Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

SCOPE: Stateful Serverless in the Edge-to-Cloud Continuum

Arianna Dragoni

arianna.dragoni@polimi.it

Politecnico di Milano

Paolo Longo

Politecnico di Milano

Alessandro Margara

Politecnico di Milano

Research Article

Keywords: Edge computing, edge-to-cloud continuum, stateful serverless computing, offloading, migration, distributed systems

Posted Date: November 5th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9815413/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

SCOPE: Stateful Serverless in the Edge-to-Cloud Continuum

Arianna Dragoni[1*], Paolo Longo[1] and Alessandro Margara[1]

[1*]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy.

*Corresponding author(s). E-mail(s): arianna.dragoni@polimi.it;
Contributing authors: paolo.longo@mail.polimi.it;
alessandro.margara@polimi.it;

Abstract

Distributed applications increasingly demand reduced response times to improve user experience. The edge computing paradigm responds to this need by exploiting the resources at the edge of the network to handle user requests close to where they originate, thus avoiding the cost of communication with a remote cloud infrastructure. Most state-of-the-art edge computing solutions build on the serverless model, where developers implement their application as a composition of functions, and the runtime manages and abstracts away deployment and execution in the edge-to-cloud continuum. However, current approaches focus predominantly on stateless functions, which limits their applicability to applications that require robust state management.

Our research aims to overcome the above limitations by proposing SCOPE, *stateful* FaaS programming model and execution platform for the edge-to-cloud continuum. The SCOPE programming model introduces the concept of *sessions* to represent portions of the application state that are relevant to a given user or group of users. Serverless functions can create, access, and modify sessions associated to the users who invoked them. The SCOPE platform automatically and dynamically decides where to store sessions in the edge-to-cloud continuum and where to execute functions, balancing between proximity to clients – to reduce response time – and availability of resources – not to overload edge devices with limited resources. Our evaluation shows the benefits of using SCOPE in heterogeneous edge-to-cloud deployments in terms of response time as well as the ability of SCOPE to adapt to mutating operational conditions.

Keywords: Edge computing, edge-to-cloud continuum, stateful serverless computing, offloading, migration, distributed systems

1 Introduction

Cloud-centric data processing and management solutions have long been the dominant approach to build scalable distributed applications. In these solutions, the application state is stored in the cloud and all requests originating from end users are transferred to the cloud and handled by cloud services [1]. This approach simplifies the deployment and operations of applications, as all components run in the same cloud environment, and cloud services offer scalable, flexible, and cost-effective access to compute power and storage [2].

However, with the expansion of areas such as Internet of Things (IoT), smart cities, and industrial automation new requirements have emerged for which conventional cloud solutions may no longer be suitable. These application domains challenge the efficacy of traditional cloud-centric architectures due to latency and bandwidth constraints associated to long-distance data transfer. IoT devices, sensors, and smartphones generate huge amounts of data at the edge of the network [1], and data transfer may become a bottleneck in the cloud-based computing paradigm [3], as communication costs increase with the distance between actors, leading to delays that may violate applications' requirements.

To satisfy these new needs, the edge computing paradigm promotes the use of resources at the edge of the network, enabling data processing and management close to end users, thus reducing communication latency and optimizing bandwidth usage.

Edge computing is a broad term that embraces heterogeneous approaches [4]. However, many solutions proposed by researchers and practitioners build on a serverless Function-as-a-Service (FaaS) programming model [5], already popular in cloud environments. In the FaaS model, developers implement applications as compositions of functions without worrying about infrastructural concerns, and an execution platform dynamically allocates resources for functions based on the actual load of requests. In edge scenarios, functions may run close to the clients invoking them, provided that nearby nodes have sufficient resources, thus reducing communication delay and enhancing user experience. This makes the FaaS model ideal for running applications in the edge-to-cloud continuum, as the FaaS platform can autonomously decide where to execute functions – in the edge, close to the users, or in the cloud – based on the actual load and the availability of resources, abstracting away deployment concerns in large and heterogeneous compute infrastructures.

However, current FaaS platforms focus predominantly on stateless functions, which limits their applicability, or delegate state management to cloud-centric services [6], thus losing the benefits of data locality.

Our research aims to overcome the above limitations by proposing SCOPE, a *stateful* FaaS programming model and execution platform for the edge-to-cloud continuum. The SCOPE programming model introduces the concept of *sessions* to represent portions of the application state that are relevant to a given user or set of users. Serverless functions can create, access, and modify sessions associated to the users who invoked them. The SCOPE platform automatically and dynamically decides where to store sessions in the edge-to-cloud continuum and where to execute functions, balancing between proximity to clients – to reduce response time – and availability of resources – not to overload edge devices with limited resources.

Table 1: Comparison of FaaS solutions.

