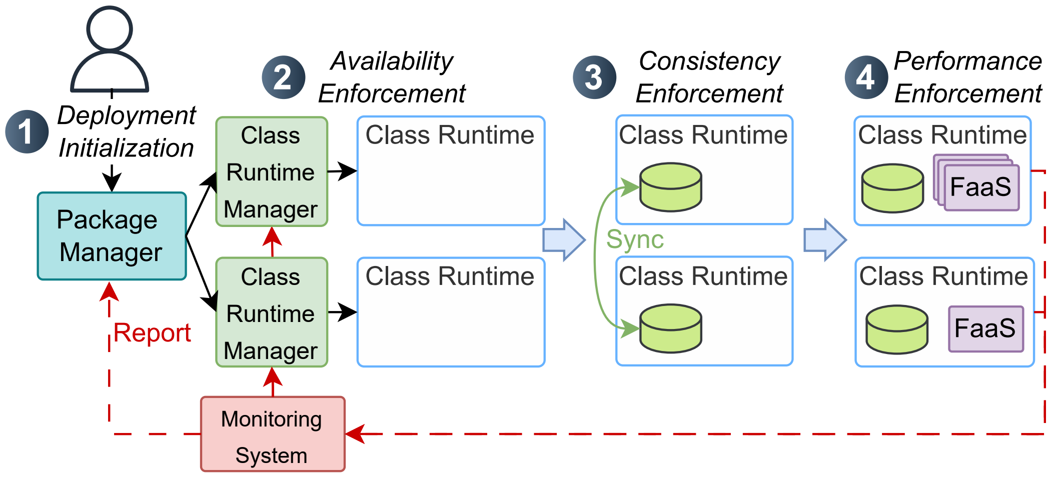


FIGURE 5.5. SLA Enforcement Workflow. The Package Manager initializes deployment (1) by distributing Class Runtime Managers to selected datacenters. Availability enforcement (2) determines replication factor and placement using failure probability metrics. Consistency enforcement (3) provisions storage with Raft (strong) or anti-entropy (bounded staleness) protocols. Performance enforcement (4) deploys pre-warmed containers to meet throughput requirements. The Monitoring System tracks metrics and triggers dynamic adaptation when violations occur.

Runtimes manage the lifecycle of all object instances associated with the class and enforce their SLA during their lifetime.

Each Class Runtime is composed of three key components: (i) *Logic Engine* that executes class methods; (ii) *Storage System* that instantiates appropriate storage backends for managing object attributes based on their data types and consistency requirements; and (iii) *Object Data Grid Manager (ODGM)* that orchestrates invocations and data access using modules for invocation routing, data consistency, and cross-datacenter communication (via Zenoh [73]).

5.4. SLA Enforcement

As in Chapter 4, SLAs define target semantics; here we adapt enforcement to cross-datacenter deployments and intermittently connected environments. Once a class is submitted and deployed, developers can instantiate objects whose lifecycles must comply with the SLAs in their class definitions. Figure 5.5 summarizes how EdgeWeaver enforces SLAs during deployment and execution, detailed next.

5.4.1. Availability Enforcement

At class deployment time, the Package Manager selects appropriate tiers (a.k.a. datacenters) to host Class Runtimes, guided by the availability SLA. It estimates the failure probability of each data center using metrics (e.g., uptime, network reliability) collected from the Monitoring System. Based on these estimates and the desired availability target, it calculates the required replication factor using 0the Meroufel and Belalem method [77]. The Package Manager then uses the replication factor to select the necessary number of data centers that satisfy developer-specified constraints (e.g., locality). If no constraints are given, it defaults to a round-robin strategy for load balancing. Class Runtimes are then deployed across these selected sites to host object replicas and ensure SLA-compliant availability.

5.4.2. Consistency Enforcement

When Class Runtimes are deployed, they provision storage and coordinate with each other to enforce the consistency SLA. For **Strong Consistency**, the Raft consensus protocol [91] is integrated into the ODGM's *Object Access API* to ensure that all replicas agree on the latest object states before processing reads or writes. **Bounded-Staleness Consistency** allows stale reads within a time-bound. Class Runtime's ODGM employs anti-entropy techniques with Merkle-Search Trees[14] to detect and repair inconsistencies within the defined window, along with CRDTs [104] to manage out-of-order changes. Read/write access is blocked if network partitions exceed the allowed staleness window. **Read-Your-Write (RYW) Consistency** allows reads to see a recent write from the same source, even under network partition. Class Runtime enforces this by routing reads and writes through the object access API to the same local storage.

5.4.3. Performance Enforcement

Each class's methods are deployed by the Logic Engine into isolated containers. If a throughput SLA is specified, containers are pre-warmed with sufficient compute resources—calculated using techniques from real-time Serverless [89]—to meet the required invocation rate. If a locality SLA is present, the preferred data center must reserve enough resources to meet both throughput and co-location requirements. The Class Runtime ensures that containers are deployed on the same machine as the object's storage to reduce access latency and pre-warms containers to avoid cold starts. When multiple replicas exist and locality is not a constraint, resource allocation is balanced across them, proportional to their resource availability.

5.4.4. SLA-compliance Execution

After deployment, users interact with objects via the API provided in Table 5.2. API calls are routed through the *Messaging Infrastructure*, which transparently directs each request to the appropriate ODGM instance based on the object ID embedded in the calls. Upon receiving a request, the ODGM's *Object Invocation* module triggers the corresponding function on the local Logic engine. If the function call targets a remote object, the ODGM leverages the *Message Driver* to relay the request to the corresponding location. This mechanism enables seamless, location-transparent invocation across the Edge–Cloud continuum. When a function needs to access object attributes, the attribute ID is passed through the *Messaging Infrastructure* to the *Object Access API* of the relevant ODGM. Before any data operation is executed, the ODGM enforces consistency guarantees by running the necessary replication and consistency protocols (see §5.4.2), to enforce SLA-compliant execution.

5.4.5. SLA Monitoring and Lifecycle Management

Monitoring System continuously collects SLA-related metrics (from Class Runtimes) that are reported to the Package Manager and the Class Runtime Manager. Upon SLA violation or runtime failure, EdgeWeaver automatically initiates corrective actions, such as reallocating resources or instantiating new runtimes to maintain SLA compliance for all objects during their lifecycle.

5.5. Performance Evaluation

5.5.1. Methodology

Goals. We evaluate EdgeWeaver across realistic settings to assess whether it fulfills its design objectives (§5.3.3) and thus, effectively addresses the challenges of IoT application development and deployment across the Edge–Cloud continuum. Specifically, we aim to answer the following key questions: (i) *Comprehensiveness and Productivity:* Does the unified object abstraction and declarative SLA interface provide a high-level view of IoT applications to support diverse QoS requirements and simplify development, ultimately improving developer productivity? (§5.5.3) (ii) *Efficiency for Practice Uses:* Can EdgeWeaver implement its abstractions and enforcement mechanisms efficiently and at scale, matching or even exceeding the performance of state-of-the-art systems, thus developers enjoy higher productivity without incurring significant trade-offs? (§5.5.4) (iii) *Adaptability for Reliable Execution:* Can EdgeWeaver dynamically respond to workload and infrastructure changes to preserve QoS with minimal developer effort? (§5.5.5)

5.5.2. Experimental Setup

We conduct experiments on Chameleon Cloud [64], using two clusters to represent the cloud and edge tiers. The *cloud cluster* consists of machines equipped with dual-socket Intel(R) Xeon(R) Platinum 8380 CPUs (240 cores total) and 768 GB of memory. The *edge cluster* uses machines with dual-socket Intel(R) Xeon(R) Gold 6240R CPUs (96 cores total) and 256 GB of memory. To reflect realistic deployment scenarios, we deploy the two clusters in geographically dispersed data centers: TACC (Texas) for the cloud and UC (Illinois) for the edge. The clusters communicate over a standard Internet connection with an average round-trip latency of 33 ms. Cloud cluster runs a full-fledged Kubernetes distribution using rke2 [109] while the edge cluster emulates resource-constrained edge environments with K3d [97], a lightweight Kubernetes distribution, in Docker containers. We configure the cloud Kubernetes with unlimited scaling, while the edge k3d consists of 8 K3d clusters, each of which has access to 8 vCPUs and 16 GB of memory, plus one machine acts as an IoT gateway, generating synthetic data and invocation requests targeting services deployed at both the edge and cloud levels. All machines in the setup communicate via Zenoh, a low-latency, publish/subscribe and query-based protocol that enables efficient coordination across the Edge–Cloud continuum. To simulate real-world network disruptions, we use Chaos Mesh [28] to inject intermittent connectivity faults. We install EdgeWeaver[2] alongside other baselines across both cloud and edge clusters to help deploy IoT applications. EdgeWeaver uses OpenRaft [92] to implement the Raft protocol while its ODGM maintains application state inside its embedded in-memory storage.

5.5.3. Comprehensiveness and Productivity

We show how EdgeWeaver improves the application development and deployment productivity via case studies and human evaluations.

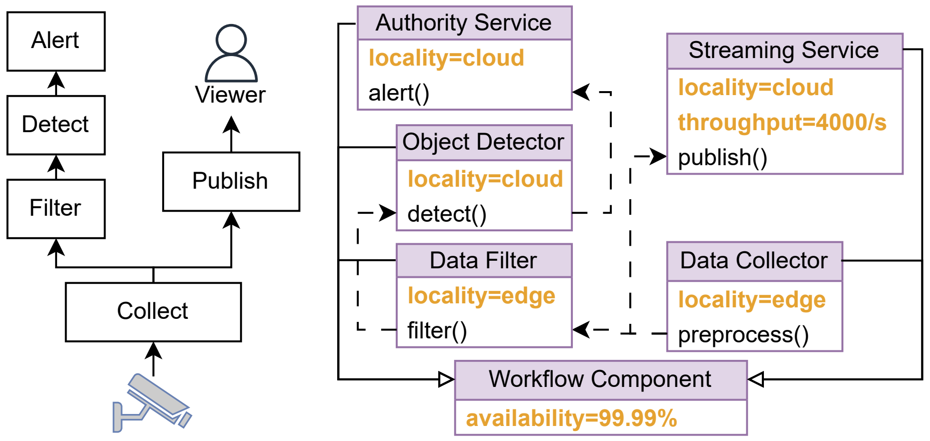


FIGURE 5.6. Real-time video analysis case study and the corresponding EdgeWeaver Implementation

Real-time Video Analysis (VA).. We begin with a representative use case in stream processing—a core building block of many modern IoT applications [100]. Figure 5.6 illustrates a real-time video analysis workflow [89] and its implementation using EdgeWeaver. The application *collects* live video from CCTV cameras, streams it to a traffic control center

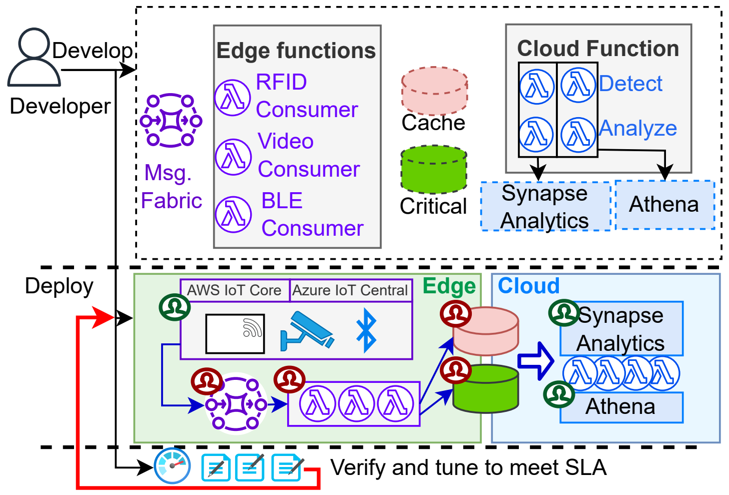
[2]The source code is available at `https://github.com/hpcclab/OaaS-IoT`.

for monitoring purposes, and simultaneously runs an object detection pipeline to detect suspicious activity (e.g., overspeeding). The pipeline includes a *filter* to extract relevant frames for an analyzer to *detect* suspicious actions and possibly issue *alert*.

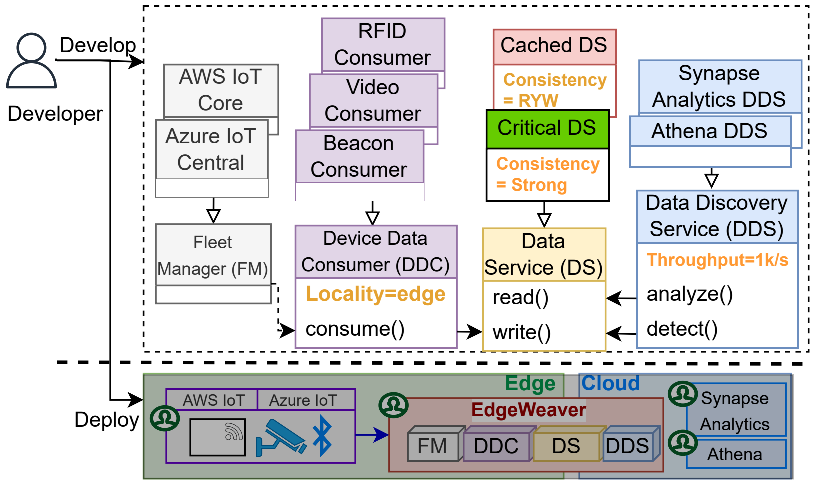
On the right-hand side of Figure 5.6 is how this workflow maps naturally into the object abstraction. Each processing component is implemented as an object, with functions encapsulating logic and attributes maintaining internal state. Component interactions are expressed through function chaining, easily configured using EdgeWeaver 's event-trigger API (Table 5.2). EdgeWeaver further simplifies QoS configuration through high-level constructs. Developers can define an abstract class (e.g., `Workflow Component`) with a shared availability SLA, and extend it for individual modules, enabling SLA inheritance without duplication. Components with intensive computation demands, such as the object detector, can specify `locality=cloud` to prefer cloud placement. For video streaming, stable throughput is critical and can be declared with an SLA: `throughput=4,000` invocations/sec with each invocation processes one second of video (i.e., 30 frames) for a camera. This enables smooth playback for up to 4000 concurrent connected devices without manual tuning.

Real-time Inventory Management (IM).. Figure 5.7 shows how developers could use traditional FaaS and EdgeWeaver to develop and deploy a Real-time Inventory Management (IM) system, a common workflow pattern of modern IoT applications. The FaaS-based architecture, recommended by Azure IoT [81] and AWS IoT [35], ingests heterogeneous data streams (e.g., RFID tags, beacons, video) from devices registered through Azure IoT Central. Each device type requires a dedicated FaaS function tailored to its protocol (e.g., `RFIDConsumer` over MQTT, `VideoConsumer` over TCP). Processed data are stored in a fast **cache** and later queried by analytics functions (e.g., Analyze) via services like **Azure Synapse Analytics** for analysis, and results are persisted in a **critical** database.

This FaaS-based approach is *complex* and *fragmented*. Developers must manage numerous protocol-specific functions, analytics modules, and data stores, while integrating multiple services (e.g., AWS IoT Core, Azure IoT Central) for cross-platform support. It also lacks native QoS enforcement, forcing a manual verify-and-tune cycle of adjusting func-



(A) FaaS-based solutions using AWS and Azure services [81]



(B) Using EdgeWeaver

FIGURE 5.7. Developing and deploying a real-time inventory management system with FaaS and EdgeWeaver. head and should icons depict system components the developer has to interact during the application life cycle (green: high-level interaction, red: direct configuration). EdgeWeaver needs fewer component interactions and doesn't require the **verify and tune** loop to meet desired SLAs.

tion placement, resource allocation, and network settings to meet SLAs.

