EdgeWeaver: Accelerating IoT Application Development Across Edge-Cloud Continuum

Pawissanutt Lertpongrujikorn[1], Juahn Kwon[1], Hai Duc Nguyen[2], and Mohsen Amini Salehi[1]
[1]High Performance Cloud Computing (HPCC) Lab, University of North Texas (UNT), USA
[2]Argonne National Laboratory and University of Chicago, USA
{pawissanutt.lertpongrujikorn, Juahn.Kwon, mohsen.aminisalehi}@unt.edu, hai.nguyen@anl.gov

***Abstract*—The rise of complex, latency-sensitive IoT applications across the Edge–Cloud continuum exposes the limitations of current Function-as-a-Service (FaaS) platforms in seamlessly addressing the complexity, heterogeneity, and intermittent connectivity of Edge-Cloud environments. Developers are left to manage integration and Quality of Service (QoS) enforcement manually, rendering application development complicated and costly. To overcome these limitations, we introduce the EdgeWeaver platform that offers a unified "object" abstraction that is seamlessly distributed across the continuum to encapsulate application logic, state, and QoS. EdgeWeaver automates "class" deployment across edge and cloud by composing established distributed algorithms (e.g., Raft, CRDTs)—enabling developers to declaratively express QoS (e.g., availability and consistency) desires that, in turn, guide internal resource allocation, function placement, and runtime adaptation to fulfill them. We implement a prototype of EdgeWeaver and evaluate it under diverse settings and using human subjects. Results show that EdgeWeaver boosts development productivity by 31%, while declaratively enforcing strong consistency and achieving 9 nines availability, 10,000× higher than the current standard, with negligible performance impact.**

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

I. Introduction

The emergence of the IoT-Edge-Cloud paradigm has transformed the IoT landscape from simple sensor-based systems to complex, data-driven applications that underpin modern smart services across a wide range of domains, such as intelligent transportation [33], industrial automation [42], healthcare [11], and smart cities [88]. However, this paradigm shift introduces new challenges, including ensuring desired performance, availability, and consistency across highly heterogeneous and intermittently connected computing tiers, thereby complicating application development and deployment.

To address these challenges, Serverless Computing, or Function-as-a-Service (FaaS), introduces the function abstraction to hide underlying infrastructure [38]. Yet, as shown in the upper part of Figure 1, the abstraction still required significant efforts. Developers need to integrate state externally (e.g., through databases) and coordinate communication using data/event fabrics such as MQTT and Kafka [41], while considering QoS factors for performance, availability, and consistency [65]. Furthermore, as existing FaaS platforms are designed for centralized cloud environments, they struggle to operate efficiently in decentralized, failure-prone edge infrastructures due to heterogeneity in the underlying infrastructure [68].

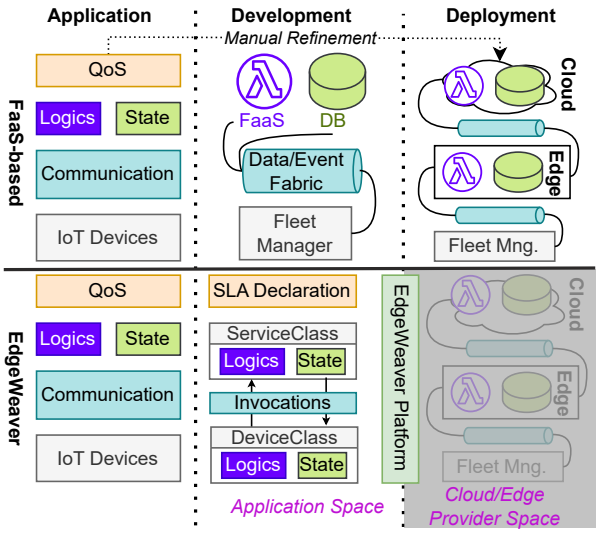


Fig. 1: EdgeWeaver vs. FaaS approach to develop and deploy applications across Edge-Cloud continuum.

CAP theorem [19] proves the infeasibility of simultaneously guaranteeing multiple QoS metrics (i.e., consistency, availability, and partition tolerance) in such distributed systems. As a result, developers have to make difficult trade-offs via prioritizing certain QoS desires over others (e.g., consistency over availability) based on the application goals and execution conditions [25], which further complicates the development of scalable, resilient IoT systems [70], [43]. Prior efforts have tackled parts of this problem through proposing high-level abstractions [49], [51], SLA-driven scheduling [59], [61], and middleware for device fleet management [4]. However, these solutions typically focus on isolated aspects, such as optimizing a single performance metric or supporting specific deployment configurations, without offering a unified, flexible framework for the full application life-cycle. This fragmentation forces developers back into the role of system integrators—manually stitching together disparate tools, thereby increasing development complexity and cost.

To overcome these challenges, we propose *EdgeWeaver*, a novel object-based abstraction to unify and streamline the development and deployment of Edge-Cloud IoT applications.

Inspired by the object-oriented programming (OOP) semantics, as shown in the lower part of Figure 1, EdgeWeaver establishes the notion of *distributed objects* to encapsulate application logic, state, IoT device interactions, and communication patterns within cohesive class-based abstractions. These *classes* are instantiated and managed as distributed *objects* that span across edge and cloud tiers. We also introduce a declarative Service Level Agreement (SLA) interface that allows developers to specify QoS requirements, including consistency, availability, and performance, for objects. EdgeWeaver automatically interprets these SLAs to guide resource allocation, function placement, and runtime adaptation, robustly fulfilling QoS requirements with minimal user intervention.

EdgeWeaver realizes object abstraction through its core component, called *distributed class runtimes*, deployed across Edge-Cloud to hide the underlying infrastructure (e.g., compute, storage, and network) complexity. This component enables intuitive, location-transparent function calls within or across distributed objects. For SLA enforcement, EdgeWeaver provides a unified architecture that automatically composes and configures established distributed mechanisms: (i) Raft consensus and CRDTs [77] for consistency; (ii) adaptive replication strategies for high availability; and (iii) rate-guarantee abstractions [61], [60], [12] for performance. The novelty lies in the framework's ability to automatically select and configure these mechanisms based on high-level SLA specifications. Evaluations demonstrate that EdgeWeaver, as a cohesive platform, can: significantly boost the developer's productivity; maintain reliable QoS under dynamic conditions; and match the efficiency of state-of-the-art systems. The contributions of this paper are as follows:

- Comprehensive Object Abstraction: We introduce a unified distributed object-based abstraction across Edge-Cloud that streamlines the development and deployment of IoT applications. Our evaluation shows this abstraction reduces implementation by up to 44.5% and configuration code by up to 10× fewer and, through a human study, demonstrates 31% development time reduction compared to conventional FaaS approaches (§IV).
- Automated Algorithm Composition via Declarative SLAs: We developed a declarative SLA mechanism that enables automated, self-adaptive QoS enforcement. This allows applications to achieve strong consistency and up to 9 nines availability—10,000× more reliable than the current standard at only 3× the cost (§ V).
- Systematic evaluation: Real-world experiments show that EdgeWeaver scales throughput by up to 70× while automatically adapting to dynamic environments and maintaining SLA compliance with minimal manual effort. (§ VI).