	Type	Decentr.	State management	Scheduling policy	Locality awareness	Resources management
Sledge	Runtime	–	Stateless	–	–	–
tinyFaaS	Runtime	–	Stateless	–	–	–
Faasm	Runtime	–	Stateful	–	–	–
AWS Lambda	Platform (cloud)	–	Stateless	Centralized	–	–
Google Cloud Functions	Platform (cloud)	–	Stateless	Centralized	–	–
Azure Functions	Platform (cloud)	–	Stateless	Centralized	–	–
Cloudflare Workers	Platform (cloud)	✓	Stateful	Distributed	✓	✓
AWS IoT Greengrass	Platform (cloud)	–	Stateless	Centralized	✓	✓
OpenFaaS	Platform	–	Stateless	Centralized	–	–
OpenWhisk	Platform	–	Stateless	Centralized	–	–
Serverledge	Platform	✓	Stateless	Distributed	✓	✓
Enoki	Platform	✓	Stateful	–	–	–
Vahabi et al.	Platform (ext.)	✓	Stateless	Distributed	–	✓
Cicconetti et al.	Platform (ext.)	✓	Stateless	Hybrid	✓	✓
Neptune	Platform (ext.)	✓	Stateless	Centralized	–	✓
SCOPE	Platform	✓	Stateful	Distributed	✓	✓

SCOPE is configured with information about the location of processing nodes in the edge-to-cloud continuum and their resources. It automates the placement of functions, the allocation of processing and storage resources for sessions and functions, and the migration of sessions across nodes based on the location of clients, the resources of nodes, and their actual load.

The paper presents the SCOPE programming model, and the design and implementation of the SCOPE platform in detail. It evaluates the benefits of using SCOPE in heterogeneous edge-to-cloud deployments in terms of response time as well as the ability of SCOPE to adapt to mutating operational conditions.

In summary, SCOPE contributes to the research on edge computing by proposing a programming and execution model that simplifies the effective exploitation of resources at the edge of the network, considering data locality and resource availability, and making them transparent to application developers.

The paper is organized as follows. Sec. 2 surveys existing approaches related to the FaaS model and the edge computing paradigm. Sec. 3 presents the core concepts behind SCOPE. Sec. 4 presents the API and implementation details of SCOPE. Sec. 5 evaluates the benefits of SCOPE in real-world deployments. Finally, Sec. 6 provides concluding remarks, suggesting directions for future research endeavors.

2 Related Work

The FaaS programming model allows developers to build and execute applications that scale automatically without the need for explicit server management and resource provisioning [7]. Various FaaS solutions have been proposed in the literature, including some designed for edge scenarios. We identify the key features that distinguish different FaaS solutions and categorize them, as illustrated in Table 1.

We consider different *types* of solutions, each focusing on specific concerns. *Runtime* solutions mainly investigate lightweight virtualization techniques that enable efficient execution of functions, thus promoting the adoption of the FaaS paradigm in resource-constrained environments, such as edge nodes. *Platforms* provide full-fledged implementations of the FaaS paradigm. We further indicate when a solution is *cloud*-based platform – i.e., it is offered by a (public) cloud vendor that manages and controls the infrastructure, and when it represents an *extension* of existing platforms.

We indicate when solutions support *decentralization*, meaning that they enable deployments where the computing infrastructure is spread across different geographical locations.

We further classify solutions based on their *state management* capabilities: *stateless* solutions only support functions that do not maintain state, while *stateful* solutions allow state sharing across invocations.

The *scheduling policy* defines the topology of the component responsible for scheduling function executions across the available computing nodes. A solution is considered *centralized* when a single controller is in charge of all scheduling decisions, *distributed* when nodes independently communicate to forward requests, or *hybrid* when scheduling decisions are handled by a subset of selected nodes. Scheduling policies may consider different criteria. In Table 1, we indicate when they consider the geographical location of the infrastructure nodes and the clients (*locality awareness* column in Table 1), and the current usage of resources (*resource management* column in Table 1).

We have selected various FaaS solutions and categorized them based on the above classifications. In the following, we analyze their similarities and differences and position SCOPE in this context.

Sledge [8], tinyFaaS [9] and Faasm [10] focus on optimizing function execution through efficient runtime environments, especially in resource-constrained settings. Sledge [8] is designed for short-lived computations and uses WebAssembly to create a lightweight execution environment that is well suited for running serverless functions on nodes with limited resources. TinyFaas [9] targets edge computing scenarios but relies on containerization to enable isolated execution of functions. Both Sledge and tinyFaaS are designed for stateless functions and lack built-in abstractions for state management. In contrast, Faasm [10] supports stateful functions. Specifically, each function in Faasm is executed in a Faaslet, a lightweight container that manages execution while providing isolation. The memory is divided into different regions: each Faaslet has its own private memory region, while a shared memory region enables controlled data exchange between multiple Faaslets. In general, runtime solutions focus on the execution of functions on a specific node in terms of computation, resource management and memory utilization and do not provide mechanisms for distributing or scheduling requests across multiple nodes.