In contrast, EdgeWeaver *unifies* and *automates* this entire process (Figure 5.7b). Developers work with high-level object abstractions instead of low-level components. Specialized handlers (e.g., `RFIDConsumer`) are derived from a reusable Device Data Consumers (DDC) class; cloud-specific integrations are encapsulated within a polymorphic Fleet Manager (FM) class; and data management is streamlined via unified Data Service (DS) and Data Discovery Service (DDS) interfaces. SLAs are attached declaratively (e.g., `Locality=edge` for latency-sensitive tasks, `Consistency=strong` for critical data), eliminating the need for manual tuning.

Guided by these declarative policies, the EdgeWeaver runtime automatically handles provisioning, placement, and networking, automatically deploying components like FM and DDC at the edge to meet locality requirements. This full-stack automation removes the manual, error-prone configuration cycle inherent in traditional FaaS systems, enabling faster, more reliable, and maintainable IoT deployments.

Productivity Improvement. To quantitatively assess EdgeWeaver's productivity gains over FaaS approach, we implemented two prototypes of the inventory management application: one using Knative (FaaS-based) and one with EdgeWeaver. Development effort was measured using three metrics: Lines of Code (LoC), Lines of Configuration Code (LoCC), and the number of developer-facing interfaces.

The results show a dramatic reduction in development overhead with EdgeWeaver. The Knative implementation required 666 LoC, while EdgeWeaver achieved equivalent functionality in only 363 LoC (44.5% reduction). The improvement in configuration effort was even more pronounced: EdgeWeaver needed just 39 LoCC, nearly 10$\times$ fewer than Knative (417 LoCC), which involves configuring multiple external services such as RabbitMQ, databases, and triggers. This highlights EdgeWeaver's strength in abstracting complex infrastructure management.

Furthermore, as shown in Figures 5.7 and 5.8, the FaaS-based design forces developers to manage at least seven distinct components, whereas EdgeWeaver consolidates these into

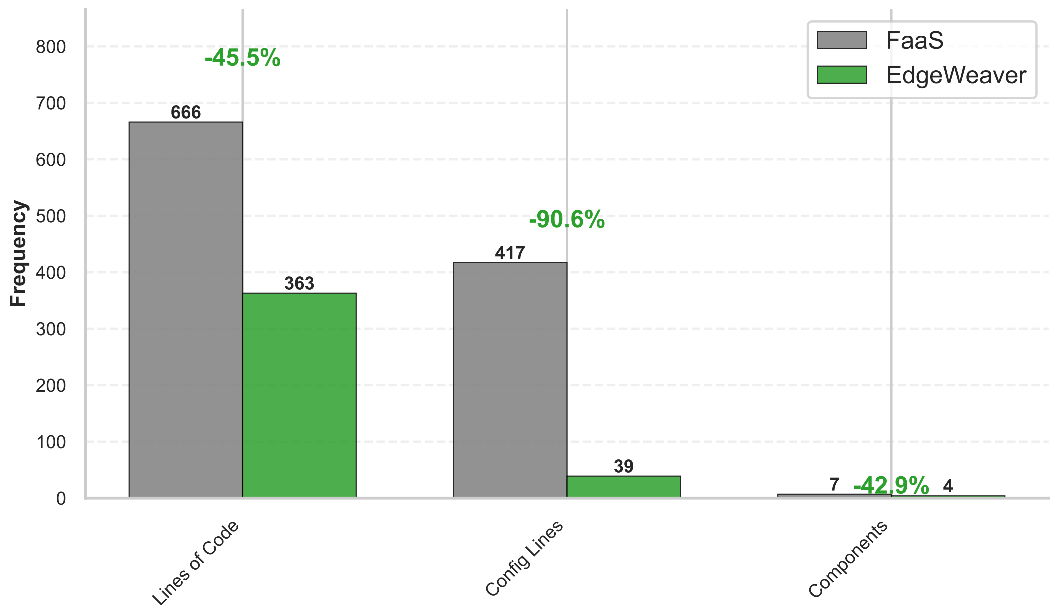


Figure 5.8. EdgeWeaver productivity improvements in the inventory management case study

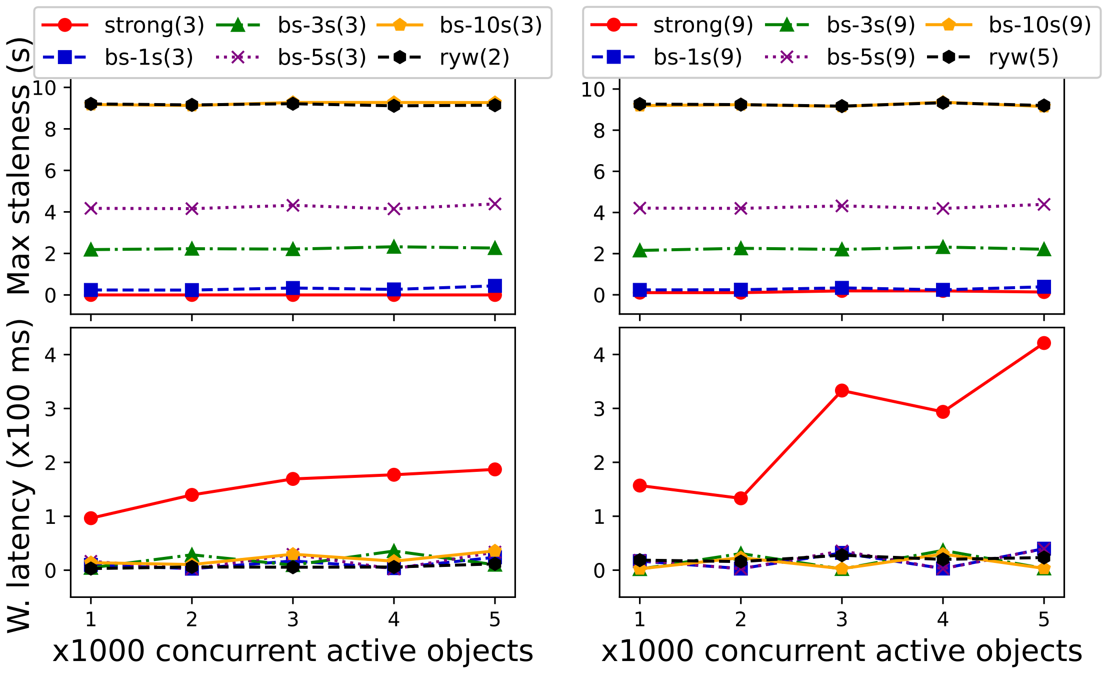
only four. Together, these results demonstrate that EdgeWeaver's unified, declarative interface and automated orchestration significantly reduce development complexity—making it far more productive, maintainable, and developer-friendly than traditional FaaS-based solutions.

Cloud fam.	**EW**	**FaaS**
Unfamiliar	84.6%	81.5%
Basic	90.0%	80.0%
Competent	92.0%	88.0%

Metrics	**EW**	**FaaS**
Time (min.)	22.43	32.40
Score (%)	52.85	53.55

Table 5.3. Human Study results (average): Quiz (left) and Programming (right). (EW=EdgeWeaver)

Developer Experience. To evaluate how EdgeWeaver's comprehensiveness and productivity translate into better developer experience, we conducted a human study with 39 college students. Participants received short (15 min.) tutorials on both the FaaS and EdgeWeaver abstractions, followed by a quiz to assess conceptual understanding and a pro-



(A) Availability = 4 nines (B) Availability = 9 nines

FIGURE 5.9. Maximum read staleness (top) and average write latency (bottom) under different consistency–availability configurations. Numbers in parentheses show the replica count required to meet both guarantees.

gramming assignment to measure practical performance. As shown in Table 5.3 (left), participants scored consistently higher on EdgeWeaver-related quiz questions than on FaaS ones, regardless of prior cloud experience. This indicates that EdgeWeaver is more intuitive and easier to learn. For the programming task (Table 5.3, right), out of the group who can complete the task in both platforms in time, students completed the assignment 31% faster using EdgeWeaver while achieving nearly identical code quality (EdgeWeaver: 53.6% vs FaaS: 52.9%). These results demonstrate that EdgeWeaver's unified abstractions and automation not only simplify development but also deliver a measurably better, faster, and more accessible.

Takeaway: *EdgeWeaver provides a unified, comprehensive abstraction that streamlines development and deployment of IoT applications across the Edge–Cloud continuum.*

5.5.4. Applicability and Efficiency

We evaluate applicability of EdgeWeaver by demonstrating its enforcement of diverse SLA combinations through case studies. We assess its efficiency in handling applications with different computational demands, showing its suitability for diverse IoT scenarios.

SLA Enforcement. We evaluate the ability to enforce various consistency levels: Read-Your-Write (*ryw*), *strong*, and bounded staleness (*bs*) under varying staleness bounds and high availability targets. According to Fig. 5.9, we deploy multiple concurrent `DataService` objects (from the Inventory case study), each issuing reads and writes. We set availability to 99.99%, comparable to leading FaaS SLAs (e.g., AWS Lambda's 99.95% [6]) and Tier-4 datacenters (99.995% [52]).

Across all configurations, EdgeWeaver consistently enforces the specified consistency guarantees: Under bounded staleness, observed staleness remains well below set thresholds. Even when stateless is relaxed in RYW, the stateless is consistently below 10 seconds. Notably, under strong consistency, zero staleness is detected, validating *reliable* consistency enforcement of EdgeWeaver. These guarantees hold at scale: With 1,000 concurrent objects, strong consistency maintains an average write latency of 100 ms, which only increases by 1.87× when scaling to 5,000 objects. RYW and bounded staleness achieve significantly lower latency (< 20 ms), over 9× faster than strong consistency, highlighting promising performance–consistency trade-offs that developers can leverage for various QoS needs.

To test robustness, we increase the SLA target to nine nines (i.e., 99.999999999%)—five orders of magnitude higher than the current standard. EdgeWeaver continues to satisfy all consistency requirements: bounded staleness remains under 10s, and strong consistency still achieves zero staleness. The write latency for strong consistency increases by at most 2×, while weaker models show a negligible impact. Finally, EdgeWeaver achieves these guarantees cost-effectively. At four nines, enforcing availability requires just 2–3 replicas. Even at nine nines, the system needs no more than nine replicas, a 3× cost increase for $10,000\times$ higher reliability.

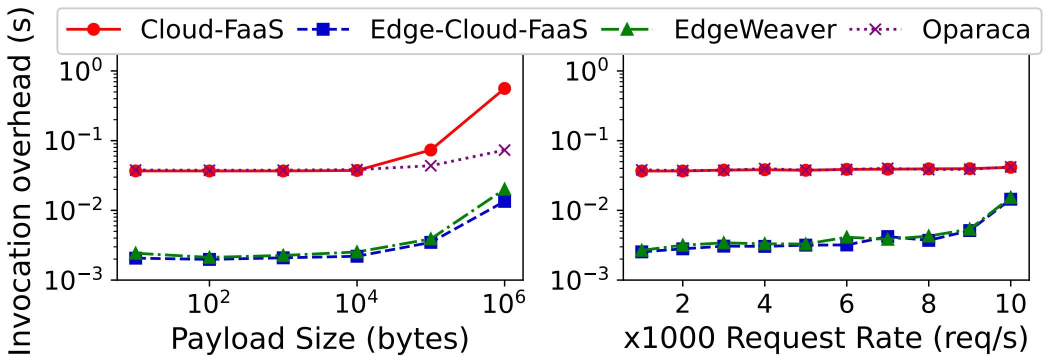


FIGURE 5.10. Latency of function invocation with increasing request rate and payload size in various Edge-Cloud deployment options.

Implementation Efficiency. We evaluate the implementation overhead introduced by EdgeWeaver's object abstraction and SLA enforcement by comparing it against equivalent FaaS-based implementations. Since EdgeWeaver builds on standard FaaS engines and employs Pub/Sub protocols for communication, we benchmark it using combinations of Knative [46] and Fission [96] with MQTT [95] and Zenoh [33].

We use a lightweight `echo` function that returns immediately upon invocation. This allows us to measure the combined overhead of network latency and platform runtime, independent of application logic. First, we evaluate the ability to minimize invocation overhead in the Edge-Cloud setting. Figure 5.10 shows the invocation overhead across four Edge-Cloud deployment options varying the request rate and payload. Note that this experiment uses a different set of baselines: Cloud-FaaS (Knative deployed in cloud only), Edge-Cloud-FaaS (Knative with manual edge placement), EdgeWeaver (EdgeWeaver with *Locality=edge*), and Oparaca (cloud-only OaaS deployment). For EdgeWeaver, we deploy the function with *Locality=edge*, forcing it to dispatch invocations at the closest edge to the IoT gateway. For Edge-Cloud-FaaS, we mimic this enforcement by manually rerouting invocation requests to the edge only.

Both EdgeWeaver and Edge-Cloud-FaaS outperform cloud-only baselines by at least 6.9×, confirming the benefits of edge deployment. In certain configurations, Edge-Cloud-FaaS can outperform EdgeWeaver by up to 6% due to its direct invocation path ver-

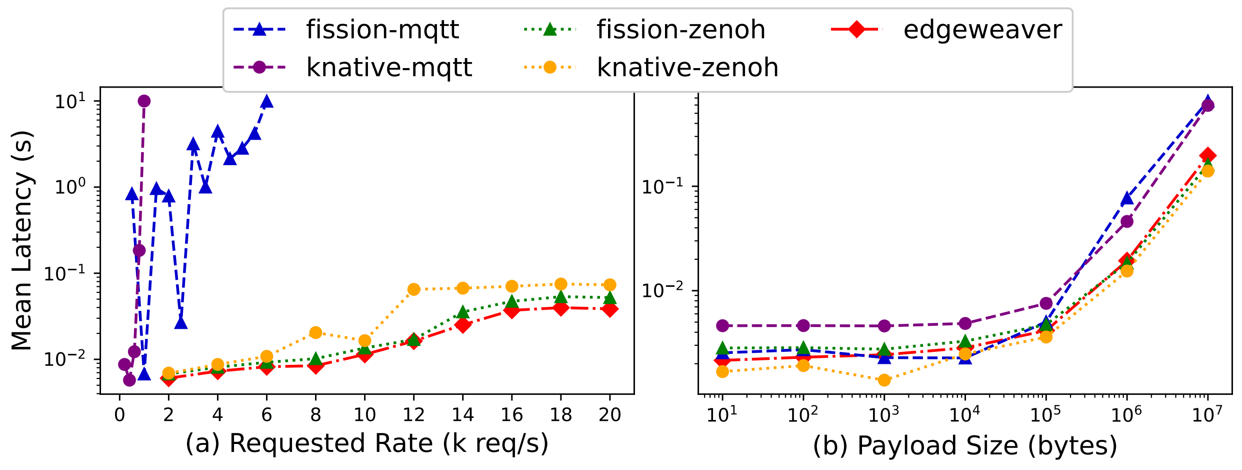
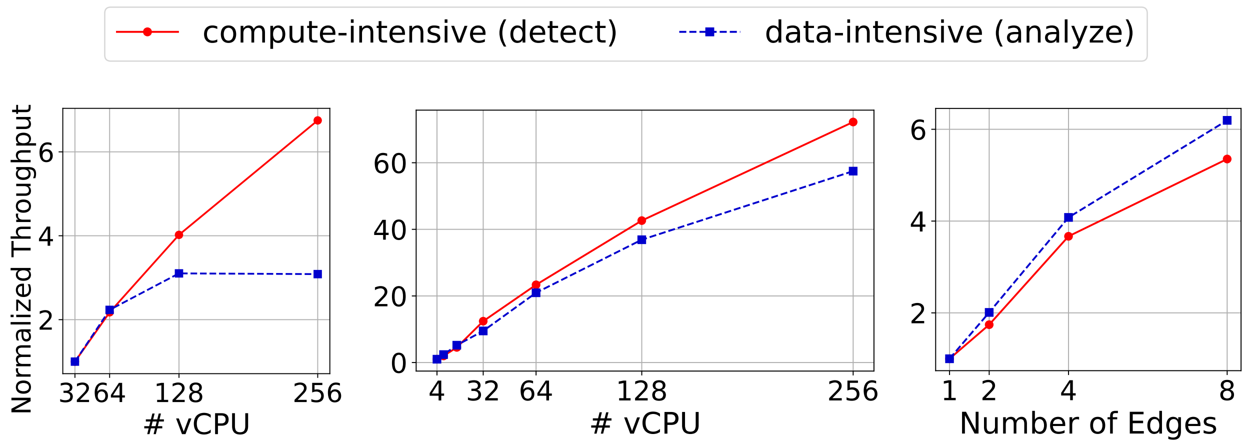


FIGURE 5.11. Latency of function invocation with increasing request rate and payload size in various baseline systems.

sus EdgeWeaver's SLA-driven routing. However, this minor difference is outweighed by EdgeWeaver automation. Unlike Edge-Cloud-FaaS, which requires manual tuning for optimal performance, EdgeWeaver automatically manages SLA-based placement to deliver comparable performance. This confirms that EdgeWeaver delivers a significant productivity boost without sacrificing performance or requiring additional developer effort.