II. Problem Statement

As illustrated in Figure 2a, modern IoT applications often rely on devices that continuously collect data from their surroundings. For instance, traffic detection systems constantly gather video from street cameras [76]. These data streams are then processed by IoT services, which may be latency-sensitive (e.g., speed violation detection [30]) or latency-tolerant (e.g., analyzing traffic patterns to optimize signal timing [56]). Modern FaaS deployments adopt the Edge–Cloud paradigm, where resource-constrained edge servers near data sources handle latency-critical functions, while the others are offloaded to the cloud (Figure 2d) [70], [73]. Although this hybrid architecture enhances responsiveness and resource efficiency, it also introduces significant design and operational challenges.

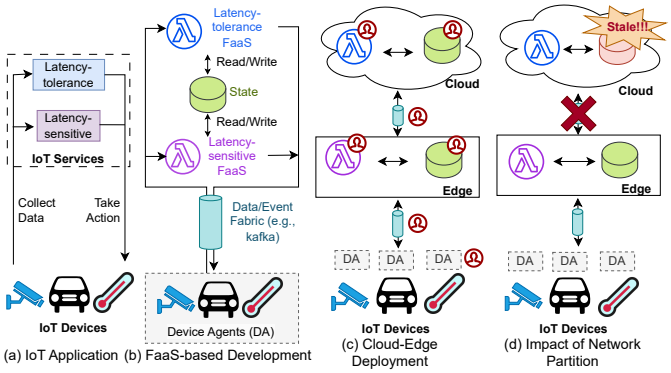


Fig. 2: FaaS-based development and deployment challenges

A. Complexity of Application Development

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 [49]. Developers must navigate multiple interfaces, complicating application design. Furthermore, the stateless nature of FaaS *lacks built-in support for long-lived functions*, which are crucial for IoT scenarios involving frequent or continuous data processing. Consequently, developers must manually coordinate intricate dataflows across many short-lived functions and shared-state services, making robust, end-to-end architectures difficult to maintain [64], [49].

In parallel, *heterogeneity* introduces additional challenges in managing and deploying IoT applications across the Edge–Cloud continuum. First, due to resource limitations, FaaS abstractions on the edge (e.g., using k3s) differ from their cloud counterparts (e.g., using k8s), increasing deployment complexity [34]. Second, large-scale IoT solutions span diverse technologies, application models, and communication protocols [62], [32]. Although IoT middlewares provide abstraction layers for easier administrations [58], these layers add management overhead and require service providers to handle fleet-wide configurations, deployments, and updates.

As a result, expanding IoT solutions across Edge-Cloud requires a significant development effort, especially at scale. As shown in Figure 2c, even with a single edge, a developer must implement and manage numerous components (marked with red head icons). With additional edge sites, this complexity escalates, as each node may differ in software stacks, capacity, network conditions, and policies. Coordinating such heterogeneous environments remains a major challenge [22], [14], *highlighting the need for new abstractions to improve*

integration, scalability, and variability in FaaS-based IoT systems.

B. No One-size-fit-all Solutions

The Edge-Cloud continuum often suffers from *intermittent connectivity* between tiers, caused by network congestion, physical obstructions, power limitations, or fluctuating bandwidth [35]. As depicted in Figure 2d, lost or delayed connections (i.e., *network partition*) can disrupt data flow, causing communication delays or losses. This undermines consistency, as the cloud's view of the application's state becomes stale while the edge node continues processing data locally, leading to serious consequences: unsynchronized data on a production line may result in incorrect machine behavior, and outdated patient information could lead to delayed or even dangerous clinical decisions [9]. Waiting for cloud confirmation ensures consistency but reduce availability, potentially delaying urgent actions like dispatching an ambulance after an accident [61].

In fact, the *CAP theorem* [19], [48] formally shows that simultaneously guaranteeing *consistency* and *availability* under network partitions is impossible. Consequently, Edge–Cloud IoT deployments must carefully adapting their design to balance trade-offs among consistency, availability, and performance for their application's specific QoS needs. This challenge is compounded by the heterogeneity of IoT environments, where devices, infrastructure, and communication protocols differ widely. Ensuring both timely responses and accurate data across such diverse and intermittently connected systems remains a core difficulty in Edge–Cloud IoT design.

III. Related Studies

A. Managing Edge-Cloud Complexity

Streamlining Edge-Cloud deployments. Contemporary Edge–Cloud solutions let applications express QoS (e.g., latency, availability, and cost) as SLAs to guide automatic FaaS placement and scheduling, shielding developers from low-level system heterogeneity [40], [59]. The approach is supported by extensive research, often by modeling it as an optimization problem to minimize cost [66], [72], [87], [17], [53] or energy consumption [13], [85] under SLA and resource constraints. Beyond QoS-based scheduling, fleet management frameworks address the operational complexity of large, heterogeneous IoT deployments. Middleware platforms such as Eclipse IoT projects [3] and EdgeX Foundry [4] provide unified interfaces, protocol translation [28], [18], and centralized registries (e.g., AWS IoT Device Management, Azure IoT Hub) for scalable onboarding, security, and monitoring. Automated tooling [74], [10] ensures consistent configuration and deployment, while policy-driven orchestration [47] applies tiered control to maintain reliability and scalability across diverse device fleets.

Unified compute-data abstraction. Many serverless platforms integrate functions and state into cohesive units, such as actors [78] or proclets [69], to streamline stateful function deployment. Azure Entity Functions [57], inspired by the virtual actor paradigm (e.g., Orleans [21]), exemplify this design by using conflict-free replicated data types (CRDTs) [75] for consistent state replication. Similarly, OaaS [50], [51], [49] and Nubes [54] apply object-oriented principles to encapsulate functions, state, and non-functional requirements into unified, cloud-native deployments.