Platform solutions go beyond the execution of functions within a single node. They are designed to manage a computing infrastructure consisting of multiple nodes. This includes coordinating the distribution of workloads, scheduling requests, and ensuring proper communication between nodes in a distributed environment. All public

cloud vendors offer a FaaS platform solution, such as AWS Lambda [1], Google Cloud Functions [2], Azure Functions [3], and CloudFlare Workers [4]. Except for CloudFlare Workers, none of these solutions provides built-in abstractions for application state management. Instead, they all rely on external state management services, such as cloud-hosted databases, forcing developers to explicitly implement the interaction with these services within their functions.

CloudFlare Workers [11] represents an exception in this scenario, as it offers the CloudFlare Durable Objects [12] API to manage application state by leveraging a globally distributed network of data centers. Each Durable Object embeds a portion of an application's state, it is stored in a single data center, but is accessible from any node in the network. When a client needs to access the object's state, the request is automatically forwarded to the data center hosting the object. If there is a flood of requests from a specific geographic region, CloudFlare can dynamically relocate the Durable Object to a data center closer to the area where the requests originate, improving response times by reducing the distance between the requesting clients and the object. This mechanism positions CloudFlare Workers among solutions that schedule the execution of functions in a distributed manner across nodes, taking into account both locality and resource utilization. However, the Durable Object abstraction is only available within the CloudFlare infrastructure, which prevents its adoption in self-managed deployments.

AWS IoT Greengrass [13] brings some AWS services on local edge devices, including AWS Lambda serverless functions. It exploits the locality of edge nodes to minimize the interaction between the clients and the services. However, when considering serverless functions, it preserves the same programming model offered by AWS Lambda in the cloud, which lacks explicit abstractions for state management. AWS IoT Greengrass considers a fixed two-layers network topology, where edge devices are individually and statically programmed to act on the data that is produced locally, enabling them to filter and aggregate data before delivering it to the cloud. The platform only works in the AWS ecosystem, preventing the adoption of complex network topologies and the integration with other cloud solutions.

Open source platforms overcome this problem, allowing self-managed deployments that enable the adoption of the FaaS programming paradigm without being dependent on public cloud providers. The most popular open source solutions are OpenFaaS [5] and OpenWhisk [6]. Both of them only support stateless functions. They assume a centralized infrastructure where all components are deployed within a single data center, and they do not account for the location of nodes when scheduling functions.

Serverledge [14] is a decentralized open-source platflorm that leverages WebAssembly to minimize the overhead of function execution. Its scheduling functionalities are distributed among nodes, as each node can decide whether to locally execute or redirect the requests it receives, based on the available resources. Serverledge organizes nodes into edge zones or cloud regions, prioritizing communication within the same

[1] https://aws.amazon.com/pm/lambda/
[2] https://cloud.google.com/functions
[3] https://learn.microsoft.com/azure/azure-functions/
[4] https://developers.cloudflare.com/workers/
[5] https://www.openfaas.com
[6] https://openwhisk.apache.org

zone to minimize latency [14], making its scheduling strategy location aware. However, it is designed for stateless serverless functions and does not provide any state abstraction for supporting stateful functions.

Enoki [15] is a decentralized open-source platform that supports stateful functions by integrating a key-value store to manage and maintain state across function executions. In Enoki, state is managed through data containers that are replicated on the node executing the function. This design allows functions to access data locally, minimizing computation overheads. Specifically, when a client invokes a function on a node, Enoki creates an instance of the function and checks for the availability of the data container. If the container is available on some other node, Enoki replicates the container on the local node. If not, it creates a new container. Updates to the data are then propagated to all nodes that hold the container. However, Enoki does not consider scheduling strategies based on geographical position of nodes or based on resource balance among nodes.

Some works extend existing platform to optimize scheduling with respect to client locality and/or availability of resources. Vahabi et al. [16] propose a model for scheduling serverless function requests that prevents resource saturation in nodes. The model considers the energy required to process each function and assigns it to the node with sufficient resources that can handle the function most efficiently. However, this model does not consider the geographical distribution of nodes.

Cicconetti et al. [17] propose a model where clients send requests to *e-routers* that redirect them to *e-computers* responsible for performing the computations. The algorithm for selecting the best e-computer balances the minimization of delay with fair resource utilization. Routers assess each request and assign it a weight relative to each e-computer, which is then used to choose the most suitable e-computer for executing the request. This results in a hybrid scheduling policy that considers both locality and resource utilization.