More broadly, we compare the latency across different FaaS platforms and transport protocols. Figure 5.11 shows the results across various baseline systems. Although EdgeWeaver introduces additional mechanisms for object abstraction and SLA enforcement, this overhead is negligible. Across different request rates and payload sizes, EdgeWeaver consistently delivers performance on par with or better than the baselines, sometimes even outperforming them (e.g., Knative-MQTT). These results confirm that EdgeWeaver's adaptable runtime realizes its object abstraction and declarative SLA enforcement without compromising its performance, providing strong evidence of implementation efficiency in practice.

Scalability. We evaluate the scalability of EdgeWeaver by examining whether its implementation efficiency, observed in earlier experiments, holds under workload and resource scaling. To reflect realistic IoT usage, we consider two representative workloads: JSON document processing as a *data-intensive* task and image object detection using the YOLO



(A) Edge-Cloud

(B) Cloud

(C) Edge

FIGURE 5.12. Scalability analysis of EdgeWeaver across different deployment configurations. Throughput is normalized to the baseline allocation and measured for two workloads: data-intensive JSON processing (*analyze*) and compute-intensive object detection (*detect*).

model as a *compute-intensive* task. Both are implemented as the *analyze* and *detect* functions of the *Data Discovery Service* (DDS) in the inventory management case study (Figure 5.7). For each workload, we deploy multiple object instances across edge and cloud nodes, allowing EdgeWeaver to automatically determine their placement without explicit Locality constraints. Each instance is driven by a dedicated load generator that repeatedly invokes its corresponding function.

Figure 5.12a shows the throughput as we scale the total number of vCPUs across edge and cloud–from 8 vCPUs at the edge and 24 in the cloud, doubling the capacity incrementally. Throughput is normalized to the lowest allocation. Both workloads show strong scalability: throughput increases nearly linearly up to 128 vCPUs for *analyze* and 256 for *detect*. Beyond those points, performance plateaus due to network saturation between the TACC and UC data centers. To isolate the network impact, we rerun the experiments independently within each site. Figure 5.12b presents the cloud-only results. The *detect* workload, being compute-intensive, scales nearly linearly—achieving a 70× throughput gain from 4 to 256 vCPUs,

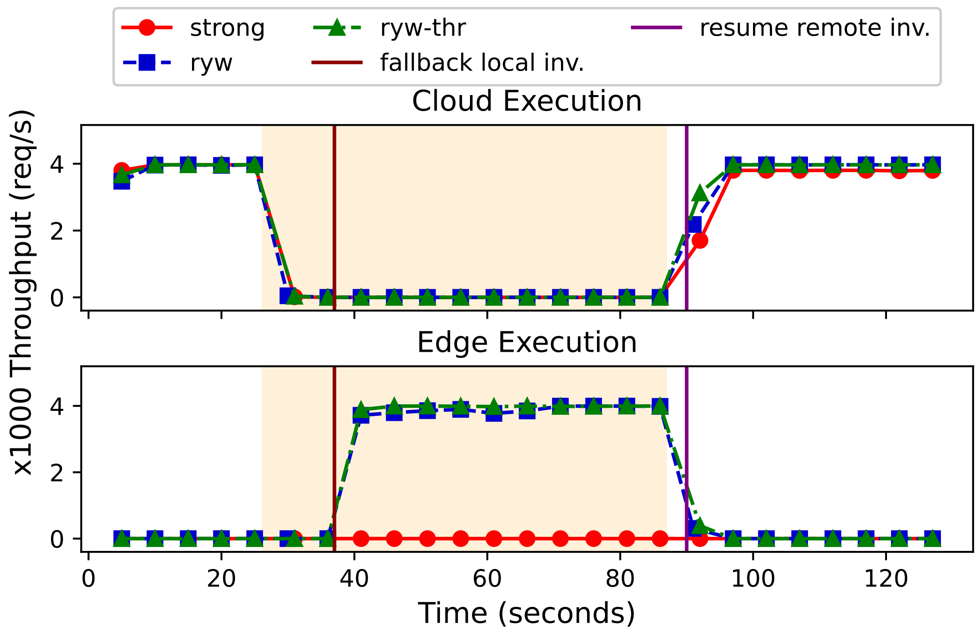


Figure 5.13. Impact of network partitioning for classes with: RYW, RYW with throughput, and Strong Consistency.

peaking at 144 invocations/sec. *analyze*, which is more data- and I/O-intensive, scales more moderately, reaching approximately 200,000 invocations/sec at 256 vCPUs. We observe similar patterns for edge-only (Fig. 5.12c): throughput scales proportionally with the number of edges (8 vCPU per edge), confirming EdgeWeaver sustains high throughput and scalability across diverse workloads and deployments.

Takeaway: *EdgeWeaver demonstrates strong applicability by providing efficient implementation to reliably enforce diverse SLAs and scale efficiently across Edge–Cloud.*

5.5.5. Adaptability

We evaluate the adaptability of EdgeWeaver by testing its ability to maintain QoS under dynamic network conditions. We deploy three functions with different SLA configurations: (i) `consume` (from the Device Data Consumer) with *Read-Your-Write* consistency (`ryw`), (ii) `write` (from the Data Service) with *RYW + throughput* guarantee (`ryw-thr`, 4,000 RPS), and (iii) `detect` (from the Data Discovery Service) with *strong* consistency. We assign high-availability SLAs, prompting EdgeWeaver to place replicas across edge-cloud.

We emulate *network partitioning* period between cloud and edge using Chaos Mesh [28] (yellow area in Fig. 5.13), where injected faults disrupt connectivity and trigger EdgeWeaver's runtime adaptation.

Initially, all functions continuously issue 4,000 RPS write requests to their associated data services. During the partition, EdgeWeaver detects the disruption and redirects invocations to the edge whenever possible. For the *ryw*, it permits continued execution by relaxing consistency, but since the underlying Logic engine is not inherently prepared for this scenario, its throughput is unstable. In contrast, the *ryw-thr* function benefits from the throughput SLA, maintaining stable performance. For strong consistency, EdgeWeaver enforces quorum strictly; as consensus cannot be achieved across the partition, throughput drops to zero, preserving correctness. Upon network restoration, *ryw-thr* quickly recovers full throughput, while *ryw* experiences a brief delay due to reactive scaling. The results show EdgeWeaver fine-grained adaptability, enabling dynamic balancing of QoS desires in response to changes.

Takeaway: *SLA-driven deployment enables EdgeWeaver to adapt automatically to dynamic environments to consistently meet application needs.*

5.6. Summary

In this chapter, we presented OaaS-IoT, an extension of the Object as a Service paradigm to the Edge-Cloud continuum, and its realization through the EdgeWeaver platform. Inspired by OOP principles, OaaS-IoT offers object abstraction that encapsulates application state, functions (logics), and SLAs, thereby providing a holistic view across the continuum. EdgeWeaver implements this paradigm by transparently handling user-defined consistency and availability trade-offs in the presence of network failure. Importantly, the benefits of OaaS-IoT as realized by EdgeWeaver do not come with any significant overhead to the system.

With the OaaS paradigm implemented for various contexts and use cases, in the next chapter, we explore the challenges and gaps to productize OaaS as a new cloud-native application development paradigm.

CHAPTER 6

EXPLORATION AND ANALYSIS OF THE OAAS PRODUCTIZATION

6.1. Overview

The proliferation of cloud, edge, and IoT computing has created a significant commercial opportunity born from a critical market pain point: profound infrastructural complexity. While Chapters 3–5 presented the technical foundations of Object-as-a-Service (OaaS), understanding the real-world market need and commercial viability requires systematic customer discovery. This complexity imposes a substantial tax on developer productivity, inflates operational costs, and acts as a barrier to innovation—representing a multi-billion-dollar market gap that OaaS is positioned to address.

This chapter presents commercialization validation from the NSF I-Corps National program (Summer 2025 Cohort 3), where we conducted 101 interviews across 86 organizations. Our customer discovery quantitatively confirmed a significant gap between modern infrastructure capabilities and teams' practical ability to utilize them effectively, validating the commercial opportunity OaaS addresses.

Section 6.2 describes our customer discovery approach and participant demographics. Section 6.3 presents quantitative validation of market pain points and expectations, with deployment complexity (38.6%), onboarding difficulty (35.6%), and system complexity (24.8%) as the top challenges. Section 6.4 examines how OaaS technical capabilities directly address these validated pain points. Section 6.5 identifies production readiness requirements and validates three target market segments, with technology SMEs and startups as the primary early-stage focus given their lower compliance barriers, while enterprise accounts require addressing production gaps first. Section 6.6 presents the Business Model Canvas synthesizing customer insights into a staged commercialization strategy prioritizing SMEs and research institutions before enterprise expansion. Section 6.7 summarizes the chapter's contributions.

6.2. Methodology

6.2.1. Customer Discovery Approach

We adopted the evidence-based customer discovery framework prescribed by the NSF I-Corps program [20], which emphasizes getting out of the building to test hypotheses about customer problems, needs, and willingness to pay. Our approach consisted of semi-structured interviews designed to explore current workflows, pain points, attempted solutions, and desired outcomes rather than pitching our technology.

6.2.2. Interview Protocol

Each interview followed a consistent protocol:

- **Duration**: 30+ minutes per interview
- **Format**: Semi-structured conversations via video conference or phone or in-person
- **Focus Areas**: Conversations explored current cloud/edge-cloud development workflows, deployment practices, primary challenges (technical and organizational), performance and cost concerns, prior attempted solutions, expectations for improvement, and potential commercialization opportunities including preferred deployment models and pricing sensitivities.
- **Data capture**: For each interview, we systematically recorded participant role, company industry and size, and coded thematic data on specific pain points and desired outcomes. Any quantitative metrics mentioned by participants regarding time savings or performance improvements were also explicitly captured.

6.2.3. Participant Demographics

Over the course of the I-Corps program, we conducted 101 interviews that 86 unique organizations (some organizations had multiple interviewees).

Role Distribution: As shown in Figure 6.1, participants spanned diverse technical and leadership roles. Table 6.1 summarizes the distribution of interview participants by role category.

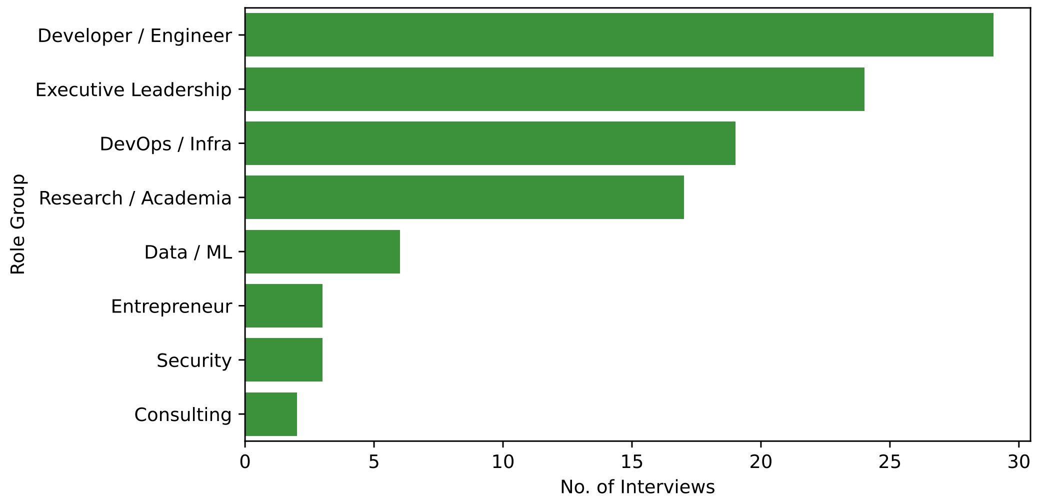


FIGURE 6.1. Distribution of interview participants by role group (N=101 interviews). Developer/Engineer and DevOps/Infra roles comprised nearly half of all interviews, reflecting our focus on technical practitioners directly involved in cloud-native development and deployment.

Industry Coverage: Figure 6.2 shows the breadth of industries represented. Table 6.2 summarizes the distribution of interviews across industry sectors.

Company Size Distribution: Figure 6.3 shows the sample spanned organizations of all sizes, from startups with fewer than 10 employees to large enterprises with over 10,000 employees. This distribution ensured that our findings captured challenges across different organizational contexts, from resource-constrained startups to large enterprises with established platform teams.

6.2.4. Data Analysis

Following each interview, we conducted systematic thematic coding to identify recurring pain points and expectations for improvement. We categorized pain points into eleven primary themes: deployment complexity, onboarding difficulty, system complexity, serverless-specific concerns, security concerns, integration challenges, observability limitations, cost management, scaling issues, documentation gaps, and responsiveness issues. We separately categorized expectations for improvement into thirteen dimensions: productivity,

Role Category	**#Interviews**	**Specific Roles**
Developer / Engineer	29	Software Engineers, Full-stack Developers, Product Engineers, QA Engineers
Executive Leadership	24	Managers, Founders, CEOs, CTOs, VPs of Engineering, Product Managers
DevOps / Infrastructure	19	DevOps Engineers, Infrastructure Engineers, SREs, IT Administrators
Research / Academia	17	Professors, Researchers, Research Scientists, PhD Students
Data / ML	6	Data Scientists, ML Engineers, Research Scientists
Security	3	Security Engineers, DevSecOps Engineers
Entrepreneur	3	Independent entrepreneurs and startup founders
Consulting	2	Technical consultants

TABLE 6.1. Distribution of interview participants by role category (N=101 interviews)

automation, onboarding, maintainability, availability, security, performance, cost optimization, programmability, integration, compliance, observability, and scalability. For interviews where participants made quantitative claims about potential productivity improvements or time savings, we extracted and recorded these metrics for aggregate analysis. This dual coding approach—capturing both current pain points and desired future states—enabled us to validate problem-solution fit between market needs and OaaS capabilities.