B. Handling Intermittent Connectivity

Due to the CAP theorem's constraints, handling intermittent connectivity across the Edge-Cloud continuum inevitably requires a trade-off between availability and consistency. As a result, developers must make application-specific decisions based on architectural needs and QoS priorities.

Providing high availability. For applications prioritizing availability, weaker consistency models like causal or eventual consistency are often adopted [16]. These models use CRDTs [83], versioning, gossip protocols [23], and session guarantees [84] to handle data divergence and maintain service during network outages. Several studies let applications specify data requires strong consistency vs those can tolerate staleness. The Edge–Cloud orchestration then adjusts function placement and replication accordingly [6], [2].

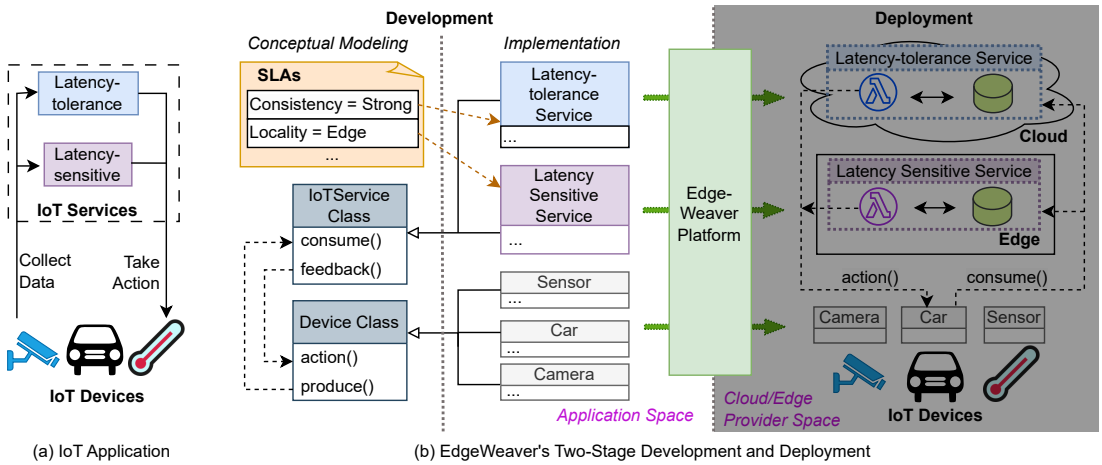
Enforcing strong consistency. When strong consistency is crucial, developers often replicate deployments across multiple Edge-Cloud regions and use consensus protocols like Paxos [46] or Raft [63] to ensure updates are agreed upon by a majority of replicas before being committed. While ensuring correctness, doing so reduces availability during network failures, as writes are paused until connectivity and synchronization are restored. To mitigate availability concerns, Google Cloud Spanner [5] leverages globally synchronized clocks with bounded error to offer strong consistency and high availability. CockroachDB [80] uses multi-version concurrency control and consensus to maintain consistency, deployment flexibility, and ease of migration.

C. Limitations

Lack of a comprehensive solution. Most efforts to simplify application deployment address *only isolated aspects* of the challenge rather than the *entire application lifecycle*. For example, OaaS [49] focuses primarily on cloud environments, where SLA enforcement often targets a single metric (e.g., throughput [61]) while overlooking other key factors such as availability and consistency. As a result, developers still expend significant effort integrating these *fragmented solutions* into cohesive and robust Edge–Cloud IoT systems [86], [67].

Lack of Unified SLA Support. Existing frameworks often treat QoS on a fragmented basis. They either offer very limited SLA support (e.g., Ekko [39] provides no SLA/QoS guarantees) or implement fundamentally narrow SLA models (e.g., TEMPOS [71] focuses exclusively on real-time execution guarantees). Consequently, there remains a gap for a unified SLA framework that concurrently spans performance, availability, and consistency.

Inflexibility. Current solutions are tightly bound to *specific* application requirements and Edge-Cloud configurations, limiting the ability to reconfigure, expand, or migrate Edge-Cloud



(a) IoT Application (b) EdgeWeaver's Two-Stage Development and Deployment

Fig. 3: Overview of the EdgeWeaver paradigm to resolve Edge-Cloud challenges for IoT applications

IoT services. Many commercial offerings are confined to a single public cloud or edge provider, making it difficult to construct hybrid deployments that span private infrastructures or multiple vendors [36]. Additionally, due to the lack of a *unifying framework*, even minor updates, such as changing a service configuration or upgrading an SLA, can result in *unpredictable ripple effects*, requiring extensive manual effort to ensure system correctness. This inherent rigidity hampers robust maintenance and scaling, thereby significantly increasing the cost of evolving Edge-Cloud IoT solutions.

IV. EdgeWeaver: 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 the EdgeWeaver platform. EdgeWeaver 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.

A. Comprehensive Object Abstraction

Inspired by unified compute-data abstractions in the cloud [49], [20], [26], EdgeWeaver adopts object-oriented programming (OOP) principles to provide a comprehensive abstraction for IoT applications through the notion of "object" abstraction. In this abstraction, applications consist of *distributed objects*, each of which contains *attributes* representing IoT data or state. *Functions (or methods)* encapsulate application logic, implemented as serverless functions. *Communication* occurs through method invocations between objects, abstracting control and data flow across edge and cloud. *QoS constraints* (e.g., performance, availability, consistency) are declared as SLAs and enforced transparently by the platform. The abstraction also includes deployment constraints (e.g., locality) to relieve the burden of low-level configuration (e.g., choosing where to deploy an object) from developers (see Table I).

This unified abstraction provides a comprehensive and intuitive view of the heterogeneous IoT devices and services while hiding implementation details, relieving developers from low-level system management. By establishing key OOP application-building features such as abstract classes, inheritance, and polymorphism across the edge–cloud continuum, EdgeWeaver promotes software reuse and modularity. Developers can define core functionalities as base classes and extend them to meet specific use cases, enhancing flexibility and maintainability. Furthermore, EdgeWeaver leverages OOP access modifiers and object composition features to organize application structure and encapsulate application flow through function calls. Crucially, EdgeWeaver abstracts away infrastructure dependencies, requiring no specific configurations or underlying technologies. As a result, applications remain fully decoupled from deployment environments and can be effortlessly deployed across diverse setups.