Neptune [18] extends OpenWhisk with scheduling policies that are designed to optimize the use of heterogeneous resources (e.g., GPUs for ML tasks) in edge-to-cloud scenarios. It relies on K3s, a lightweight distribution of Kubernetes to manage containers across nodes.

None of the platform extensions discussed above accounts for stateful functions.

Several works formulate functions and data placement and migration in heterogeneous edge-to-cloud environments as optimization problems [6, 19–23]. They differ in terms of the assumptions they make (e.g., presence of heterogenous hardware or presence of state), on the optimization criteria (e.g., minimizing energy consumption or maximizing load balancing), and on the techniques to solve the problem (e.g., linear programming or heuristics). The interested reader can refer to the comprehensive survey by Bellendorf and Mann [24], which provides a classification of recent approaches.

These works are orthogonal to our proposal, which provides a full-fledged platform for serverless state computing. In principle, SCOPE could incorporate some of the proposed optimization strategies as part of its scheduling policies.

In summary, while several solutions address individual aspects of stateful or edge-based FaaS, no existing FaaS platform jointly addresses stateful execution, data

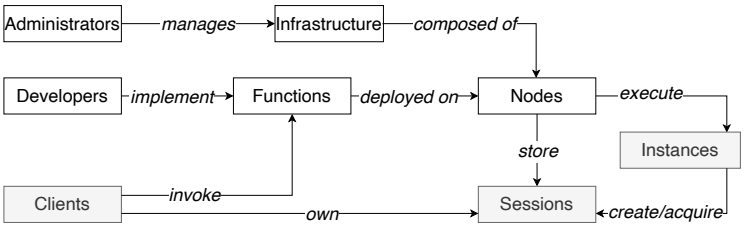


Fig. 1: Overview of concepts and their relations.

locality, and resource-aware scheduling in the edge-to-cloud continuum. Specifically, it lacks (i) a platform that allows for in-memory data access and storage to maintain state close to the computation platform, (ii) mechanisms to prioritize communication between nearby nodes, (iii) mechanisms to effectively balance computational and storage resources between nodes.

SCOPE aims to close this gap by integrating these capabilities into a single platform. It extends the serverless FaaS paradigm for edge scenarios by introducing sessions to manage application state. To optimize performance, it supports dynamic migration of sessions between nodes within the edge-to-cloud infrastructure. Just like Enoki, it includes a key-value storage within the nodes to support stateful functions. Similar to CloudFlare, state is kept as close to the client as possible, with the difference that Durable Objects are tightly integrated into Cloudflare's infrastructure and state is still managed in the data centers rather than directly in a custom edge node. In addition, SCOPE distributes its scheduling capabilities across the nodes and considers both client (and data) locality and resource utilization in its scheduling policies.

3 Programming and Execution Model

This section presents the programming and execution model of SCOPE. Its *programming model* lets developers build stateful applications as serverless functions that operate on application state organized into sessions. Its *execution model* automatically places functions and sessions across an edge-to-cloud infrastructure and migrates them to keep state close to the clients that use it, while balancing the load on nodes.

Sec. 3.1 presents the programming model. The remaining subsections describe the execution model: Sec. 3.2 defines how each node models the resources it offers and their utilization; Sec. 3.3 describes how nodes are organized into a logical hierarchy; Sec. 3.4 explains how a client determines which node to contact when invoking a function; Sec. 3.5 details how and when SCOPE migrates sessions across the infrastructure; and Sec. 3.6 describes how sessions can be optionally replicated across nodes.

3.1 Programming model

Fig. 1 shows the core concepts of SCOPE and how they interact. *Developers* write serverless *functions* and *clients* invoke them, owning the *sessions* that hold the application state. Both functions and sessions are placed on the *nodes* of the edge-to-cloud infrastructure, which is managed by *administrators*. In SCOPE, functions are *stateful*:

they create, access, and modify *sessions*, the units into which the application state is organized. A session is a self-contained unit of state that exists independently of any function and persists across invocations.

This decoupling of state from computation distinguishes SCOPE from related stateful models. Unlike virtual actor models such as Orleans [25] or Durable Objects [12], where state and functions are tightly coupled within the same entity, SCOPE completely decouples them. In SCOPE, clients are the owners of the state (sessions) and can dynamically deploy new serverless functions that acquire and operate on their state.

Clients interact with the application by invoking its functions on a node. SCOPE runs each invocation as a function *instance*, which can create new sessions and access existing ones during its execution. When a function creates a session, SCOPE returns a *session token* to the invoking client. The token identifies the session and records where it is currently stored. When a function accesses an existing session, the client attaches the corresponding token to the invocation, and SCOPE routes the request to the node holding the session (Sec. 3.4). Clients thus need no knowledge of the infrastructure, as tokens are sufficient to reach their state.