6.3. Market Validation: Pain Points and Customer Expectations

Our customer discovery revealed a consistent set of critical, data-supported challenges that quantitatively confirm the market opportunity OaaS addresses. The analysis demonstrates a significant gap between infrastructure capabilities and teams' practical ability to

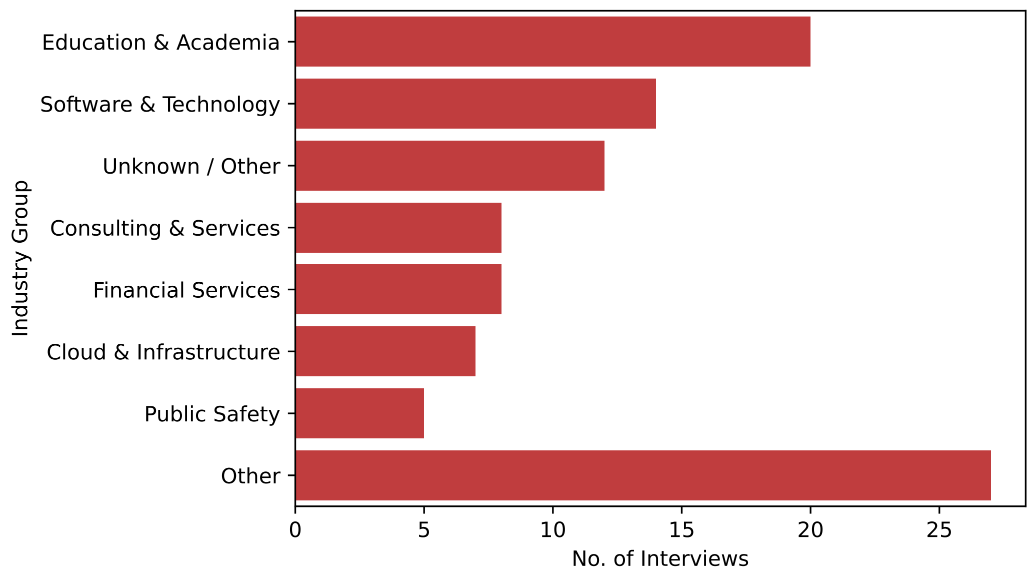


FIGURE 6.2. Distribution of interviews by industry group (N=101 interviews across 86 companies). Education, Software & Technology, and Financial Services represented the largest segments, providing diverse perspectives on cloud-native development challenges.

utilize them effectively—a challenge creating substantial operational drag on development organizations.

6.3.1. Pervasive Complexity and Operational Overhead

Figure 6.4 summarizes the frequency of pain points across all 101 interviews. The top challenges were:

(1) **Deployment complexity (38.6%)**: The most prevalent pain point involved fragmented tooling, configuration sprawl, and the burden of orchestrating multiple services (compute, storage, networking, monitoring) to deploy applications. Practitioners described spending significant time on "glue code" and infrastructure-as-code templates, with deployments often requiring coordination across multiple teams and tools.

(2) **Onboarding difficulty (35.6%)**: Steep learning curves for new team members and long time-to-productivity were frequently cited. The proliferation of cloud

Industry Sector	**#Interviews**	**#Companies**	**Description**
Education & Academia	20	13	Universities and research institutions
Software & Technology	14	14	Software development firms and technology companies
Financial Services	8	7	Banking, insurance, and fintech
Consulting & Services	8	7	IT consulting and managed services
Cloud & Infrastructure	7	4	Cloud providers and infrastructure companies
Public Safety	5	4	Public safety technology and emergency services
Healthcare & Life Sciences	4	4	Digital health and pharmaceutical companies
Research & HPC	4	4	High-performance computing and research facilities
Other Industries	12	12	Logistics & Transportation, Marketing & Advertising, Cybersecurity, Retail, Telecommunications, and more

TABLE 6.2. Distribution of interviews by industry sector (N=101 interviews across 86 companies)

services, each with distinct APIs, configuration patterns, and best practices, created a significant barrier to entry.

(3) **System complexity (24.8%)**: The cognitive load of managing microservices architectures with numerous dependencies, distributed state, and failure modes overwhelmed many teams.

(4) **Serverless-specific concerns (14.9%)**: Cold start latency, execution time limits,

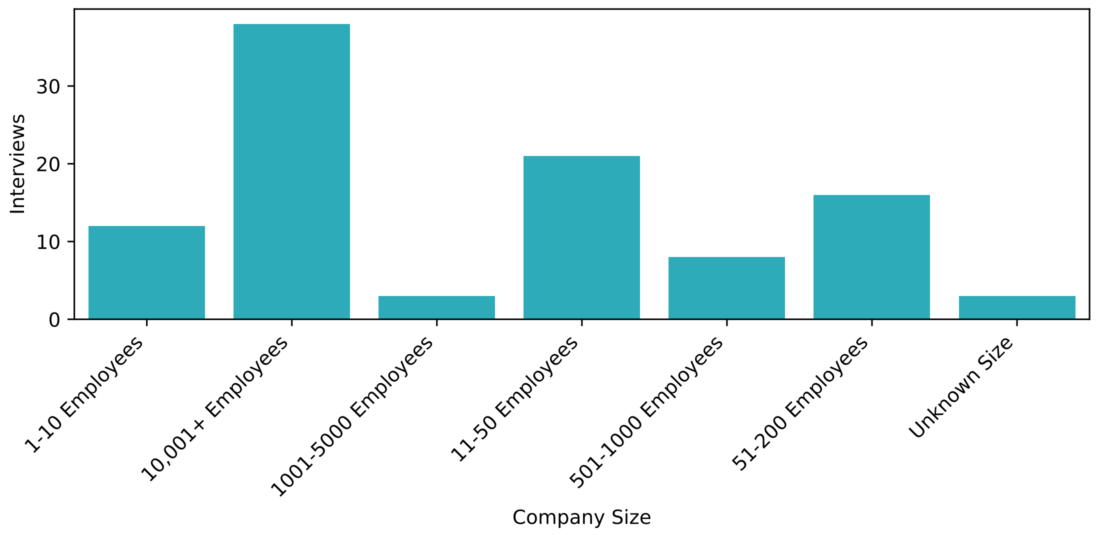


FIGURE 6.3. Distribution of interviews by company size. The sample included significant representation from both large enterprises (10,001+ employees) and small-to-medium companies (1–200 employees), providing insights across organizational scales.

and vendor lock-in specific to Function-as-a-Service platforms.

(5) **Security concerns (14.9%)**: Security configuration complexity, credential management, and compliance requirements added significant overhead, especially in regulated industries like finance and healthcare.

(6) **Integration challenges (11.9%)**: Connecting heterogeneous services (databases, message queues, APIs, functions) required significant custom integration code and debugging effort.

(7) **Observability limitations (11.9%)**: Inadequate visibility into application performance, resource utilization, and failure modes hindered troubleshooting and optimization.

(8) **Cost management (9.9%)**: Unpredictable costs and difficulty correlating resource usage with business value emerged as concerns, particularly for organizations running serverless or autoscaling workloads. The opacity of cost-performance trade-offs and lack of intuitive budget controls made it challenging to optimize spending.

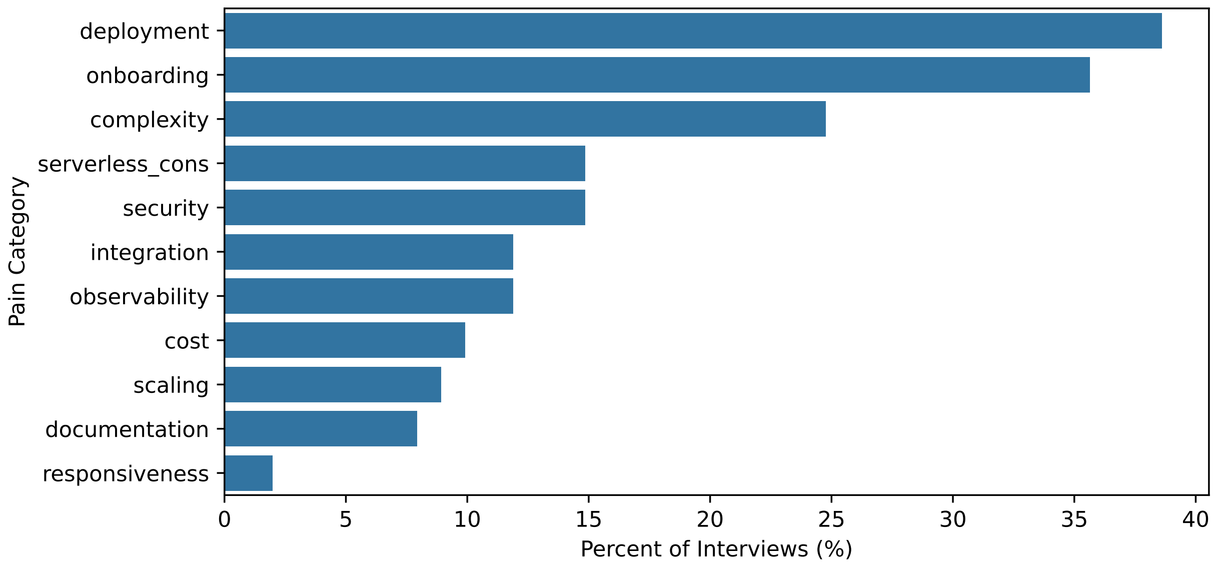


FIGURE 6.4. Frequency of pain points mentioned across 101 interviews. Deployment complexity and onboarding difficulty dominated, while cost management and security also featured prominently.

(9) **Scaling issues (8.9%)**: Difficulty managing autoscaling policies and handling load spikes, particularly with stateful workloads.

(10) **Documentation gaps (7.9%)**: Inadequate or outdated documentation for cloud services and internal platforms hindered developer productivity and increased reliance on tribal knowledge.

Importantly, these pain points often co-occurred. Our analysis revealed that complexity and onboarding challenges frequently appeared together, suggesting an underlying issue of *cognitive overload*—the sheer amount of knowledge required to effectively use modern cloud platforms.

6.3.2. Desired Outcomes and Expectations for Improvement

Beyond identifying pain points, we asked participants about their desired outcomes and expectations for improvement. We systematically categorized these expectations across thirteen dimensions, revealing clear priorities (Figure 6.5):

(1) **Productivity improvements (53.5%)**: The most prevalent expectation focused

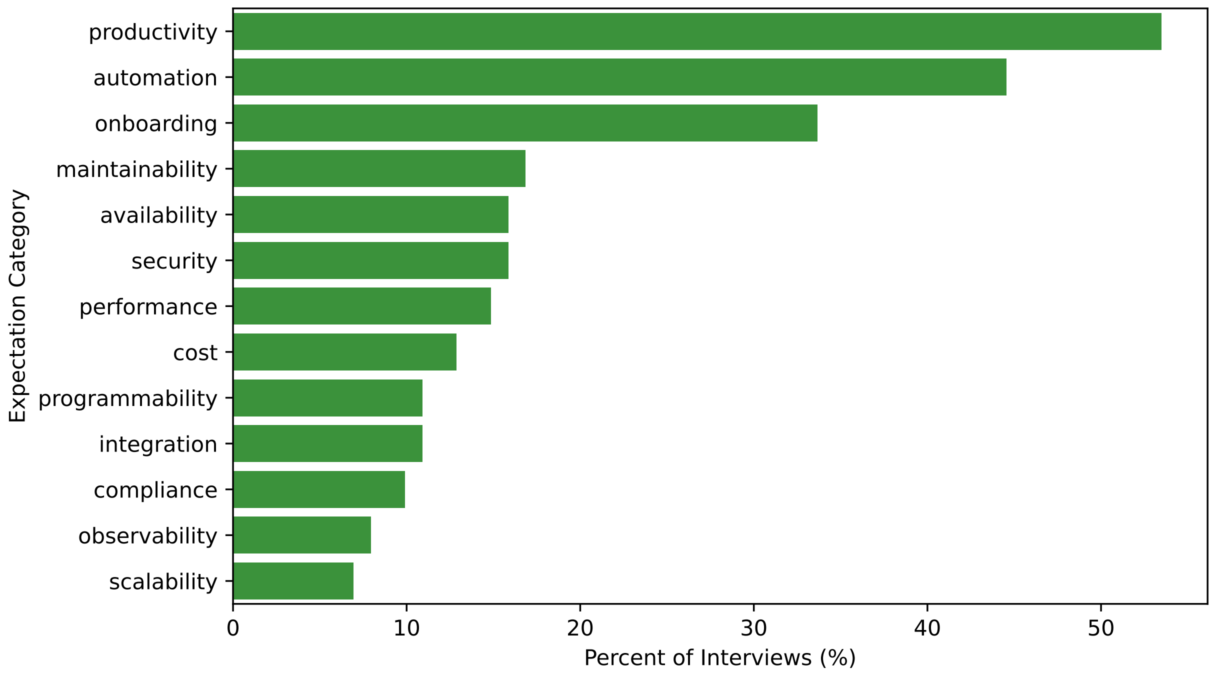


FIGURE 6.5. Frequency of expectations for improvement mentioned across 101 interviews. Productivity improvements, automation, and onboarding dominated, revealing that practitioners prioritize developer experience and operational efficiency over raw performance optimization.

on overall time-to-value improvements in development, deployment, and operations. Participants wanted faster time-to-market, reduced development cycles, and quantifiable efficiency gains across the entire software delivery lifecycle.

(2) **Automation (44.6%)**: Strong demand for increased automation in CI/CD pipelines, infrastructure provisioning, auto-scaling, and rollout/rollback processes. Participants wanted declarative, infrastructure-as-code approaches that eliminate manual steps and approval bottlenecks.

(3) **Onboarding improvements (33.7%)**: Expectations for significantly reduced ramp time for new engineers, from months to weeks. Participants emphasized the need for intuitive interfaces, comprehensive examples, and streamlined environment setup.

(4) **Maintainability (16.8%)**: Desire for reduced operational toil, safer updates and rollbacks, simplified patching, and fewer flaky tests or pipelines. Participants wanted systems that are easier to operate and maintain over time.

(5) **Availability and reliability (15.8%)**: Expectations for improved uptime, faster recovery from failures (lower MTTR), and more resilient systems with fewer outages.

(6) **Security (15.8%)**: Expectations for stronger security controls, better access management, data locality enforcement, and encryption—particularly acute in regulated industries.

(7) **Performance (14.9%)**: Improved runtime characteristics including lower latency, higher throughput, reduced cold starts, and better tail latency (p95/p99).

(8) **Cost optimization (12.9%)**: Expectations for lower cloud and operational costs, better ROI, reduced idle capacity, and more predictable spending.

(9) **Programmability (10.9%)**: Better developer ergonomics through simpler APIs and SDKs, reduced glue code, unified frameworks, and richer documentation with examples.

(10) **Integration (10.9%)**: Easier integration with legacy systems, third-party services, and internal platforms, with fewer API mismatches and simpler service composition.

(11) **Compliance (9.9%)**: Smoother alignment with regulatory requirements (PCI, HIPAA, ITAR, GDPR), faster governance approvals, and better audit trail capabilities.

(12) **Observability (7.9%)**: Improved end-to-end visibility through better tracing, logging, metrics, and dashboards for debugging and root cause analysis.

(13) **Scalability (6.9%)**: Better elasticity, horizontal/vertical scaling, multi-region support, and improved capacity management during traffic spikes.

Participants were notably less interested in raw performance gains than in *productivity*, *automation*, and *simplified onboarding*. This pattern reveals that practitioners prioritize developer experience and operational efficiency over technical performance optimization.

Takeaway: *Customer discovery reveals deployment complexity, onboarding difficulty, and system complexity as dominant pain points, while practitioners prioritize productivity and automation expectations over performance—demonstrating a clear demand for developer experience improvements.*

6.4. OaaS Technical Capabilities Addressing Market Needs

The customer discovery findings demonstrate compelling alignment between validated market pain points, practitioner expectations, and the technical contributions of OaaS presented in Chapters 3–5. This section examines how OaaS's core architectural innovations—unified object abstraction, declarative non-functional requirements, and edge-cloud orchestration—directly address the challenges identified in Section 6.3.

6.4.1. Unified Object Abstraction for Deployment and Onboarding

The top pain points—deployment complexity (38.6%), onboarding difficulty (35.6%), and system complexity (24.8%)—stem from fragmented tooling, service sprawl, and the cognitive load of coordinating multiple infrastructure abstractions. Practitioners reported spending significant time on glue code, configuration management, and cross-team coordination just to deploy applications.