B. Declarative SLA-Driven Deployment

EdgeWeaver abstracts deployment complexity through SLA specifications, allowing developers to define non-functional requirements, such as performance, availability, and consistency, at the class level or for individual attributes and methods. These SLAs guide the platform's automated deployment process, ensuring that applications meet their QoS targets without manual intervention. By structuring application logic and state as objects, EdgeWeaver has a unified view to efficiently allocate resources, proactively distributing functions and associated data together to minimize latency and data transfer overhead. This SLA-driven resource management not only improves runtime efficiency but also adapts to dynamic conditions and infrastructure heterogeneity, enabling seamless and reliable execution across Edge-Cloud continuum.

C. Application Development and Deployment

As illustrated in Figure 3, the EdgeWeaver 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 EdgeWeaver '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

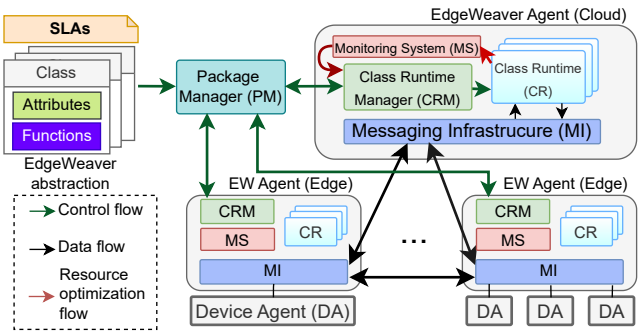


Fig. 4: EdgeWeaver Architecture

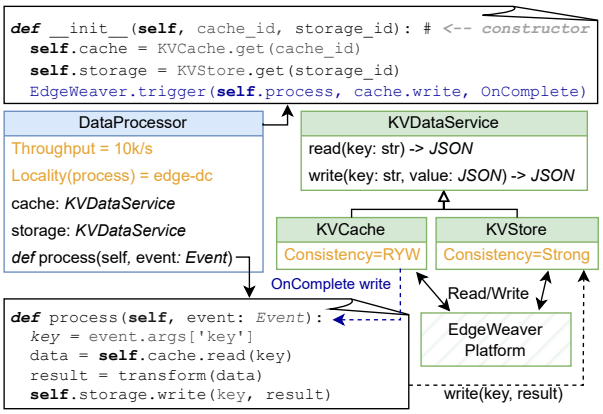


Fig. 5: Modeling and implementing a simple IoT processing service with EdgeWeaver

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 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 3b, 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, EdgeWeaver 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.

V. EdgeWeaver Realization

A. Design Goals and Architecture

We design a EdgeWeaver platform, following the architecture shown in Figure 4, to 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 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 4) 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.

B. Object Abstraction Realization

At its core, an EdgeWeaver 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 de-

SLA	Value Type	Unit	Definition	Algorithmic Mechanism
Consistency	Strong	N/A	Read always reflects the latest write.	Raft Consensus: Leader-based replication ensures linearizability.
	Bounded Staleness (Δ)	sec	Read can lag behind the latest write, but only within Δ seconds.	Merkle Trees + CRDTs: Efficient state synchronization and conflict resolution within time bounds.
	Read your Write (RYW)	N/A	Ensure a client's next read includes its most recent write.	Session Guarantees: Client-centric routing to local replicas.
Availability	Real	%	The percentage of time an object/function must be available for service.	Adaptive Replication: Replica count calculated via probabilistic failure models [55].
Throughput	Integer	RPS	Minimum number of invocations guaranteed to be executed per second.	Resource Pre-warming: Predictive autoscaling based on real-time Serverless [61].
Locality	Preferred datacenters	N/A	Preferred data centers that will be used for deployment.	SLA-driven Placement: Constraint solving to place data/compute near sources.

TABLE I: SLAs Supported by EdgeWeaver and their Corresponding Distributed Mechanisms

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 II: EdgeWeaver's API

ployment, 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 I). EdgeWeaver also provides a high-level API (Table II) for inter-object communication, event-driven triggers, and system interactions, all without handling low-level details like networking protocols or deployment scripts.

Figure 5 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 [63]). 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 per method (e.g., `Locality(process)=edge-dc`). All are managed entirely by EdgeWeaver; without any custom orchestration, messaging setup, or deployment scripting.

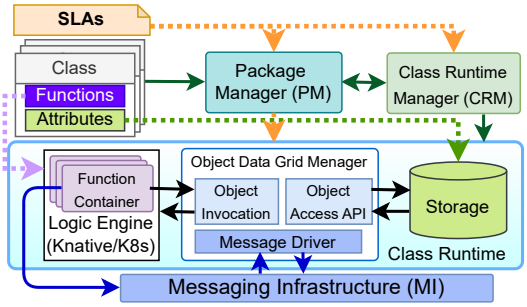


Fig. 6: Class Deployment with Class Runtime

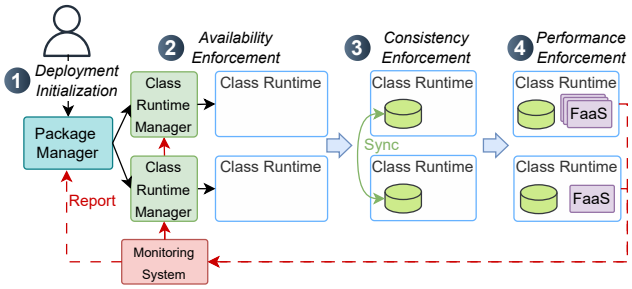


Fig. 7: Enforcing SLAs through class deployment

C. Class Deployment

Upon development completion, developers submit their applications to EdgeWeaver as a collection of class definitions annotated with SLAs. Figure 6 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 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 [52]).

D. SLA Enforcement

Once a class is successfully submitted and deployed, developers can begin instantiating objects from it. These objects represent running instances of the application logic, and their deployment must comply with the SLAs specified in the original class definition. As summarized in Table I, EdgeWeaver enforces these diverse SLA targets by automatically composing and configuring appropriate distributed algorithms. Figure 7 shows how EdgeWeaver enforces SLAs during deployment and execution, which we detail in the following subsections. These three SLA dimensions operate synergistically: the Package Manager first determines the number and placement of replicas to ensure *availability*; the ODGM then binds these distributed replicas through consensus or synchronization protocols to establish *consistency*; finally, the Class Runtime Manager (CRM) pre-allocates and co-locates compute resources around these storage replicas to guarantee *performance*.