We consider clients as *logical client* that may correspond to a single end-user or to a group of users sharing the same portion of the application state (e.g., devices in a household or sensors deployed in the same building). The logical client also serves as the unit of locality, since SCOPE assumes that the physical clients behind a logical client are geographically close, allowing a single session placement to be near all of them simultaneously. When a logical client represents a group of real clients, its members share the same session by sharing its token. Distributing the token within the group is left to the application, which already knows the group's membership. SCOPE provides no separate discovery service, as access to a session is always mediated by its token. By tying locality to the logical client and confining each invocation's state to a single session, SCOPE achieves tractable, locality-aware placement. The session thus becomes the granularity for state placement and migration, allowing data to dynamically follow its logical client. This model fits applications whose state naturally partitions into per-client units known a priori. Access patterns that cannot be bound to a specific client are better served by complementary mechanisms, such as content-distribution networks for read-only data.

To exemplify, consider a scenario of patient monitoring in a hospital. Each patient (client) has sensors that record their health parameters. These parameters represent a session specific to a client, and together with the parameters of the other clients (i.e., sessions of other clients), form the state of the application. Now consider a stateful function that identifies anomalies for a specific patient. When the function is called for a patient, it only accesses the session relative to that patient. SCOPE automatically decides where to store the state of patients and where to execute functions that access the state of patients based on the deployment infrastructure and on the specific load of requests.

The anomaly-detection function of the example above is implemented in SCOPE as shown in Listing 1: it acquires the patient's session, appends the new reading, and raises an alert if the readings become anomalous.

Listing 1: A stateful function that records a patient reading and checks for anomalies.

```
func MonitorPatient(req Request) Response {
    token := req.SessionToken
    return AcquireSession(token, ReadWrite, func(s *Session) Response {
        s.Readings = append(s.Readings, req.Reading)
        if isAnomalous(s.Readings) {
            return Alert(s.PatientID)
        }
        return Ok()
    })
}
```

3.2 Anatomy of a node

Nodes are geographically distributed and may have highly heterogeneous capabilities, ranging from end-user devices, to intermediate edge nodes, up to cloud data centers. In SCOPE, nodes offer resources, execute functions, and store sessions. Function executions and sessions occupy the resources of the nodes on which they are deployed.

SCOPE provides a flexible model to represent the resources of a node, where developers are free to indicate multiple types of resources and their availability within that node. As an example, developers may be interested in computational and memory resources and can indicate two distinct values to define the amount of each resource available within a node. This model accommodates complex scenarios with heterogeneous or ad-hoc resources. For instance, in scenarios that involve machine learning (ML) workloads, nodes may be characterized based on their availability of resources such as GPUs or other ML-specific accelerators.

Functions and sessions consume resources of the node storing on which they are deployed. For instance, a function consumes computational and memory resources for the entire duration of its execution. Likewise, a session may consume storage resources for persisting its data, as well as computational resources based on the number of invocations of serverless functions that need to acquire the session.

More formally, a node n has a maximum capacity C_r^n for each resource r. SCOPE operates under the assumption that this capacity is not an absolute upper limit beyond which no new functions can be executed, but exceeding it may lead to a degradation in quality of service, in terms of increased response times.

Let us call $I_{f,r}$ the amount of resource r occupied by the execution of a stateless function f, and $I_{s,r}$ the amount of resource r occupied by a session s. Notice that $I_{s,r}$ involves also the processing resources required to execute the functions that access session s as part of their execution.

The sum of the usage impacts of resource r for all functions executed in node n and for all sessions stored within node n estimates how much of resource r is consumed within that node. Therefore, each node is assigned to a *usage index* for each resource r as:

$$U_r^n = \frac{1}{C_r^n} \cdot (\sum_f I_{f,r} + \sum_s I_{s,r})$$

This metric estimates how much of the resource r is used within the node, in relation to the total capacity of the node. SCOPE allows the definition of a threshold for resource utilization I^{th}, which determines when no further sessions should be assigned to a node. This threshold is configurable and can be adjusted depending on the predictability of the application workload. In the case of constant and predictable workloads, a higher threshold can be set to maximize resource utilization. Conversely, in environments with fluctuating workloads and peaks of utilization of certain sessions, a lower threshold can be beneficial to keep performance stable and avoid overloads.

3.3 Deployment infrastructure

SCOPE requires a definition of the deployment infrastructure. We assume a team of infrastructure *administrators* responsible for defining and managing it. For instance, the role of administrators may be fulfilled by a telecommunication infrastructure company that provides and interconnects edge resources with public cloud infrastructures, or by the system administrators of a private company.

In SCOPE, nodes are organized in a logical hierarchy based on the assumption that nodes at a higher level always provide more resources than nodes at a lower level.