The object abstraction presented in Chapter 3 directly addresses these challenges by consolidating compute (methods as serverless functions), state (attributes backed by databases or object storage), and workflow (dataflow/macro functions) into a unified deployment unit. Developers define application components using familiar object-oriented programming concepts in declarative specifications, while the platform automatically handles provisioning, configuration, and orchestration across FaaS runtimes, storage backends, and messaging infrastructure.

This architectural consolidation eliminates the fragmentation that drives deployment complexity, reduces the learning curve for new team members by providing a single consistent abstraction, and decreases system complexity by hiding heterogeneous infrastructure details behind a unified programming model. The declarative specification approach further reduces cognitive load by allowing developers to focus on application logic rather than infrastructure mechanics.

6.4.2. Declarative Non-Functional Requirements for QoS Guarantees

Practitioners identified performance (14.9%), availability and reliability (15.8%), and cost optimization (12.9%) as critical expectations, though cost management appeared less frequently as a direct pain point. The underlying concern across these dimensions was unpredictable behavior and opacity in resource-to-outcome mapping, particularly for serverless and autoscaling workloads where teams struggle to correlate spending with performance.

The non-functional requirement (NFR) interface presented in Chapter 4 addresses these challenges through declarative QoS specifications. Developers specify desired outcomes—throughput targets, availability levels, and consistency requirements—while the platform's Class Runtime Manager continuously monitors actual performance against declared SLAs and dynamically adjusts resource allocations to meet targets.

The *accept-or-reject* admission control model provides upfront validation: if an SLA cannot be met within specified constraints, the system rejects deployment with actionable feedback and alternative configurations rather than allowing deployment and discovering issues in production. This mechanism directly addresses availability concerns (preventing outages from under-provisioning), performance expectations (guaranteeing latency/throughput targets), and cost predictability (budget-aware admission control). The declarative approach eliminates manual performance tuning and resource provisioning, reducing operational toil identified in maintainability expectations (16.8%).

Takeaway: *OaaS core technical innovations—unified object abstraction and declarative NFR interface—directly map to validated market pain points, addressing deployment complexity through architectural consolidation and performance unpredictability through automated QoS management.*

6.5. Production Readiness and Commercialization Pathway

While Section 6.4 established technical validation, transforming OaaS from research prototype to production-ready commercial platform requires addressing critical gaps identified during customer discovery. This section outlines production requirements, target market positioning, and the pathway from research contribution to market-ready product.

6.5.1. Production Readiness Requirements

Customer interviews revealed three priority areas for adoption. **Security and compliance** emerged as particularly acute in regulated industries like finance and healthcare, requiring defense-in-depth strategies including robust identity and access management (IAM), multi-tenant isolation, encryption at rest and in transit, and audit logging. Several financial services participants explicitly stated that security certification (SOC 2, PCI-DSS) would be prerequisites for evaluation.

Developer experience demands go beyond the unified abstraction to include idiomatic SDKs for popular languages (Python, JavaScript, Java, Go), comprehensive documentation with real-world examples, quickstart templates, and a library of reusable object patterns. Onboarding friction (35.6% of pain points) can only be fully addressed through polished tooling and learning resources that enable developers to be productive within hours rather than days.

Observability and integration requirements reflect operational realities. Observability limitations (11.9%) and integration challenges (11.9%) necessitate distributed tracing across object invocations, structured logging with correlation IDs, metrics dashboards for resource utilization and SLA compliance, and anomaly detection for performance degradation. Integration expectations (10.9%) demand seamless connectivity with existing CI/CD pipelines, monitoring tools (Datadog, Prometheus, Grafana), and legacy systems through standard protocols and adapters.

6.5.2. Validated Target Market Segments

Interview data validates three primary market segments for commercialization. Technology sector SMEs and startups (33 interviews across 20+ organizations) demonstrated acute sensitivity to deployment complexity and onboarding friction, representing the most suitable early-stage adoption target given their tolerance for emerging platforms and lower security/compliance barriers. Research institutions (17 interviews, 13 organizations) prioritized rapid prototyping and multi-cloud flexibility over enterprise operational features, offering a complementary early adopter segment. Enterprise accounts in regulated industries (12 interviews in finance and healthcare) emphasized security, compliance, and observability alongside operational efficiency—requirements that necessitate addressing the production readiness gaps identified in Section 6.5 before large-scale enterprise pursuit. This staged approach focuses initial commercialization efforts on technology SMEs and research institutions while building the enterprise-grade capabilities required for regulated industry adoption. Section 6.6 presents the complete Business Model Canvas that operationalizes these validated segments into a coherent go-to-market strategy with detailed customer personas, acquisition channels, and phased execution plans.

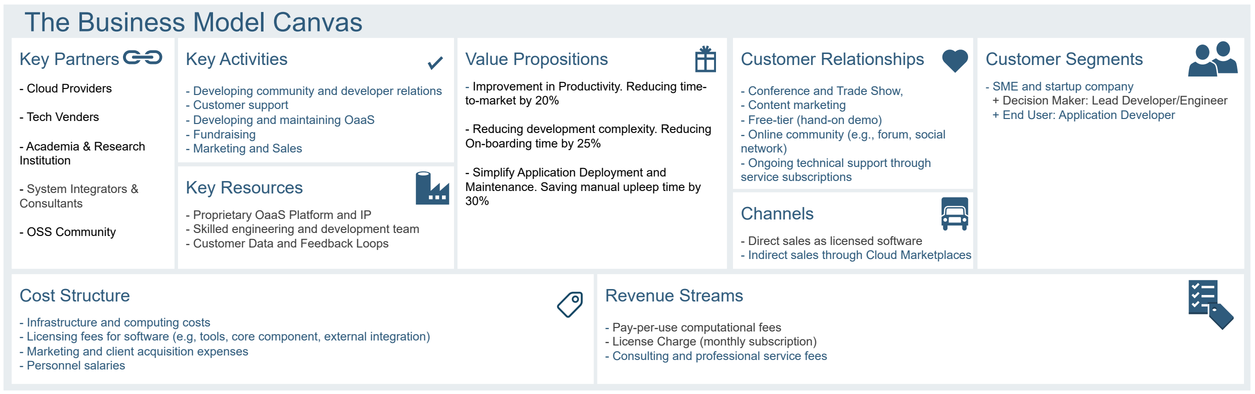
Takeaway: *Transforming OaaS from research prototype to commercial platform requires addressing security, developer experience, and observability gaps for enterprise adoption, while validated target segments—technology SMEs, research institutions, and regulated enterprises—provide evidence-based foundation for the commercialization strategy detailed in Section 6.6.*

6.6. Business Model and Go-to-Market Strategy

Building on the validated pain points and value propositions from customer discovery, this section presents the business model framework that translates OaaS technical capabilities into a sustainable commercial venture. The Business Model Canvas (BMC) [93] synthesizes insights from our 101 interviews into a coherent strategy for market entry, revenue generation, and customer acquisition.

6.6.1. Business Model Canvas

Figure 6.6 presents the OaaS Business Model Canvas, developed iteratively throughout the I-Corps program through hypothesis testing and customer feedback. The canvas articulates nine interdependent components that define how OaaS creates, delivers, and captures value.


The Business Model Canvas

Key Partners
- Cloud Providers
- Tech Venders
- Academia & Research Institution
- System Integrators & Consultants
- OSS Community

Key Activities
- Developing community and developer relations
- Customer support
- Developing and maintaining OaaS
- Fundraising
- Marketing and Sales

Key Resources
- Proprietary OaaS Platform and IP
- Skilled engineering and development team
- Customer Data and Feedback Loops

Value Propositions
- Improvement in Productivity. Reducing time-to-market by 20%
- Reducing development complexity. Reducing On-boarding time by 25%
- Simplify Application Deployment and Maintenance. Saving manual upleep time by 30%

Customer Relationships
- Conference and Trade Show,
- Content marketing
- Free-tier (hand-on demo)
- Online community (e.g., forum, social network)
- Ongoing technical support through service subscriptions

Channels
- Direct sales as licensed software
- Indirect sales through Cloud Marketplaces

Customer Segments
- SME and startup company
  + Decision Maker: Lead Developer/Engineer
  + End User: Application Developer

Cost Structure
- Infrastructure and computing costs
- Licensing fees for software (e.g, tools, core component, external integration)
- Marketing and client acquisition expenses
- Personnel salaries

Revenue Streams
- Pay-per-use computational fees
- License Charge (monthly subscription)
- Consulting and professional service fees

FIGURE 6.6. OaaS Business Model Canvas developed through NSF I-Corps customer discovery.

Figure 6.7 visualizes the OaaS customer ecosystem, revealing the multi-sided market dynamics and stakeholder relationships that shape platform adoption and value delivery. The diagram illustrates how OaaS creates value for multiple interconnected personas while navigating influence patterns from infrastructure providers, consultants, and key opinion leaders who shape purchasing decisions.

Customer Segments. Our customer discovery identified SME and startup companies as the primary target segment, with a complex buying ecosystem involving multiple interconnected stakeholders (Figure 6.7). The ecosystem reveals three internal buyer personas and four external influencer categories, each shaping platform adoption through distinct evaluation criteria and decision authority.

Financial Decision Makers (CFOs or budget holders) control purchasing authority and require quantified ROI evidence—minimum 20% productivity improvements, and reduced time-to-market. *Technical Decision Makers* (Lead Engineers, Engineering Managers) evaluate technical fit, validate feasibility, and champion adoption based on deployment complexity reduction and toolchain integration. Their approval is prerequisite to financial sign-off. *End Users* (Application Developers) prioritize developer experience, intuitive APIs, and reduced cognitive load. While developers rarely control purchasing decisions, negative experiences can derail adoption even after executive approval.

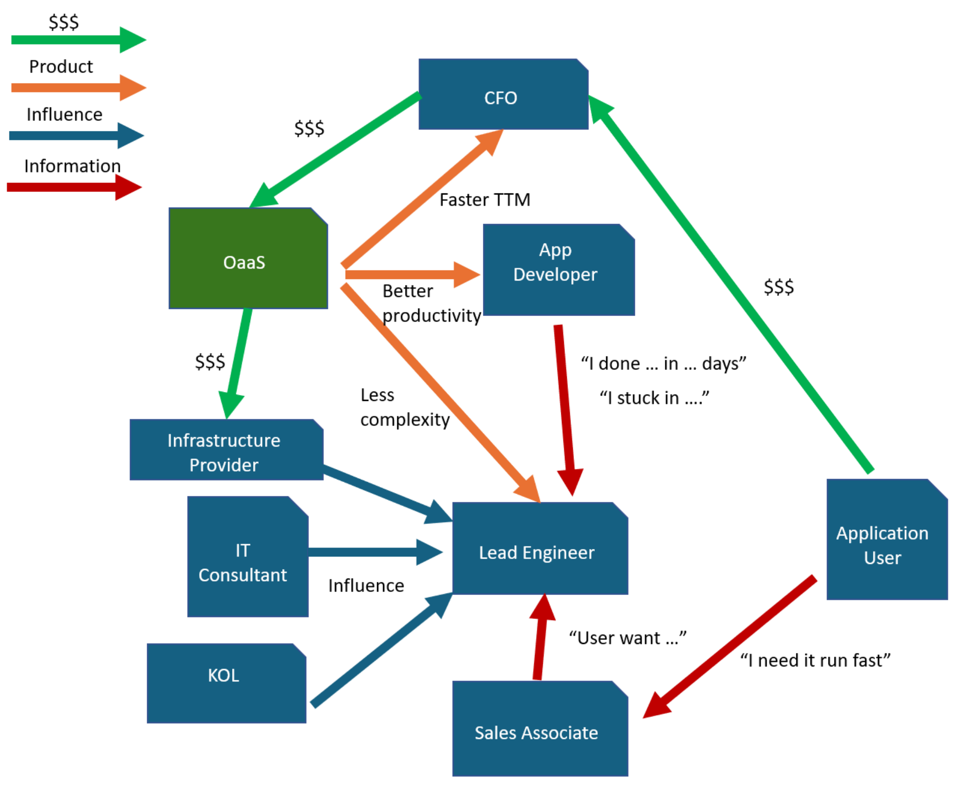


Figure 6.7. OaaS customer ecosystem showing stakeholder relationships and value flows. Green arrows indicate financial flows, orange arrows represent value delivery (faster time-to-market, improved productivity, reduced complexity), blue arrows show influence patterns, and red arrows indicate requirements and feedback flows from end users.

External influencers include *Infrastructure Providers* (AWS, Azure, Google Cloud) creating partnership requirements, *IT Consultants and System Integrators* recommending platforms during modernization projects, *Key Opinion Leaders* shaping developer mindshare through thought leadership, and *Application End Users* providing performance feedback loops. This multi-stakeholder ecosystem necessitates staged engagement: developers discover through hands-on trials, technical decision makers validate through architecture assessments, and financial decision makers approve based on quantified outcomes. Technology SMEs and startups are prioritized as early adopters given their tolerance for emerging platforms, shorter decision cycles, and lower compliance barriers.

Value Propositions. The canvas articulates three core value propositions validated through customer interviews, each tied to specific pain-expectation pairs identified during discovery. First, customers expressed that any solution addressing deployment complexity must deliver at least 20% improvement in development productivity and meaningful acceleration in time-to-market; OaaS targets this expectation through unified abstraction that eliminates fragmented tooling and glue code. Second, customers indicated that reducing onboarding difficulty requires solutions that lower the skill barrier for new hires by over 25%; OaaS addresses this expectation through declarative, object-oriented programming models that hide infrastructure details. Third, customers specified that simplifying application deployment must reduce manual maintenance efforts by approximately 30%; OaaS targets this expectation through automated provisioning, configuration, and lifecycle management. These quantitative thresholds represent the minimum value customers require from any platform claiming to solve their core pain points, establishing clear benchmarks against which OaaS value delivery will be measured.

Channels. Customer acquisition follows a multi-channel strategy aligned with validated buyer behavior, visualized in Figure 6.8. The diagram illustrates three distinct go-to-market pathways validated through customer interviews, with cloud marketplaces as the primary focus for our SME and startup target segment.

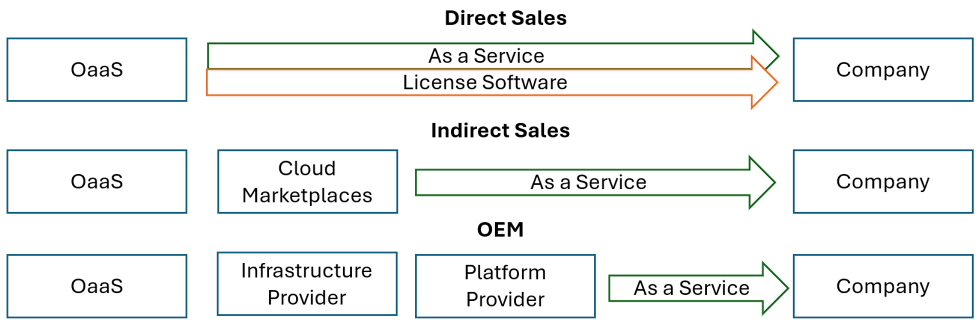


Figure 6.8. OaaS product channel strategy showing three validated go-to-market pathways.

Indirect sales through cloud marketplaces—such as AWS Marketplace, Azure Marketplace, and Google Cloud Marketplace—serves as the primary go-to-market channel for technology SMEs and startups. Customer interviews revealed strong preference for marketplace-based discovery because it provides immediate discoverability, frictionless procurement through existing cloud billing, rapid trial-to-purchase conversion, and built-in trust from established providers. The managed service model addresses deployment complexity while preserving flexibility to migrate to self-hosted deployments.

Direct sales as licensed software provides a complementary channel for enterprise customers requiring deep customization, on-premise deployments, and multi-year licensing agreements. This channel enables technical engagement for complex integration requirements, custom SLA negotiations, and dedicated support arrangements that enterprise buyers expect. While customers explicitly indicated preference for self-hosted deployment models to preserve architectural flexibility and avoid vendor lock-in, this channel primarily targets larger organizations with established procurement processes rather than our primary SME segment.