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 the Meroufel and Belalem method [55]. 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.

2) Consistency Enforcement: When Class Runtimes are deployed, they provision storage and coordinate with each other to enforce the consistency SLA.

Strong Consistency. For applications requiring linearizability, EdgeWeaver employs the Raft consensus algorithm [63]. Raft operates by electing a leader among replicas to manage the replication log. All write requests are routed to the leader, which replicates the entry to followers. A write is committed only after a majority acknowledge it, ensuring the system can survive minority failures without data loss. EdgeWeaver integrates Raft directly into the ODGM's *Object Access API*, ensuring that all replicas agree on the latest object state before processing reads or writes.

Bounded-Staleness. For applications tolerating temporary lag, EdgeWeaver allows stale reads within a time-bound Δ. To manage this, ODGM employs Merkle Search Trees (MST) [15], a data structure that hashes the storage content hierarchy. Replicas periodically exchange root hashes; if they differ, they traverse the tree to identify divergent keys efficiently, minimizing bandwidth usage during synchronization. To resolve conflicts during these merges, EdgeWeaver uses Conflict-free Replicated Data Types (CRDTs) [77], ensuring that state converges mathematically without manual intervention. Read/write access is blocked only if network partitions exceed the allowed staleness window Δ.

Read-Your-Write (RYW). This model allows reads to see a recent write from the same client, even under network partition. Class Runtime enforces this by routing reads and writes through the object access API to the same local storage replica, effectively providing "session guarantees" without requiring global consensus.

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 [61], 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.

4) SLA-compliance Execution: After deployment, users interact with objects via the API provided in Table II. 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 §V-D2), to enforce SLA-compliant execution.

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.

E. Implementation

We implemented a EdgeWeaver prototype utilizing widely adopted open-source technologies:

Control Plane: The Package Manager and Class Runtime Manager are primarily implemented in Java. We also provide the SDK in Python for class implementation and definition.

Container Orchestrator: The platform is built on Kubernetes (for cloud) and K3s (for edge), leveraging Knative for serverless function orchestration.

Messaging Infrastructure: Inter-component communication uses Zenoh [27]. By design, Zenoh abstracts away physical placement and network routing across the Edge-Cloud continuum, enabling seamless cross-datacenter message passing.

Class Runtimes: Integrating computation and state, Class Runtimes leverage Knative to orchestrate serverless function containers (Logic Engine). The ODGM, implemented in Rust, acts as a core translation layer that subscribes to Zenoh topics and converts them into local HTTP calls for Knative, bridging the gap without modifying Knative's core. For object state (Storage), we integrate OpenRaft [31] for strong consistency, alongside custom Merkle Search Trees and CRDT wrappers for Bounded-Staleness and RYW. By routing all state synchronization payloads through Zenoh, these algorithms operate seamlessly across the edge-cloud without custom networking logic.

Source code: including the runtime, SDKs, and example applications, is available at https://github.com/hpcclab/OaaS-IoT.

VI. Performance Evaluation

A. Methodology

Goals. We evaluate EdgeWeaver across realistic settings to assess whether it fulfills its design objectives (§V-C) and thus, effectively addresses the challenges of IoT application development and deployment across the Edge–Cloud continuum (§II). 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? (§VI-B1) (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? (§VI-B2) (iii) *Adaptability for Reliable Execution:* Can EdgeWeaver dynamically respond to workload and infrastructure changes to preserve QoS with minimal developer effort? (§VI-B3)

Experimental Setup. We conduct experiments on Chameleon Cloud [45], 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 [79] while the edge cluster emulates resource-constrained edge environments with K3d [44], 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 with 8 vCPUs and 16 GB of memory. We strictly enforce these limits to accurately emulate edge constraints, utilizing the underlying hardware's full capacity only for scalability stress tests. One machine acts as an IoT gateway, generating synthetic data and issuing invocation requests to services deployed at both the edge and the cloud. To simulate real-world network disruptions, we use Chaos Mesh [24] to inject intermittent connectivity faults. We install EdgeWeaver (implementation details in §V-E) alongside other baselines across both cloud and edge clusters.

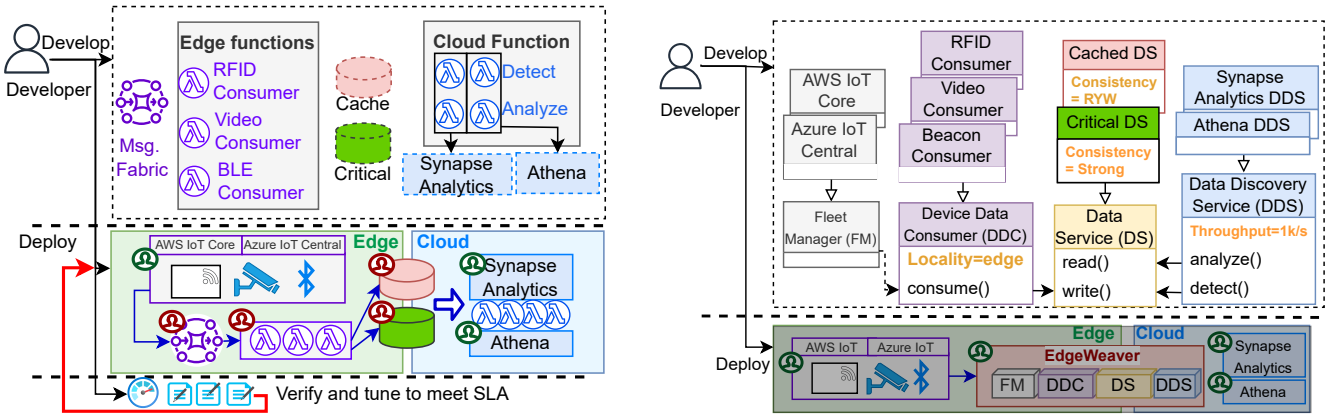
B. Experimental Results

1) Comprehensiveness and Productivity: We show how EdgeWeaver improves the application development and deployment productivity via case studies and human evaluations.

Comprehensiveness: Design Comparison. Figure 8 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 [7] and AWS IoT [29], 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 function placement, resource allocation, and network settings to meet SLAs.