Administrators assign nodes to levels in the hierarchy. For each level l, we define N_l as the set of nodes in l. Within the hierarchy, except for nodes at the highest level, each node n in N_l has a single *parent* node p in N_{l+1}. We also say that n is a *child* of p. Although the choice of a parent node is arbitrary, in practice SCOPE works best if the latency between a node and its parent is low. The current implementation relies on the geographical distance to approximate the network distance (and hence the latency) between nodes. This is a common choice in geographically distributed systems, as exemplified by peer-to-peer protocols such as Vivaldi [26] and Phoenix [27].

We currently assume that the geographical location of nodes (latitude and longitude coordinates) is fixed and made available as a static configuration parameter. Instead, clients can be mobile (i.e., they can change their position over time), in which case they may either advertise their position or the position may be inferred based on their network addresses.

SCOPE exploits the logical hierarchy to distribute information about the current status of nodes, that is, about changes in the usage indexes of nodes, through an HTTP API. Specifically, each node periodically notifies its parent and its children about its current usage indices of resources. Moreover, nodes at the same level continuously exchange aggregate information about their children using a gossip protocol.

The failure of a node n is detected when there is a failure in the communication between nodes, specifically during the communication of state updates. When node n fails, all of its children initiate a procedure to identify a new parent, which is again chosen as the one with minimum geographical distance among those in the upper level. If the original node resumes working, it contacts all the nodes that were previously its children and asks them to return to their original parent. This ensures that the hierarchical structure and the distribution of tasks within the network return to their pre-failure state.

3.4 Client resolution

To interact with the system, clients need to determine which node to contact when they want to invoke a serverless function. We define this process as *client resolution*. SCOPE exploits client resolution to reduce communication latency and to balance resource usage in the network.

There are three possible cases: (i) the client wants to execute a stateless function, that is, a function that does not acquire any session; (ii) the client wants to execute a function that creates a new session; (iii) the client wants to execute a function that accesses an existing session owned by that client.

In the case of a stateless function, SCOPE aims to balance between locality and optimization of resources, that is, it tries to execute the function as close as possible to the invoking client if nearby nodes have enough resources. To identify nearby nodes, SCOPE may rely on different strategies that depend on the network protocols available. One possible strategy is to assign anycast IP addresses to nodes, allowing the system to route clients to the physically closest node. After the client has resolved the IP address of a node, it sends the request. If the node that receives the request has sufficient resources, it accepts it and processes it. Otherwise, it forwards the request to the parent node, which is assumed to be the closest node with more available resources based on the network design.

The same protocol also applies to the case of function executions that create a new session, with two differences: nodes will consider the resources required by the session to decide whether to accept or to redirect the invocation, and the response to the client will return a *token* that uniquely identifies the created session and contains the address of the node where the session is stored. To determine whether a node is capable of executing a function, SCOPE automatically tracks both the resources used, which for new sessions are specified by the client or set to a default value, and the resources available on each node during the execution of functions.

Finally, when a function invocation refers to an existing session (it contains the session token as part of its parameter), the client directly invokes the node that stores the session, as indicated in the token.

3.5 Session migration

Sessions consume the resources of the nodes storing them. To adapt to mutating operational conditions, such as saturation of a node's resources or mobility of clients, SCOPE adopts a *session migration* process that dynamically moves sessions across nodes. This section explains the process in details.

Functions can acquire a session in non-exclusive (read-only) or exclusive (read-write) mode. The acquisition mode influences the state of a session. SCOPE avoids sessions to be migrated while a function is modifying them. Sessions that are currently acquired in read-write mode cannot be migrated and need to wait until the acquiring function releases them. When a migration is performed, the session being migrated can still be acquired in read-only mode. Any attempt to acquire the session in read-write mode fails and may either wait or retry, depending on how the platform is configured.

When a client invokes a function on a node and the session to be acquired has been migrated, the node redirects the client to the new owner of the session.

3.5.1 Triggering conditions

We identified the following four conditions to trigger a migration process, which aim to ensure that SCOPE dynamically adapts to changing traffic patterns and customer behaviors.

High invocation load. A node that receives a high volume of function invocations (with respect to its resources) due to the sessions it currently owns may trigger the migration process to offload some sessions to less burdened nodes.

Imbalanced children. If a node detects an imbalance in the usage index of its children, it can trigger migration between children with a high usage index and children with a low usage index.

Mobility of the clients. The movement of a client causes SCOPE to recalculate its closest node: if this node changes, sessions of that client may be migrated to the closest node.

Manual. Finally, a session migration may be triggered programmatically within a function. This may be used for instance to guarantee that a session is stored within a node belonging to a specific administrative domain.