OEM partnerships through infrastructure providers and platform providers represent a third channel for embedded distribution. Infrastructure providers (AWS, Azure, GCP) can bundle OaaS capabilities into their native services, while platform providers can integrate OaaS as middleware for their customers. This channel extends reach to customers who prefer turnkey solutions but requires careful partnership structuring to maintain OaaS brand identity and customer relationships.

Notably, *direct sales as a fully managed platform* was explicitly rejected by customers due to concerns about vendor lock-in, limited customization, and loss of infrastructure control. This feedback shaped our channel strategy to prioritize licensed software and marketplace distribution over proprietary hosting.

Customer Relationships. Building trust and enabling success requires ongoing engagement beyond initial sales. *Content marketing*—technical blogs, architecture guides, case studies, and conference talks—establishes thought leadership and educates potential customers on OaaS capabilities. *Ongoing technical support through service subscriptions* provides tier-based assistance, with premium tiers offering dedicated support engineers for enterprise customers. *Online community engagement* through forums, Slack channels, and social media fosters peer-to-peer knowledge sharing, early feedback loops, and developer advocacy. *Conference and trade show participation* enables face-to-face relationship building and live demonstrations that address complex technical questions. *Free-tier or hands-on demos* lower adoption barriers by allowing developers to experiment with OaaS without procurement approvals, accelerating evaluation cycles.

Revenue Streams. The business model employs a diversified revenue strategy with three complementary streams. *License charges via monthly subscriptions* provide predictable recurring revenue through tiered plans based on usage volume, number of developers, or deployment scale. This aligns with enterprise software procurement cycles and enables upselling as adoption grows. *Pay-per-use computational fees* offer consumption-based pricing for compute, storage, and data transfer, appealing to cost-conscious startups and variable-workload customers. This model directly addresses cost optimization expectations by ensuring customers pay only for resources consumed. *Consulting and professional services fees* generate additional revenue through implementation support, migration assistance, custom integration development, and training programs, particularly for large enterprise deployments requiring hands-on expertise.

Key Resources. Successful commercialization depends on strategic assets developed and acquired over time. The *proprietary OaaS platform and intellectual property*—including patents, trade secrets, and core algorithms—form the defensible technical foundation. A *skilled engineering and development team* with expertise in distributed systems, serverless computing, and developer tooling is essential for continuous innovation and production readiness. *Customer data and feedback loops* captured through telemetry, support tickets, and community interactions inform product roadmap prioritization and feature development.

Key Activities. Day-to-day operations focus on five strategic pillars. *Developing community and developer relations* involves nurturing the ecosystem, organizing hackathons, publishing documentation, and engaging with early adopters to build advocacy and gather feedback. *Customer support* ensures rapid issue resolution, high satisfaction, and ongoing technical assistance through tiered service levels. *Developing and maintaining OaaS* includes continuous platform enhancements, bug fixes, performance optimizations, and security updates aligned with customer expectations. *Fundraising* from venture capital, government grants (e.g., NSF SBIR/STTR), or strategic investors provides capital for scaling operations. *Marketing and sales* drive demand generation, lead qualification, and conversion through targeted campaigns aligned with validated customer segments.

Key Partners. Ecosystem collaboration amplifies reach and capabilities through strategic partnerships that balance mutual value creation with inherent risks. Tables 6.3 and 6.4 analyze the partnership dynamics across four key partner categories, articulating value exchange and risk considerations.

Partner Category	**OaaS Provides**	**OaaS Receives**
Cloud Providers (AWS, Azure, GCP)	Drives core service usage; adds value to their platform through expanded use cases	Infrastructure access; marketplace distribution; co-marketing opportunities; technical support
System Integrators & Consultants	Innovative client solutions; partner program with training, support, and co-selling enablement	Enterprise sales channel; implementation and migration services; customer referrals
Tech Vendors (IoT Devices, Edge Infrastructure)	Ecosystem expansion through complementary integrations; broader market access	Ecosystem expansion; broader use case coverage; technical partnerships
Academia & OSS Community	Research platform; OSS contributions; validation testbed; student engagement opportunities	Talent pipeline; cutting-edge innovation; developer credibility; community-driven features

Table 6.3. Partnership value exchange: what OaaS provides and receives from key partner categories

Cloud Providers (AWS, Azure, Google Cloud) offer infrastructure discounts, co-selling opportunities, and technical support for joint customers, creating strategic alliances that

Partner Category	Partnership Risks and Mitigation Considerations
Cloud Providers	A provider could leverage partnership insights to launch a competing native service, requiring careful IP protection and strategic positioning of unique value propositions.
System Integrators & Consultants	Natural hesitancy to recommend unproven technologies that could put their reputation at risk with clients, requiring substantial evidence of reliability, ROI, and production-readiness before securing recommendations.
Tech Vendors	Poor integration quality from partners could damage OaaS user experience; IoT device manufacturers and edge infrastructure vendors showed fragmentation and proprietary protocols during discovery, requiring careful partner vetting and integration quality standards.
Academia & OSS Community	Research collaborations are speculative and may not yield direct commercial value; requires balancing open-source community engagement with proprietary competitive advantages.

Table 6.4. Partnership risk assessment and mitigation strategies for key partner categories

drive mutual platform value. *System Integrators and Consultants* extend market reach by recommending OaaS to their enterprise clients during cloud migration or modernization projects. *Tech Vendors* provide complementary technologies and integration opportunities, expanding both ecosystems, though careful partner vetting is essential to maintain user experience quality. *Academia and Research Institutions* provide access to cutting-edge research, student talent, and validation testbeds for novel features, while *OSS Community* contributors accelerate development through community-driven enhancements, plugins, and integrations.

Cost Structure. Operating expenses fall into four primary categories. *Infrastructure and computing costs* for hosting the platform, running CI/CD pipelines, and supporting customer workloads scale with adoption. *Licensing fees for software*—including third-party

tools, core components, and external integrations—add recurring overhead. *Marketing and client acquisition expenses* cover digital advertising, event sponsorships, content production, and sales enablement. *Personnel salaries* for engineering, sales, marketing, and operations represent the largest fixed cost.

Cloud Providers (AWS, Azure, Google Cloud) offer infrastructure discounts, co-selling opportunities, and technical support for joint customers, creating strategic alliances that drive mutual platform value. However, the risk remains that a provider could leverage insights from the partnership to launch a competing native service. *System Integrators and Consultants* extend market reach by recommending OaaS to their enterprise clients during cloud migration or modernization projects, though their natural hesitancy to recommend unproven technologies requires substantial evidence of reliability and ROI. *Tech Vendors* provide complementary technologies and integration opportunities, expanding both ecosystems, though IoT device manufacturers and edge infrastructure vendors received negative feedback during discovery due to fragmentation and proprietary protocols that could compromise user experience. *Academia and Research Institutions* provide access to cutting-edge research, student talent, and validation testbeds for novel features, while *OSS Community* contributors accelerate development through community-driven enhancements, plugins, and integrations, though these collaborations remain speculative with uncertain commercial returns.

Cost Structure. Operating expenses fall into four primary categories. *Infrastructure and computing costs* for hosting the platform, running CI/CD pipelines, and supporting customer workloads scale with adoption. *Licensing fees for software*—including third-party tools, core components, and external integrations—add recurring overhead. *Marketing and client acquisition expenses* cover digital advertising, event sponsorships, content production, and sales enablement. *Personnel salaries* for engineering, sales, marketing, and operations represent the largest fixed cost.

6.6.2. Revenue Model and Pricing Strategy

The revenue model balances accessibility for startups with scalability for enterprises. The *subscription-based licensing* tier offers predictable monthly or annual pricing with features segmented into Free (individual developers, limited usage), Professional (small teams), and Enterprise (large organizations, custom pricing). This model aligns with customer expectations for cost predictability while enabling land-and-expand growth strategies. The *usage-based pricing* tier charges for actual compute time, storage consumption, and data transfer, appealing to customers with variable workloads who prioritize cost optimization. Hybrid plans combining base subscriptions with usage overages provide flexibility and revenue upside during traffic spikes. *Consulting and professional services* are offered for implementation support, migration assistance, and custom development, generating high-margin revenue while deepening customer relationships and accelerating adoption.

Pricing strategy employs a hybrid approach combining value-based and competitive-based methodologies. Value-based pricing is determined from quantified time and cost savings on the customer side, including reduced developer hours, faster time-to-market, decreased infrastructure complexity, and lower operational overhead. These savings translate into concrete ROI metrics that justify premium pricing for customers realizing significant productivity gains. Competitive-based pricing is positioned relative to commercial serverless platforms (AWS Lambda, Azure Functions, Google Cloud Functions) to ensure market competitiveness while emphasizing differentiated value from unified abstractions, declarative SLAs, and edge-cloud capabilities that traditional FaaS offerings lack. This dual-anchor approach ensures OaaS pricing remains justified by delivered value while staying competitively positioned within the serverless market landscape. Customers expressed willingness to pay premiums for platforms that demonstrably reduce onboarding time, deployment complexity, and operational toil.

6.6.3. Go-to-Market Strategy

The go-to-market strategy prioritizes rapid adoption in validated customer segments while building ecosystem momentum. Phase 1 targets *early adopters* in technology startups, research institutions, and consulting firms identified during I-Corps as most receptive to innovation and least constrained by legacy processes. Tactics include launching a free tier with generous usage limits, publishing comprehensive documentation and tutorials, and securing 5–10 lighthouse customers willing to provide testimonials and case studies. Developer evangelism through conference talks, blog posts, and open-source contributions establishes credibility and drives organic traffic.

Phase 2 expands into *small-to-medium technology companies* (11–200 employees) facing acute deployment complexity and onboarding challenges. Cloud marketplace listings (AWS, Azure, Google Cloud) serve as the primary channel, enabling frictionless discovery, one-click deployments, and streamlined procurement through existing cloud billing relationships. Marketplace visibility is amplified through strategic co-marketing with cloud providers, featured placement in serverless and developer tools categories, and integration with native cloud services. Strategic partnerships with system integrators and consultants provide referral channels and professional services leverage. Customer success programs focus on reducing time-to-value, publishing ROI metrics, and identifying expansion opportunities within existing accounts.

Phase 3 pursues *enterprise accounts* in financial services, healthcare, and logistics where security, compliance, and observability requirements are well-defined. Enterprise sales cycles are longer but yield higher contract values and multi-year commitments. Dedicated account teams, custom SLAs, and on-premise deployment options address enterprise buying criteria. Success metrics include 20% month-over-month user growth in Phase 1, $500K ARR by end of Phase 2, and $5M ARR by end of Phase 3.

Channel strategy prioritizes cloud marketplaces as the primary acquisition channel for SMEs and startups, leveraging their existing cloud infrastructure commitments and purchasing behaviors. This marketplace-first approach addresses the validated preference for frictionless procurement and rapid trial-to-value conversion identified during customer discovery. Direct sales supplement marketplace distribution for enterprise deals requiring customization, complex integrations, and relationship building. Community-led growth through open-source contributions, developer advocacy, and technical content attracts bottom-up adoption that influences top-down purchasing decisions and drives marketplace traffic. Strategic OEM partnerships with platform providers remain exploratory but could unlock distribution leverage if properly structured to avoid channel conflict.

Takeaway: *Validated market segments provide clear prioritization: technology SMEs and startups offer the optimal entry point given their acute pain points, tolerance for emerging platforms, and lower compliance barriers, while enterprise requirements define the production roadmap.*

6.7. Summary

This chapter presented commercialization validation from NSF I-Corps National (Summer 2025 Cohort 3) through 101 interviews across 86 organizations. Deployment complexity (38.6%), onboarding difficulty (35.6%), and system complexity (24.8%) emerged as dominant pain points, while practitioners prioritized productivity, automation, and faster onboarding—validating OaaS's focus on unified abstractions and declarative interfaces. Customer discovery validated OaaS's core capabilities while identifying security, compliance, and observability gaps critical for production readiness.

Three market segments emerged: technology SMEs and startups (33 interviews), research institutions (17 interviews), and enterprise accounts (12 interviews). Strategic analysis identified SMEs and startups as the primary early-stage target given their tolerance for emerging platforms and lower compliance barriers, with research institutions as a complementary segment. The Business Model Canvas establishes a staged commercialization strategy with diversified revenue streams (subscription licensing, usage-based pricing, professional services), strategic partnerships across four key categories (cloud providers, system integrators, tech vendors, and academia), and phased go-to-market execution targeting SME/startup organizations before enterprise expansion. The convergence of validated pain points, practitioner expectations, and demonstrated problem-solution fit establishes an evidence-based pathway from research prototype to commercial platform focused on the SME and startup market.

CHAPTER 7

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

7.1. Conclusion

Cloud computing has fundamentally transformed how applications are developed and deployed, yet a persistent gap remains between the promise of serverless computing and its practical realization. This dissertation addressed this gap through empirical validation and technical innovation, establishing the Object-as-a-Service (OaaS) paradigm as a path toward simplified, unified cloud-native platforms.

Grounded in empirical evidence from an initial practitioner interview study (Chapter 1, 21 participants) and validated through NSF I-Corps National customer discovery (Chapter 6, 101 interviews across 86 organizations), this work demonstrates that infrastructure complexity imposes substantial tax on developer productivity. Both studies consistently identified complexity in deployment, onboarding, and system management as the most pervasive challenges—particularly for small enterprises, domain experts, and resource-constrained organizations. Critically, practitioners prioritize productivity and automation over cost optimization, contradicting much academic research that emphasizes cost savings.

This dissertation makes five major contributions that collectively advance the state-of-the-art in serverless computing:

1. Empirical Validation: Through three complementary empirical studies—an initial practitioner interview study (Chapter 1, 21 participants), a human study evaluating developer experience (Chapter 5, 39 participants), and NSF I-Corps National customer discovery (Chapter 6, 101 interviews)—we identified and quantified critical pain points, practitioner expectations, and developer productivity gains across diverse organizational contexts. The studies reveal that practitioners consistently prioritize operational maintainability, development simplicity, and onboarding efficiency over cost optimization. Non-technical organizations face significantly higher development complexity rates, while the human study quantified 31% faster task completion with unified abstractions, validating that simplified interfaces deliver measurable productivity improvements.

2. Object-as-a-Service Paradigm (Chapter 3): OaaS unifies resource, state, and workflow management into a single object-oriented abstraction, addressing fragmentation by consolidating FaaS compute, managed databases, and orchestrators into one coherent interface. The Oparaca prototype demonstrates that this unified abstraction streamlines cloud-native programming with negligible performance overhead while achieving scalability comparable to state-of-the-art systems.

3. SLA-Driven OaaS Paradigm (Chapter 4): Declarative Non-functional Requirement (NFR) management enables performance optimization through intuitive, high-level specifications. Developers can specify outcomes—availability, throughput, consistency, latency targets—without understanding low-level mechanisms, directly addressing practitioner demands for service quality assurance and simplified programmability.

4. OaaS across Edge-Cloud (Chapter 5): EdgeWeaver extends OaaS across the edge-cloud continuum with SLA-driven placement, connectivity-aware invocation, and NFR trade-off management, addressing responsiveness challenges and enabling deployment in geo-distributed, resource-constrained, and connectivity-challenged environments critical for IoT and edge computing scenarios.

5. Commercialization Validation (Chapter 6): NSF I-Corps National customer discovery corroborates and expands the initial practitioner study, validating market demand across diverse sectors. The findings confirmed deployment complexity, onboarding difficulty, and system complexity as dominant pain points, with industry-specific patterns revealing acute security concerns in regulated industries, elevated onboarding challenges in education, and observability gaps in healthcare. These insights establish a staged commercialization pathway with technology SMEs and startups as the primary early-stage target given their tolerance for emerging platforms and lower compliance barriers, complemented by research institutions, while deferring enterprise accounts in regulated industries until production readiness requirements (security, compliance, observability) are addressed.