In contrast, EdgeWeaver *unifies* and *automates* this entire process (Figure 8b). Developers work with high-level object abstractions instead of low-level components. Specialized han-



(a) FaaS-based solutions using AWS and Azure services [7]

(b) Using EdgeWeaver

Fig. 8: 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.

dlers (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.

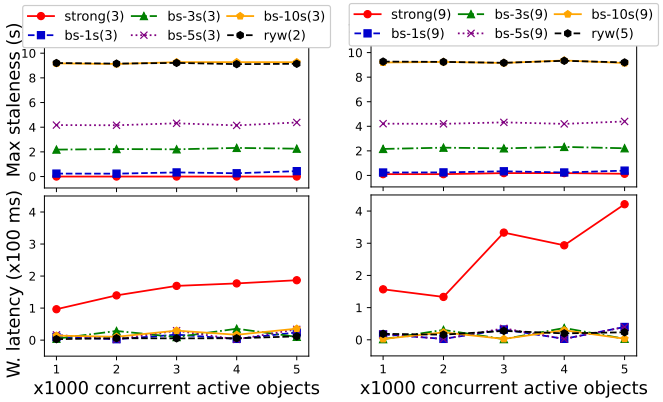
The results show a dramatic reduction in development overhead with EdgeWeaver. The Knative implementation required 666 LoC, while EdgeWeaver achieved equivalent functionality in only 363 LoC (44.5% reduction). The improvement in configuration effort was even more pronounced: EdgeWeaver needed just 39 LoCC, nearly 10× 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 Figure 8, the FaaS-based design forces developers to manage at least seven distinct components, whereas EdgeWeaver consolidates these into 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.

Developer Experience. To evaluate how EdgeWeaver's comprehensiveness and productivity translate into better developer experience, we conducted a human study with 39 college students. While this participant pool may not fully represent seasoned professional developers, it effectively measures the *learnability* and ease of adoption of new paradigms—a critical factor for platform adoption. Participants received a 15-minute tutorial on the FaaS and EdgeWeaver abstractions. The duration was short enough to avoid overwhelming participants, yet sufficiently comprehensive to provide essential understanding. The tutorial was followed by a quiz to assess conceptual comprehension and a programming task to evaluate practical performance. As shown in Table III (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 III, 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%). The programming task grading was based on a rubric weighing Conceptually Correctness (40%), Functionally Completeness (40%), and Code Quality (20%). These results demonstrate that EdgeWeaver 's unified abstractions and automation not only simplify development but also deliver a measurably better, faster, and more accessible programming experience than traditional FaaS-based approaches.

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



(a) Availability = 4 nines (b) Availability = 9 nines

Fig. 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.

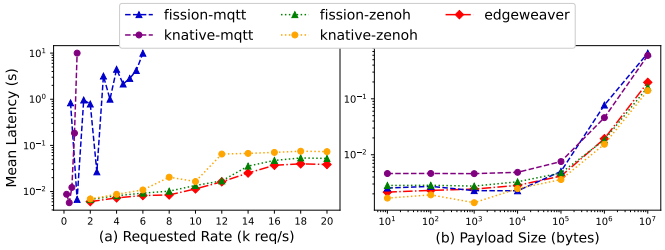


Fig. 10: Latency of function invocation with increasing request rate and payload size in various baseline systems.

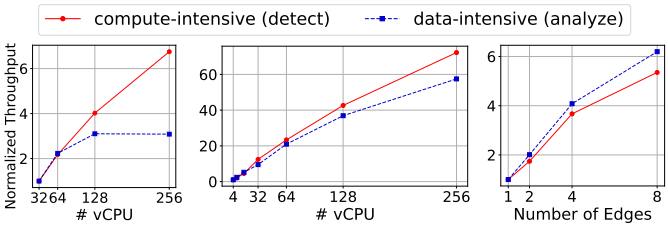
Takeaway: *EdgeWeaver provides a unified, comprehensive abstraction that streamlines development and deployment of IoT applications across the Edge–Cloud continuum.*

2) 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. EdgeWeaver's core technical contribution is the automated composition of established distributed algorithms (Raft, CRDTs, Merkle Trees) to enforce declarative SLAs. Here, we evaluate whether this automated composition reliably delivers the specified QoS guarantees across diverse configurations.

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. 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% [1]) and Tier-4 datacenters (99.995% [8]).

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



(a) Edge-Cloud (b) Cloud (c) Edge

Fig. 11: Scalability analysis of EdgeWeaver

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× higher reliability.

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 [37] and Fission [81] with MQTT [82] and Zenoh [27].

Figure 10 shows the latency of a lightweight echo function with no workload, representing the intrinsic system overhead of each approach. 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 model as a

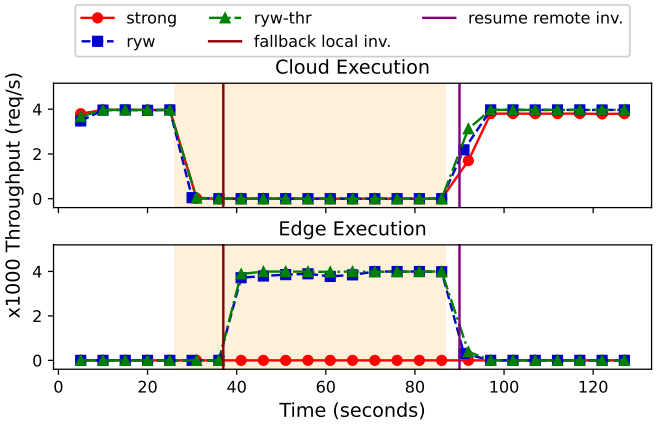


Fig. 12: Impact of network partitioning for classes with: RYW, RYW with throughput, and Strong Consistency.

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 8). 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 11a 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 11b presents the cloud-only results. The *detect* workload, being compute-intensive, scales nearly linearly—achieving a 70× throughput gain from 4 to 256 vCPUs, 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. 11c): 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.*

3) 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 [24] (yellow area in Fig. 12), 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.*

VII. Conclusion and Future Works

In this paper, we presented EdgeWeaver, a platform to streamline IoT application development and deployment across the Edge-Cloud continuum. Inspired by OOP principles, EdgeWeaver offers the object abstraction that encapsulates application state, functions (logics), and SLAs, thereby providing a holistic view across the continuum. Moreover, it can transparently handle user-defined consistency and availability trade-offs in the presence of network failure. Importantly, the benefits of EdgeWeaver do not come with any significant overhead to the system. In the future, we plan to extend EdgeWeaver to support multi-cloud development, allowing objects to run functions across different clouds.