3.5.2 Determining a candidate node

When a migration is triggered, SCOPE needs to determine a candidate node for the session to be migrated. To do so, it considers two types of information: (1) static information about the logical topology (hierarchy of nodes), and (2) dynamic information about resource utilization.

Specifically, for a given session s and for a given node n, let us define the *placement index* P_s^n that balances the current usage of n (I^n) and the distance between node n and the client owner of session s ($dist_s^n$).

$$P_s^n = I^n \cdot dist_s^n$$

Where I^n is an aggregate value of the usage indexes across all resources. The current implementation of the platform uses the maximum for simplicity, but other functions may be used to better capture the requirements of applications in specific domains. Determining a candidate node for migrating session s involves finding a node n with a low placement index.

In SCOPE, every node has the complete view of the logical topology from the highest level (included) to the lowest level (excluded). As nodes in the network are expected to be fairly stable, this enables fast navigation of the hierarchy. The exclusion of the lowest level limits the amount of information that each node needs to store, also considering that the lowest level may be subject to more frequent changes, as its nodes may be less stable. Instead, resource utilization continuously fluctuates over time. Accordingly, this information is not distributed across all nodes, but only across

Algorithm 1 Determining a candidate node for a session.

Require: session s owned by client c_s; utilization threshold I^{th}
Ensure: a candidate node for s, or $\perp$ if none is found

```
1:  x ← node with a complete view of the surroundings of c_s
2:  while x ≠ ⊥ do
3:    best ← ⊥;    P_best ← +∞
4:    for all node n among the direct and indirect children of x do
5:      if I^n < I^th then
6:        P_s^n ← I^n · dist_s^n {placement index}
7:        if P_s^n < P_best then
8:          best ← n;    P_best ← P_s^n
9:        end if
10:     end if
11:   end for
12:   if best ≠ ⊥ then
13:     return best
14:   end if
15:   x ← parent of x
16: end while
17: return ⊥
```

the levels of the hierarchy. Specifically, parents know the usage index of their children and the (aggregate) usage index of their siblings.

Let us call c_s the client owner of the session s to be migrated. Algorithm 1 formalizes how SCOPE determines a candidate node for s. It is run by the node that currently stores s, which triggers the migration upon detecting its own overload. The procedure operates internally within SCOPE: its outcome is handled locally by that node and is never exposed to the client c_s, which is unaware of it. Based only on geographical information, the search starts from a node x that has a complete view of the surroundings of c_s, within a distance defined as a parameter (line 1). Node x iteratively inspects its direct and indirect children and computes the placement index of every node that still has spare capacity, that is, whose aggregate usage index is below the threshold I^{th} (lines 4–8). If at least one such node exists, the one with the lowest placement index is selected as the candidate (lines 9–10). Otherwise, the search moves one level up in the topology and restarts from the parent of x (line 11), until a candidate is found or the hierarchy has been explored up to its highest level.

While the above discussion focuses on a single session, in practice SCOPE may consider the migration of multiple sessions simultaneously for several reasons: (i) If a node triggers a migration because it is overloaded (high invocation load triggering condition, see Sec. 3.5.1), it may attempt to migrate multiple sessions to rapidly reduce its load. (ii) Under the same triggering condition, the migration of a session may be impossible if the session is currently acquired in read-write mode. Considering multiple sessions increases the probability that some of them are not currently acquired and can be migrated. (iii) If a client moves (mobility of the clients triggering condition, see Sec. 3.5.1), it may be beneficial to move all of its sessions to a nearby node.

Algorithm 2 Migration protocol for a session.

Require: session s; candidate node d; last visited node ℓ of s
Ensure: s is stored on d and reachable by clients through ℓ
1: wait until s is not acquired in read-write mode
2: mark s as *offloading* {read-only acquisitions are still served}
3: open a stream over the state of s from the local storage
4: $loc \leftarrow$ transfer s to d, which stores it and returns its location
5: **if** the transfer of s to d failed **then**
6: mark s as *available* {the local copy of s is preserved}
7: **return** failure
8: **end if**
9: notify the last visited node ℓ of loc
10: **if** ℓ did not acknowledge the new location **then**
11: **return** failure
12: **end if**
13: replace the local copy of s with a forwarding reference to loc
14: **return** loc

3.5.3 Migration protocol

Once the candidate node has been selected, SCOPE performs a migration protocol to move sessions from the current node to the selected candidate node. At the end of the migration protocol, the session will be stored in the candidate node.

Recall that clients retain a token for each session they own, and that the token contains the address of the node where the session was stored when the client last accessed it. Let us call such node the *last visited node*: this is the node that the client will contact the next time it needs to acquire the session.