7.1.1. Limitations and Scope

While this dissertation makes significant contributions to serverless computing research and establishes the viability of the Object-as-a-Service paradigm, several limitations constrain the scope and generalizability of the findings:

Dataflow Function Limitations: While Chapter 3 introduced dataflow (macro) functions as a mechanism to chain operations across objects via declarative data dependencies, the dissertation does not extend this capability with formal guarantees across the edge-cloud continuum. The complexity of ensuring end-to-end consistency, fault tolerance, and latency bounds for dataflow executions that span intermittently connected edge and cloud tiers remains unaddressed. Consequently, Chapters 4 and 5 focus on individual object invocations rather than guaranteed workflow orchestration.

Runtime Implementation Trade-offs: The Oparaca and EdgeWeaver prototypes leverage Knative as the underlying serverless runtime, simplifying implementation by reusing existing container orchestration and auto-scaling mechanisms. However, this design choice introduces performance overhead from networking, and resource management layers. The dissertation does not explore alternative serverless runtime implementations—such as WebAssembly-based execution—that could potentially reduce invocation latency and resource footprint at the cost of increased implementation complexity.

Multi-tenancy and Network Isolation: While the Class Runtime architecture (Chapters 4 and 5) supports multi-tenancy through deployment of separate class runtimes per tenant, the dissertation does not comprehensively address network isolation mechanisms. Specifically, the integration between the Messaging Infrastructure and tenant-specific network policies, including traffic segregation, quota enforcement, and secure inter-tenant communication boundaries, remains underspecified. This gap limits deployment in environments requiring strong isolation guarantees.

Security Scope: Security was not a primary focus of this dissertation. Comprehensive security mechanisms, including encryption-at-rest for object state, secure key management, audit logging for compliance, and defense against common attack vectors—are not deeply explored. The dissertation prioritizes demonstrating the viability of unified abstractions and declarative NFR management rather than establishing production-grade security hardening.

Class Runtime Updates and Migration: The dissertation does not detail mechanisms for seamlessly migrating Class Runtimes when class definitions are updated. This includes handling live object state migration, maintaining service availability during transitions, ensuring data consistency across old and new runtime versions, and managing rollback scenarios when updates fail. Consequently, the current prototypes assume relatively static class definitions or require service interruption during updates—a limitation for continuously evolving production applications.

These limitations establish boundaries for the dissertation's claims while identifying concrete research directions addressed in Section 7.2. Importantly, they do not invalidate core contributions—the unified object abstraction, declarative NFR management, and edge-cloud extension demonstrate viability at research scale and establish architectural patterns for production systems. However, they highlight that transitioning OaaS from research prototype to enterprise-grade platform requires systematic work across security hardening, multi-region deployment, comprehensive tooling, and domain-specific validation.

7.2. Future Research Directions

The limitations identified in Section 7.1.1 establish immediate technical challenges, while empirical findings from customer discovery (Chapter 6) reveal broader adoption barriers. Future research must address both dimensions: resolving technical gaps in the current prototypes while expanding OaaS capabilities for enterprise deployment. We organize this research agenda as nested milestones (represented as M_x) that build upon the dissertation's foundation (Figure 7.1). Milestone M1 directly addresses the technical limitations before progressively expanding toward production readiness and transformative capabilities in M2-M4.

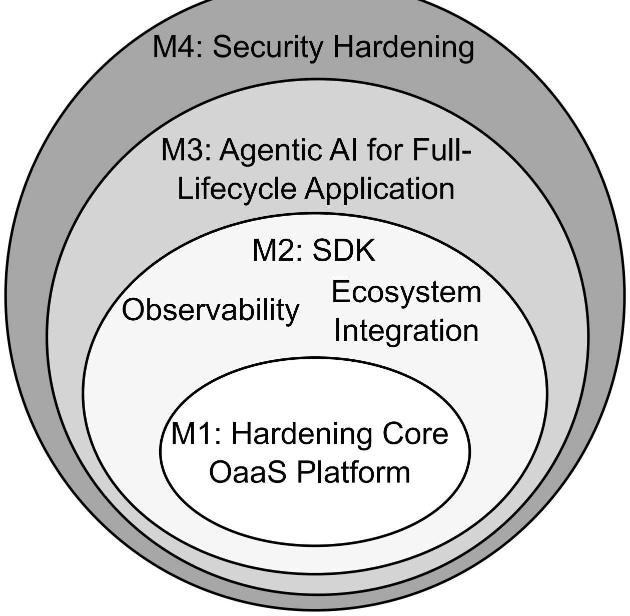

M4: Security Hardening
M3: Agentic AI for Full-Lifecycle Application
M2: SDK
Observability
Ecosystem Integration
M1: Hardening Core OaaS Platform

FIGURE 7.1. OaaS future research roadmap showing nested milestones from core platform hardening (M1) through ecosystem integration (M2) and agentic AI (M3) to security hardening (M4), with observability as a cross-cutting concern.

7.2.1. M1: Addressing Technical Limitations and Core Platform Hardening

This foundational milestone resolves the technical gaps acknowledged in Section 7.1.1 while establishing production readiness for enterprise adoption. Research challenges span immediate technical limitations and broader platform capabilities. We first address core technical limitations before expanding to broader production readiness:

(i) Dataflow Orchestration with Guarantees: Extending macro functions (Chapter 3) to provide end-to-end consistency, fault tolerance, and latency bounds across edge-cloud deployments. Research challenges include developing distributed transaction protocols that maintain ACID properties over intermittently connected tiers, designing compensation mechanisms for partial failures in multi-step workflows, and establishing latency prediction models that account for network variability between edge and cloud. This extension would enable developers to declaratively specify workflow-level SLAs while the platform automatically

handles failure recovery and performance optimization across the continuum.

(ii) Optimized Serverless Runtime: Exploring alternatives to Knative-based implementations to reduce invocation overhead. Research directions include custom lightweight container runtimes that minimize networking layers and resource management overhead, WebAssembly-based execution engines offering microsecond-scale cold starts and fine-grained sandboxing, and hybrid approaches that dynamically select execution strategies based on workload characteristics. Critical challenges involve maintaining auto-scaling and multi-tenancy guarantees while achieving performance improvements, developing migration paths from container-based deployments, and ensuring compatibility with existing FaaS ecosystems.

(iii) Network Isolation and Multi-tenancy: Comprehensively addressing integration between Messaging Infrastructure (MI) and tenant-specific network policies. Research challenges include designing policy specification languages that express traffic segregation rules, quota limits, and inter-tenant communication boundaries; implementing efficient enforcement mechanisms that validate every object invocation against tenant policies with minimal latency overhead; and developing monitoring systems that detect policy violations and unauthorized access patterns in real-time. Successfully addressing these challenges enables deployment in shared infrastructure environments requiring strong isolation guarantees.

(iv) Seamless Class Runtime Migration: Developing mechanisms for zero-downtime updates when class definitions evolve. Research directions include live migration protocols that transfer object state between runtime versions while maintaining consistency guarantees, versioning strategies supporting simultaneous operation of multiple class versions during gradual rollout, automated compatibility testing that validates new runtime versions against production workloads, and rollback mechanisms enabling instant reversion when updates cause failures. Critical challenges involve minimizing migration-induced latency spikes, preserving in-flight invocations during transitions, and managing data schema evolution across runtime versions.

Beyond addressing these immediate technical limitations, the milestone expands platform capabilities for production readiness:

(v) Enhanced SLA Support: Expanding the NFR interface beyond current performance metrics (availability, throughput, consistency, locality) to encompass security policies as first-class requirements, cost constraints with budget-aware admission control, and regulatory compliance specifications—requiring expressive yet accessible specification languages and efficient runtime enforcement mechanisms.

(vi) Multi-region Deployment: Enabling geo-distributed object placement with cross-region state management, consistency protocols that balance performance against data durability guarantees, and latency-aware placement algorithms that optimize for user proximity while respecting data sovereignty constraints.

(vii) Production-grade Reliability: Implementing comprehensive fault tolerance across distributed components, automated failure detection and recovery with minimal downtime, and formal verification of correctness properties to ensure platform behavior matches specifications.

(viii) Performance Optimization at Hyperscale: Developing efficient scheduling algorithms that minimize resource fragmentation, intelligent resource packing to maximize utilization, and cold-start elimination techniques that maintain sub-second response times even during traffic spikes.

This milestone establishes the robust technical foundation required for enterprise adoption in regulated industries, directly responding to both the dissertation's acknowledged limitations and production readiness requirements identified during customer discovery. Successfully addressing these challenges transforms OaaS from research prototype to production-capable platform while maintaining the core benefits of unified abstraction and declarative NFR management.

7.2.2. M2: SDK, Ecosystem Integration, and Observability

Building on the hardened platform, this milestone expands developer experience through comprehensive tooling and ecosystem integration. The first research thrust focuses on *developer productivity tools*: idiomatic language SDKs for Python, JavaScript, Java, and Go that provide intelligent code completion and type safety; local development environments that simulate production behavior for rapid iteration; and migration tooling to simplify transitions from existing FaaS platforms. The second thrust targets *operational integration*: CI/CD pipeline integration with automated testing and canary deployments; and third-party service connectors for databases, message queues, and monitoring tools. The third thrust addresses *comprehensive observability*: distributed tracing across object invocations with low overhead, SLA compliance monitoring with real-time violation detection and root cause analysis, cost attribution linking resource consumption to business value, performance profiling identifying bottlenecks across the edge-cloud continuum, and actionable insights through intelligent dashboards and anomaly detection.

This milestone directly addresses empirically validated pain points: onboarding challenges (35.6% of interviews), time-to-productivity concerns, and observability gaps (11.9% of interviews). By providing polished SDKs and local development environments, OaaS reduces the steep learning curve identified in customer discovery. Comprehensive observability enables practitioners to understand system behavior, diagnose failures, and optimize resource usage—capabilities critical for production adoption but absent in current serverless platforms.

A particularly compelling application domain enabled by this ecosystem is *Digital Twin systems*, where OaaS's unified object abstraction naturally models physical assets (sensors, actuators, machinery) as distributed objects with state synchronization across the edge-cloud continuum. OaaS's declarative SLAs provide consistency and latency guarantees essential for real-time physical-virtual synchronization, while edge-cloud placement ensures low-latency processing at the network edge. Research challenges include designing intuitive APIs for physical-virtual state mapping, ensuring synchronization under intermittent connectivity, and providing domain-specific observability for manufacturing, smart cities, and infrastructure monitoring—domains where OaaS's strengths in state management and edge deployment directly address critical requirements.

7.2.3. M3: Agentic AI for Full-Lifecycle Application Management

A transformative research frontier explores autonomous AI agents capable of managing complete application lifecycles, representing a paradigm shift from developer-driven to AI-assisted cloud-native development. Research challenges span: (i) *Intent-to-implementation synthesis* using large language models to translate natural language requirements or high-level specifications into correct, optimized OaaS object definitions with appropriate SLAs and architectural patterns; (ii) *Continuous optimization* through reinforcement learning agents that autonomously tune SLAs, resource allocations, and placement strategies by analyzing telemetry, predicting workload patterns, and adapting to cost-performance trade-offs; and (iii) *Intelligent debugging and remediation* leveraging program analysis and causal inference to diagnose distributed failures, identify root causes across the edge-cloud continuum, and synthesize fixes through automated code patches or configuration adjustments.

This milestone addresses the ultimate developer experience challenge: enabling practitioners to express *what* they want without specifying *how* to achieve it, directly targeting the productivity and automation expectations prioritized by 53.5% and 44.6% of customer discovery participants respectively. Critical research challenges include ensuring correctness and preventing hallucination-induced errors through formal verification techniques, establishing verifiable trust boundaries between developer intent and autonomous execution, and creating interpretable decision traces that maintain developer understanding and control. Successfully addressing these challenges positions OaaS as a substrate for next-generation cloud platforms where AI agents handle infrastructure complexity while developers focus purely on business logic and application requirements.

7.2.4. M4: Security Hardening

The outermost milestone addresses comprehensive security across all layers, transforming OaaS into an enterprise-grade platform suitable for regulated industries. Research directions include: (i) *Defense-in-depth architecture* implementing multi-tenant isolation through secure sandboxing, enforcing principle of least privilege via fine-grained access controls, and deploying zero-trust networking where every object invocation requires authentication and authorization; (ii) *Cryptographic enforcement* ensuring data locality constraints through verifiable placement proofs, mandatory encryption at rest and in transit, and immutable audit trails for compliance verification; (iii) *Compliance automation* translating regulatory requirements (PCI-DSS for payment processing, HIPAA for healthcare data, GDPR for privacy, SOC 2 for service operations) into declarative policy specifications that the platform continuously enforces and validates; (iv) *Vulnerability management* integrating automated dependency scanning to detect CVEs, orchestrating security patches across distributed deployments, and implementing runtime exploit mitigation techniques; and (v) *Security certification pathways* establishing clear frameworks and evidence collection mechanisms to streamline third-party audits and regulatory approvals.

This milestone directly addresses acute security concerns identified during customer discovery, where financial services and healthcare participants emphasized that security certification and compliance guarantees were prerequisites for evaluation. Research challenges span designing security abstractions that remain intuitive despite cryptographic complexity, balancing performance overhead from encryption and access control against security guarantees, and creating compliance frameworks that adapt to evolving regulatory landscapes while maintaining verifiable correctness. Successfully addressing these challenges will enable OaaS adoption in high-assurance environments where current serverless platforms face significant trust and compliance barriers.

Collectively, these nested research directions chart a path from production-ready platform (M1) through enhanced developer experience and observability (M2) to intelligent automation (M3) and comprehensive security (M4), addressing empirically validated pain points while progressively expanding OaaS capabilities. By systematically addressing these challenges, future work can advance cloud platforms that are simultaneously powerful, accessible, secure, and adaptive to the evolving needs of diverse applications across the computing continuum.

BIBLIOGRAPHY

[1] Tahir Abbas, Ali Haider Khan, Khadija Kanwal, Ali Daud, Muhammad Irfan, Amal Bukhari, and Riad Alharbey, *Iomt-based healthcare systems: A review.*, Computer Systems Science & Engineering 48 (2024), no. 4.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa, *Firecracker: Lightweight virtualization for serverless applications*, Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 419–434.

[3] Amazon, *AWS Lambda | Serverless Compute - Amazon Web Services*, `https://aws.amazon.com/lambda/`, Online; Accessed on 12 Nov. 2023.

[4] ———, *AWS Step Functions | Serverless Microservice Orchestration*, `https://aws.amazon.com/step-functions`, Online; Accessed on 23 Jul. 2025.

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

[6] ———, *AWS Lambda Service Level Agreement*, `https://aws.amazon.com/lambda/sla`, May 2022, Online; Accessed on 15 July 2024.

[7] ———, *Building a Modular and Scalable Virtual Network Architecture with Amazon VPC*, `https://docs.aws.amazon.com/whitepapers/latest/building-scalable-secure-multi-vpc-network-infrastructure/welcome.html`, 2023, Online; Accessed on 15 Jul. 2024.

[8] ———, *Build a Serverless Web Application to Search for Images Using Amazon Bedrock*, `https://aws.amazon.com/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/`, 2024, Online; Accessed on 15 July 2024.

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

[10] ———, *Dynamic Video Content Moderation and Policy Evaluation Using AWS Generative AI Services*, `https://aws.amazon.com/solutions/implementations/content-moderation/`, 2024, Online; Accessed on 22 Dec. 2024.

[11] Apache, *Apache Flink Stateful Functions*, `https://nightlies.apache.org/flink/flink-statefun-docs-stable`, Online; Accessed on 12 Nov. 2023.

[12] Onur Ascigil, Argyrios G Tasiopoulos, Truong Khoa Phan, Vasilis Sourlas, Ioannis Psaras, and George Pavlou, *Resource provisioning and allocation in function-as-a-service edge-clouds*, IEEE Transactions on Services Computing 15 (2021), no. 4, 2410–2424.