Acknowledgement

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

References

[1] AWS Lambda Service Level Agreement. https://aws.amazon.com/lambda/sla/. Online; Accessed on 8 Oct. 2025.
[2] Consistency levels in Azure Cosmos DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels. Online; Accessed on 30 Sep. 2025.
[3] Eclipse Ditto. https://projects.eclipse.org/projects/iot.ditto. Online; Accessed on 31 Jan. 2025.
[4] EdgeX Foundry. https://www.edgexfoundry.org/. Online; Accessed on 30 Sep. 2025.
[5] Google Cloud Spanner. https://cloud.google.com/spanner. Online; Accessed on 1 Oct. 2025.
[6] Linux Foundataion Project EVE. https://lfedge.org/projects/eve/. Online; Accessed on 30 Sep. 2025.
[7] Tutorial: Deploy a smart inventory-management application template. https://learn.microsoft.com/en-us/azure/iot-central/retail/tutorial-iot-central-smart-inventory-management. Online; Accessed on 6 Sep. 2025.

[8] What are Data Center Tiers? https://www.hpe.com/us/en/what-is/data-center-tiers.html. Online; Accessed on 14 Sep. 2025.

[9] 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(4), 2024.

[10] Iván Alfonso, Kelly Garcés, Harold Castro, and Jordi Cabot. A model-based infrastructure for the specification and runtime execution of self-adaptive iot architectures. *Computing*, 105(9):1883–1906, 2023.

[11] Muntadher Alsabah, Marwah Abdulrazzaq Naser, AS Albahri, OS Albahri, AH Alamoodi, Sadiq H Abdulhussain, and Laith Alzubaidi. A comprehensive review on key technologies toward smart healthcare systems based iot: technical aspects, challenges and future directions. *Artificial Intelligence Review*, 58(11):1–122, 2025.

[12] R Andreoli, R Mini, P Skarin, H Gustafsson, J Harmatos, L Abeni, and T Cucinotta. A multi-domain survey on time-criticality in cloud computing. *IEEE Transactions on Services Computing*, 2025.

[13] Mohammad Sadegh Aslanpour, Adel N Toosi, Muhammad Aamir Cheema, and Mohan Baruwal Chhetri. faashouse: Sustainable serverless edge computing through energy-aware resource scheduling. *IEEE Transactions on Services Computing*, 2024.

[14] 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*, 154:266–280, 2024.

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

[16] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[17] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. Neptune: a comprehensive framework for managing serverless functions at the edge. *ACM Transactions on Autonomous and Adaptive Systems*, 19(1):1–32, 2024.

[18] Gunjan Beniwal and Anita Singhrova. A systematic literature review on iot gateways. *Journal of King Saud University-Computer and Information Sciences*, 34(10):9541–9563, 2022.

[19] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[20] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

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

[22] Gonçalo Carvalho, Bruno Cabral, Vasco Pereira, and Jorge Bernardino. Edge computing: current trends, research challenges and future directions. *Computing*, 103(5):993–1023, 2021.

[23] Roberto Casadei, Danilo Pianini, Mirko Viroli, and Antonio Natali. Self-organising coordination regions: A pattern for edge computing. In *Coordination Models and Languages: 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 21*, pages 182–199. Springer, 2019.

[24] Chaos Mesh Authors. Chaos mesh. https://chaos-mesh.org/, 2025. Online; Accessed on 8 Oct. 2025.

[25] Rajat Chaudhary, Gagangeet Singh Aujla, Neeraj Kumar, and Pushpinder Kaur Chouhan. A comprehensive survey on software-defined networking for smart communities. *International Journal of Communication Systems*, 38(1):e5296, 2025.

[26] Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman Böhringer, Rodrigo Bruno, and Torsten Hoefler. Process-as-a-service: Unifying elastic and stateful clouds with serverless processes. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 223–242, 2024.

[27] Angelo Corsaro, Luca Cominardi, Olivier Hecart, Gabriele Baldoni, Julien Enoch Pierre Avital, Julien Loudet, Carlos Guimares, Michael Ilyin, and Dmitrii Bannov. Zenoh: Unifying communication, storage and computation from the cloud to the microcontroller. In *Proceedings of the 26th Euromicro Conference on Digital System Design (DSD)*, pages 422–428. IEEE, September 2023.

[28] Mayur Dafare, Sandesh Waghmare, Abhijeet Bhoyar, Abhijit S Titarmare, and Pankaj Chandankhede. LoRa-Enabled Smart RS485 Data Logger and MQTT Gateway for Industrial IoT Applications Using ESP32. In *Proceedings of the International Conference on Circuit Power and Computing Technologies (ICCPCT)*, pages 1297–1302. IEEE, 2023.

[29] Jason Dalba, Navnit Shukla, Vetri Natarajan, and Sindhura Palakodety. Reference guide to build inventory management and forecasting solutions on AWS. Apr 2023. Online; Accessed on 6 June 2025.

[30] Yousef-Awwad Daraghmi, Mamoun Abu Helou, Eman-Yasser Daraghmi, and Waheeb Abu-Ulbeh. Iot-based system for improving vehicular safety by continuous traffic violation monitoring. *Future internet*, 14(11):319, 2022.

[31] databendlabs. openraft: rust raft with improvements. https://github.com/databendlabs/openraft. Online; Accessed on 8 Oct. 2025.

[32] 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)*, 15:45–60, 2024.

[33] Ganesh Gopal Devarajan, U Kumaran, Gopalakrishnan Chandran, Rajendra Prasad Mahapatra, and Ahmed Alkhayyat. Next generation imaging methodology: An intelligent transportation system for consumer industry. *IEEE Transactions on Consumer Electronics*, 2024.

[34] 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(2):44, 2025.

[35] Gonçalo Esteves, Filipe Fidalgo, Nuno Cruz, and José Simão. Long-range wide area network intrusion detection at the edge. *IoT*, 5(4):871–900, 2024.

[36] Wigananda Firdaus and Anjik Sukmaaji. Exploring opportunities and challenges in multi-cloud and hybrid cloud implementation. *Information Technology International Journal*, 2(2), 2024.

[37] Cloud Native Foundation. Knative. https://knative.dev/. Online; Accessed on 8 Oct. 2025.

[38] Muhammed Golec, Guneet Kaur Walia, Mohit Kumar, Felix Cuadrado, Sukhpal Singh Gill, and Steve Uhlig. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys*, 57(3):1–36, 2024.

[39] P. Gou et al. Ekko: Fully decentralized scheduling for serverless edge computing. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2023.

[40] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. Scheduling iot applications in edge and fog computing environments: A taxonomy and future directions. *ACM Computing Surveys*, 55(7):1–41, 2022.

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

[42] Jiong Jin, Zhibo Pang, Jonathan Kua, Quanyan Zhu, Karl H Johansson, Nikolaj Marchenko, and Dave Cavalcanti. Cloud-fog automation: The new paradigm towards autonomous industrial cyber-physical systems. *IEEE Journal on Selected Areas in Communications*, 2025.

[43] Nteziriza Nkerabahizi Josbert, Min Wei, Ping Wang, and Ahsan Rafiq. A look into smart factory for industrial iot driven by sdn technology: A comprehensive survey of taxonomy, architectures, issues and future research orientations. *Journal of King Saud University-Computer and Information Sciences*, 36(5):102069, 2024.

[44] K3D. Rancher Lab's minimal Kubernetes distribution. https://k3d.io/stable/. Online; Accessed on 8 Oct. 2025.

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

[46] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[47] Rodger Lea, Toni Adame, Alexandre Berne, and Selma Azaiez. The internet of things, fog, and cloud continuum: Integration challenges and opportunities for smart cities. *Future Internet*, 17(7):281, 2025.

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

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

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

[51] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. Object as a service: Simplifying cloud-native development through serverless object abstraction. *arXiv preprint arXiv:2408.04898*, 2024.

[52] 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.

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

[54] Kinga Anna Marek, Luca De Martini, and Alessandro Margara. Nubes: Object-oriented programming for stateful serverless functions. In *Proceedings of the 9th International Workshop on Serverless Computing*, pages 30–35, 2023.

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

[56] Florence Michael, Fadi Al-Turjman, Mubarak Auwal, and Chadi Altrjman. Traffic management system using different internet of things devices: literature review. *Artificial Intelligence of Things (AIoT)*, pages 47–53, 2025.

[57] Microsoft. Durable entities - Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities, 2025. Online; Accessed on 8 Oct. 2025.

[58] 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(4):1–26, 2025.

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

[60] Hai Duc Nguyen and Andrew A Chien. Efficient performance guarantees for function-as-a-service with cloud allocators. In *Proceedings of the 26th International Middleware Conference*, pages 99–113, 2025.

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

[62] 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(1):8626882, 2022.

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

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

[65] Shivananda Poojara, Pelle Jakovits, Rajkumar Buyya, and Satish Narayana Srirama. Scaling approaches for serverless data pipelines in edge and fog computing environments: A performance evaluation. *ACM Transactions on Autonomous and Adaptive Systems*, 2025.

[66] Thomas Pusztai and Stefan Nastic. Chunkfunc: Dynamic slo-aware configuration of serverless functions. *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[67] Amir Masoud Rahmani, Amir Haider, Parisa Khoshvaght, Farhad Soleimanian Gharehchopogh, Komeil Moghaddasi, Shakiba Rajabi, and Mehdi Hosseinzadeh. Optimizing task offloading with metaheuristic algorithms across cloud, fog, and edge computing networks: A comprehensive survey and state-of-the-art schemes. *Sustainable Computing: Informatics and Systems*, page 101080, 2025.

[68] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27(3):50–64, 2023.

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

[70] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. Serverless functions in the cloud-edge continuum: Challenges and opportunities. In *Proceedings of the 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 321–328. IEEE, 2023.

[71] Gabriele Russo Russo et al. Tempos: Qos management middleware for edge cloud computing faas in the internet of things. *IEEE Access*, 11, 2023.

[72] Gabriele Russo Russo, Daniele Ferrarelli, Diana Pasquali, Valeria Cardellini, and Francesco Lo Presti. Qos-aware offloading policies for serverless functions in the cloud-to-edge continuum. *Future Generation Computer Systems*, 156:1–15, 2024.

[73] Gabriele Russo Russo, Pierpaolo Spaziani, and Valeria Cardellini. Towards qos-aware serverless function offloading in the edge-cloud continuum through reinforcement learning. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1073–1080. IEEE, 2025.

[74] Álvaro Santos, Jorge Bernardino, and Noélia Correia. Automated application deployment on multi-access edge computing: A survey. *IEEE Access*, 2023.

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

[76] Himani Sharma and Navdeep Kanwal. Video surveillance in smart cities: current status, challenges & future directions. *Multimedia Tools and Applications*, 84(16):15787–15832, 2025.

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

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

[79] SUSE Rancher Prime. RKE2: Rancher's Enterprise-Ready Kubernetes Distribution. https://docs.rke2.io/. Online; Accessed on 8 Oct. 2025.

[80] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.

[81] The Fission Team. Fission: Serverless functions for kubernetes. https://fission.io/. Online; Accessed on 8 Oct. 2025.

[82] The RabbitMQ Team. RabbitMQ: Open source message broker. https://www.rabbitmq.com/, 2025. Online; Accessed on 8 Oct, 2025.

[83] Ilyas Toumlilt, Pierre Sutra, and Marc Shapiro. Highly-available and consistent group collaboration at the edge with colony. In *Proceedings of the 22nd International Middleware Conference*, pages 336–351, 2021.

[84] Animesh Trivedi, Lin Wang, Henri Bal, and Alexandru Iosup. Sharing and caring of data at the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[85] Shahrokh Vahabi, Francesca Righetti, Carlo Vallati, and Nicola Tonellotto. Energy-efficient resource management for real-time applications in faas edge computing platforms. In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pages 1–6, 2023.

[86] Renchao Xie, Qinqin Tang, Shi Qiao, Han Zhu, F. Richard Yu, and Tao Huang. When serverless computing meets edge computing: Architecture, challenges, and open issues. *IEEE Wireless Communications*, 28(5):126–133, 2021.

[87] Xuyi Yao, Ningjiang Chen, Xuemei Yuan, and Pingjie Ou. Performance optimization of serverless edge computing function offloading based on deep reinforcement learning. *Future Generation Computer Systems*, 139:74–86, 2023.

[88] Fan Zeng, Chuan Pang, and Huajun Tang. Sensors on internet of things systems for the sustainable development of smart cities: a systematic literature review. *Sensors*, 24(7):2074, 2024.