[13] Mohammad Sadegh Aslanpour, Adel N Toosi, Muhammad Aamir Cheema, Mohan Baruwal Chhetri, and Mohsen Amini Salehi, *Load balancing for heterogeneous serverless edge computing: A performance-driven and empirical approach*, Future generation computer systems 154 (2024), 266–280.

[14] Alex Auvolat and François Taïani, *Merkle search trees: Efficient state-based crdts in open networks*, Proceedings of the 38th Symposium on Reliable Distributed Systems (SRDS), IEEE, 2019, pp. 221–22109.

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

[16] S. Bangera, *Devops for serverless applications: Design, deploy, and monitor your serverless applications using devops practices*, Packt Publishing, 2018.

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

[18] Bruno Guazzelli Batista, Carlos Henrique Gomes Ferreira, Danilo Costa Marim Segura, Dionisio Machado Leite Filho, and Maycon Leone Maciel Peixoto, *A qos-driven*

approach for cloud computing addressing attributes of performance and security, Future Generation Computer Systems 68 (2017), 260–274.

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

[20] Steve Blank, *The four steps to the epiphany: Successful strategies for products that win*, 5th ed., John Wiley & Sons, Hoboken, NJ, 2020.

[21] Praveen Borra, *Comparison and analysis of leading cloud service providers (aws, azure and gcp)*, International Journal of Advanced Research in Engineering and Technology (IJARET) Volume 15 (2024), 266–278.

[22] Eric Brewer, *Cap twelve years later: How the "rules" have changed*, Computer 45 (2012), no. 2, 23–29.

[23] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu, *Netherite: Efficient execution of serverless workflows*, Proceedings of the VLDB Endowment 15 (2022), no. 8, 1591–1604.

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

[25] Gonçalo Carvalho, Bruno Cabral, Vasco Pereira, and Jorge Bernardino, *Edge computing: current trends, research challenges and future directions*, Computing 103 (2021), no. 5, 993–1023.

[26] Inês Carvalho, Filipe Sá, and Jorge Bernardino, *Nosql document databases assessment: Couchbase, couchdb, and mongodb*, Proceedings of the 11th International Conference on Data Science, Technology and Applications, INSTICC, SciTePress, 2022, pp. 557–564.

[27] Dheeraj Chahal, Ravi Ojha, Manju Ramesh, and Rekha Singhal, *Migrating large deep*

learning models to serverless architecture, Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2020, pp. 111–116.

[28] Chaos Mesh Authors, *Chaos mesh: A powerful chaos engineering platform for kubernetes*, `https://chaos-mesh.org/`, 2025, Online; Accessed on 1 Jan. 2025.

[29] Google Cloud, *Cloud Functions | Google Cloud*, `https://cloud.google.com/functions/`, Online; Accessed on 12 Nov. 2023.

[30] Cloud Native Foundation, *Kubernetes*, `https://kubernetes.io/`, Online; Accessed on 12 Nov. 2023.

[31] Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman B"ohringer, Rodrigo Bruno, and Torsten Hoefler, *Process-as-a-service: Unifying elastic and stateful clouds with serverless processes*, Proceedings of the 15th ACM Symposium on Cloud Computing, 2024, pp. 223–242.

[32] Gatling Corp, *Gatling - Professional Load Testing Tool*, `https://gatling.io/`, Online; Accessed on 12 Nov. 2023.

[33] Angelo Corsaro, Gabriele Baldoni, and Luca Corfi, *Zenoh: The zero overhead network protocol*, Proceedings of the 10th ACM/IEEE Conference on Internet of Things Design and Implementation, ACM, 2023, pp. 1–13.

[34] Breno Costa, Joao Bachiega Jr, Leonardo Rebouças de Carvalho, and Aleteia PF Araujo, *Orchestration in fog computing: A comprehensive survey*, ACM Computing Surveys (CSUR) 55 (2022), no. 2, 1–34.

[35] Marco Dalba, Michele Scazzariello, and Elisabetta Di Nitto, *Reference architecture for integrating edge and cloud in iot systems*, IEEE Access 11 (2023), 98765–98783.

[36] Deep Manish Kumar Dave and Bharath Kumar Mittapally, *Data integration and interoperability in iot: challenges, strategies and future direction*, Int. J. Comput. Eng. Technol. (IJCET) 15 (2024), 45–60.

[37] Elias Dritsas and Maria Trigka, *A survey on the applications of cloud computing in the industrial internet of things*, Big data and cognitive computing 9 (2025), no. 2, 44.

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

[39] Ana Ebrahimi, Mostafa Ghobaei-Arani, and Hadi Saboohi, *Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions*, Journal of Systems Architecture (2024), 103115.

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

[41] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup, *The state of serverless applications: Collection, characterization, and community consensus*, IEEE Transactions on Software Engineering 48 (2021), no. 10, 4152–4166.

[42] Alex Ellis, *OpenFaaS - Serverless Functions Made Simple*, `https://www.openfaas.com/`, Online; Accessed on 12 Nov. 2023.

[43] Gonçalo Esteves, Filipe Fidalgo, Nuno Cruz, and José Simão, *Long-range wide area network intrusion detection at the edge*, IoT 5 (2024), no. 4, 871–900.

[44] Zainab Fatima, Muhammad Hassan Tanveer, Waseemullah, Shehnila Zardari, Laviza Falak Naz, Hina Khadim, Noorah Ahmed, and Midha Tahir, *Production plant and warehouse automation with iot and industry 5.0*, Applied Sciences 12 (2022), no. 4, 2053.

[45] Apache Software Foundation, *Apache OpenWhisk is a serverless, open source cloud platform*, `https://openwhisk.apache.org/`, Online; Accessed on 12 Nov. 2023.

[46] Cloud Native Foundation, *Knative*, `https://knative.dev/`, Online; Accessed on 12 Nov. 2023.

[47] Alexander Fuerst and Prateek Sharma, *Faascache: keeping serverless computing alive*

with greedy-dual caching, Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 386–400.

[48] GitHub, *GitHub Actions*, `https://github.com/features/actions`, Online; Accessed on 21 Apr. 2025.

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

[50] Bindu Mohan Harve, Darshan Mohan Bidkar, Manjunatha Sughaturu Krishnappa, Gokul Pandy, Vivekananda Jayaram, Prema Kumar Veerapaneni, and Gaurav Mehta, *The cloud-native revolution: Microservices in a cloud-driven world*, 2024 International Conference on Intelligent Cybernetics Technology & Applications (ICICyTA), 2024, pp. 1043–1048.

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

[52] Hewlett Packard Enterprise, *Hpe multi-tier cloud computing architecture*, `https://www.hpe.com/us/en/solutions/cloud.html`, 2024, Online; Accessed on 14 Oct. 2024.

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

[54] Mohamed K Hussein, Mohamed H Mousa, and Mohamed A Alqarni, *A placement architecture for a container as a service (caas) in a cloud environment*, Journal of Cloud Computing 8 (2019), no. 1, 1–15.

[55] ArangoDB Inc., *Arangodb*, `https://www.arangodb.com`, Online; Accessed on 12 Nov. 2023.

[56] MinIO Inc., *MinIO | High Performance, Kubernetes Native Object Storage*, `https://min.io/`, Online; Accessed on 12 Nov. 2023.

[57] Red Hat Inc., *Infinispan*, `https://infinispan.org/`, 2023, Online; Accessed on 12 Nov. 2023.

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

[59] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li, *Astrea: Auto-serverless analytics towards cost-efficiency and qos-awareness*, IEEE Transactions on Parallel and Distributed Systems 33 (2022), no. 12, 3833–3849.

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

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

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

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

[64] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs, *Lessons learned*

from the chameleon testbed, Proceedings of the USENIX Annual Technical Conference, USENIX ATC '20, USENIX Association, July 2020.

[65] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan W Weckwerth, Brian S Xia, et al., *Apiary: A dbms-backed transactional function-as-a-service framework*, CoRR (2022).

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

[67] Edward A Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard, *Quantifying and generalizing the cap theorem*, arXiv preprint arXiv:2109.07771 (2021).

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

[69] Pawissanutt Lertpongrujikorn, Hai Duc Nguyen, and Mohsen Amini Salehi, *Streamlining cloud-native application development and deployment with robust encapsulation*, Proceedings of the ACM Symposium on Cloud Computing (SoCC '24), 2024, pp. 847–865.

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

[71] ———, *Object as a service: Simplifying cloud-native development through serverless object abstraction*, IEEE Transactions on Computers (2025).

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

[73] Wen-Yew Liang, Yuyuan Yuan, and Hsiang-Jui Lin, *A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds*, arXiv preprint arXiv:2303.09419 (2023).

[74] Changyuan Lin and Hamzeh Khazaei, *Modeling and optimization of performance and cost of serverless applications*, IEEE Transactions on Parallel and Distributed Systems 32 (2020), no. 3, 615–632.

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

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

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

[78] Microsoft, *Azure Functions Serverless Compute*, `https://azure.microsoft.com/en-us/services/functions/`, Online; Accessed on 12 Nov. 2023.

[79] ———, *Durable entities - Azure Functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities`, Online; Accessed on 12 Nov. 2023.

[80] ———, *Chatty i/o antipattern*, `https://learn.microsoft.com/en-us/azure/architecture/antipatterns/chatty-io/`, 2024, Online; Accessed on 4 July 2024.

[81] Microsoft Azure, *Real-time inventory management with azure iot*, `https://learn.microsoft.com/en-us/azure/architecture/solution-ideas/articles/real-time-inventory-management`, 2024, Online; Accessed on 14 October 2024.

[82] Joan Miquel Solé, Roger Pueyo Centelles, Felix Freitag, Roc Meseguer, and Roger Baig, *Middleware for distributed applications in a lora mesh network*, ACM Transactions on Embedded Computing Systems 24 (2025), no. 4, 1–26.

[83] Oluwole Temidayo Modupe, Aanuoluwapo Ayodeji Otitoola, Oluwatayo Jacob Oladapo, Oluwatosin Oluwatimileyin Abiona, Oyekunle Claudius Oyeniran, Adebunmi Okechukwu Adewusi, Abiola Moshood Komolafe, and Amaka Obijuru, *Re-*

viewing the transformational impact of edge computing on real-time data processing and analytics, Computer Science & IT Research Journal 5 (2024), no. 3, 603–702.

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

[85] Wilmer Moina-Rivera, Miguel Garcia-Pineda, Jose M Claver, and Juan Gutiérrez-Aguado, *Event-driven serverless pipelines for video coding and quality metrics*, Journal of Grid Computing 21 (2023), no. 2, 20.

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

[87] Hai Duc Nguyen and Andrew A Chien, *Storm-rts: Stream processing with stable performance for multi-cloud and cloud-edge*, Proceedings of the 16th International Conference on Cloud Computing (CLOUD), IEEE, 2023, pp. 45–57.

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

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

[90] Muhammad Noaman, Muhammad Sohail Khan, Muhammad Faisal Abrar, Sikandar Ali, Atif Alvi, and Muhammad Asif Saleem, *Challenges in integration of heterogeneous internet of things*, Scientific Programming 2022 (2022), no. 1, 8626882.

[91] Diego Ongaro and John Ousterhout, *In search of an understandable consensus algorithm*, Proceedings of the USENIX annual technical conference (USENIX ATC 14), 2014, pp. 305–319.

[92] OpenRaft Contributors, *Openraft: Advanced raft consensus algorithm implementation*,

https://github.com/datafuselabs/openraft, 2024, Online; Accessed on 14 Oct. 2024.

[93] Alexander Osterwalder and Yves Pigneur, *Business model generation: A handbook for visionaries, game changers, and challengers*, John Wiley & Sons, Hoboken, NJ, 2010.

[94] Richard Patsch and Karl Michael Göschka, *Make applications faas-ready: Challenges and guidelines*, Proceeding of the 6th International Conference on Information Technology and Computer Communications, 2024, pp. 57–64.

[95] Pivotal Software, *Rabbitmq: Open source message broker*, https://www.rabbitmq.com/, 2024, Online; Accessed on 14 Oct. 2024.

[96] Platform9 Systems Inc., *Fission: Fast and Simple Serverless Functions for Kubernetes*, https://fission.io, 2024, Online; Accessed on 1 Apr. 2024.

[97] Rancher Labs, *k3d: Little helper to run rancher lab's k3s in docker*, https://k3d.io/, 2024, Online; Accessed on 14 Oct. 2024.

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

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

[100] Yuya Sasaki, Tomoki Hayashi, Hiroaki Ueda, and Hiroyuki Ohsaki, *A survey on iot stream processing systems: Analysis and the open challenges*, IEEE Internet of Things Journal 8 (2021), no. 16, 12559–12577.

[101] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini, *Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider*, Proceedings of the USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association, July 2020, pp. 205–218.

[102] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski, *Conflict-free repli-*

cated data types, Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13, Springer, 2011, pp. 386–400.

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

[104] Miloš Simić, Milan Stojkov, Goran Sladić, and Branko Milosavljević, *Crdts as replication strategy in large-scale edge distributed system: An overview*, 2020.

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

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

[107] Jonas Spenger, Paris Carbone, and Philipp Haller, *A survey of actor-like programming models for serverless computing*, Active Object Languages: Current Research Trends, Springer, 2024, pp. 123–146.

[108] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov, *Cloudburst: Stateful functions-as-a-service*, Proceedings of the VLDB Endowment 13, no. 11.

[109] SUSE Inc., *RKE2*, `https://docs.rke2.io`, 2024, Online; Accessed on 15 July 2024.

[110] Mark Szalay, Peter Matray, and Laszlo Toka, *Real-time faas: Towards a latency bounded serverless cloud*, IEEE Transactions on Cloud Computing 11 (2022), no. 2, 1636–1650.

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

[112] Aastik Verma, Anurag Satpathy, Sajal K Das, and Sourav Kanti Addya, *Lease: Leveraging energy-awareness in serverless edge for latency-sensitive iot services*, Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events 2024 (PerCom Workshops), IEEE, 2024, pp. 302–307.

[113] VMware, Inc., *VMware*, `https://www.vmware.com/`, Online; Accessed on 23 Jul. 2022.

[114] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy, *Lass: Running latency sensitive serverless computations at the edge*, Proceedings of the 30th international symposium on high-performance parallel and distributed computing, 2021, pp. 239–251.

[115] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift, *Peeking behind the curtains of serverless platforms*, Proceedings of the USENIX Annual Technical Conference (USENIX ATC 18) (Boston, MA), USENIX Association, July 2018, pp. 133–146.

[116] Wei Wang, Jiajun Yang, and Richard Muntz, *Efficient query processing in geographic web search engines*, Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, 2018, pp. 277–288.

[117] Shangrui Wu, Chavit Denninnart, Xiangbo Li, Yang Wang, and Mohsen Amini Salehi, *Descriptive and predictive analysis of aggregating functions in serverless clouds: The case of video streaming*, 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, 2020, pp. 19–26.

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

[119] Guangba Yu, Pengfei Chen, Zibin Zheng, Jingrun Zhang, Xiaoyun Li, and Zilong He,

Faasdeliver: Cost-efficient and qos-aware function delivery in computing continuum, IEEE Transactions on Services Computing 16 (2023), no. 5, 3332–3347.

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

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

[122] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang, *In-memory big data management and processing: A survey*, IEEE Trans. on Knowledge and Data Eng. 27 (2015), no. 7, 1920–1948.

[123] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu, *Fault-tolerant and transactional stateful serverless workflows*, 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20, USENIX Association, Nov. 2020, pp. 1187–1204.

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

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

[126] Xuan Zhang, Hongjun Gu, Guopeng Li, Xin He, and Haisheng Tan, *Online function caching in serverless edge computing*, Proceedings of the 29th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2023, pp. 2295–2302.

[127] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou, *Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows*, Proceed-

ings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, 2022, pp. 1–14.