Migrations take place independently of the clients' interactions, meaning that clients are not automatically notified when some of the sessions they own migrate from node to node. Instead, SCOPE maintains a reference to the node in which a session is currently stored within the last visited node. This way, when a client tries to access the session by contacting the last visited node, its request will be forwarded to the node where the session is currently stored. Notice that, even in the case the session undergoes multiple migrations, the last visited node always remains responsible for maintaining a reference to the current node, enabling clients to retrieve the session with a single redirection.

Algorithm 2 formalizes the protocol, executed by the node that currently stores the session s to be migrated, once a candidate node d has been determined as in Algorithm 1. The protocol operates internally within SCOPE: its outcome is handled locally by the node that performed the migration and is never returned to the client. In line with the previous discussion, the client is not notified of the migration, but discovers the new location of s only upon its next access, through the redirection performed by the last visited node ℓ.

Since SCOPE never migrates a session while a function is modifying it, the protocol first waits until s is no longer acquired in read-write mode (line 1). It then marks s as

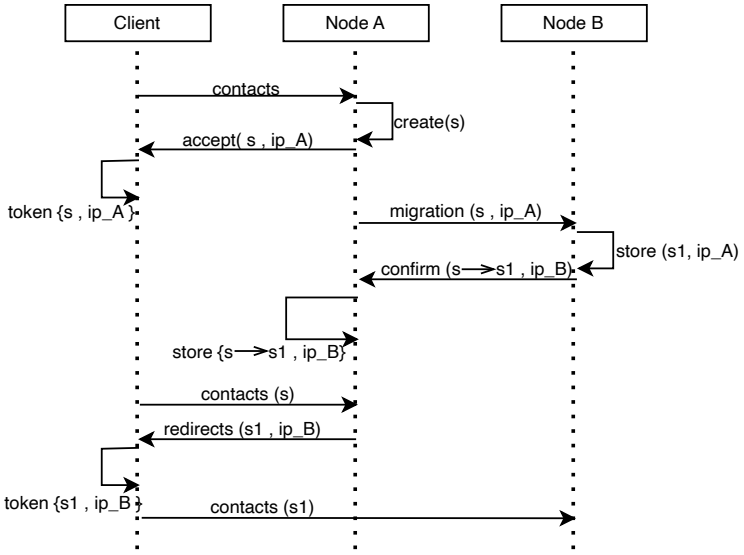


Fig. 2: Sequence diagram of the migration protocol.

offloading: while the migration is in progress, read-only acquisitions are still served, whereas read-write acquisitions are rejected (line 2). The state of s is streamed to the candidate node d, which stores it and returns its new location (lines 3–4); streaming lets the transfer overlap the reading of s from the source storage with its writing into the destination storage. If the transfer fails, the migration is aborted and the local copy of s is preserved, leaving the session available on its original node (lines 5–7). Otherwise, SCOPE notifies the last visited node ℓ, which updates its reference so that it points to the new location of s (line 8). Finally, the node that performed the migration replaces its local copy of s with a forwarding reference to d (line 11), so that clients contacting a node that previously stored s are redirected to its current location.

Fig. 2 describes the migration of a session. Specifically, after the client first contacts a node (node A in Fig. 2), it saves a token containing the session ID and the node's address, which acts as both the last visited node and the current node. SCOPE can trigger a migration to a candidate node (node B in Fig. 2). At the end of the process, the last visited node (node A) is no longer responsible for the session, as it has been substituted by the candidate node (node B). During the next interaction, the client contacts the last visited node (node A in Fig. 2), as its address is stored in the token. This node then informs the client that the session has been migrated, sending the new session ID and the new IP address.

In the case a session is migrated more than once before a client tries to access it, nodes that previously held the session are informed of the migration. This way, if clients contact nodes that were previously responsible for a session, they get redirected to the node that is currently responsible for that session.

3.6 Replication

Edge devices may be unreliable and may intermittently fail during their execution. Fully addressing fault tolerance, however, is out of the scope of this paper: FaaS systems, particularly in edge environments, are inherently best-effort, and SCOPE embraces the same design philosophy.

SCOPE is designed to support session replication across multiple nodes using a single-leader replication protocol [28]. Replication in SCOPE is currently employed to reduce access latency and to optimize the migration process, rather than to guarantee strict durability against failures, although it also provides a best-effort mitigation of the issues that may arise in the case of failures. Let us consider a session `s`: the node in which session `s` is currently stored acts as a leader in the replication protocol. Let us call this node `L`. When a client invokes a function on node `L` that acquires session `s` and changes its content, `L` is responsible for propagating the changes to all replicas.

The replication process could be configured as *synchronous* or *asynchronous*, allowing for a fine-tuned trade-off between fault-tolerance and response time. With synchronous replication, each acquisition cannot be released until the changes to a session have been delivered to all replicas and acknowledged. With asynchronous replication, the leader performs the changes locally and returns to the client before receiving acknowledgments from replicas. In both cases, changes are propagated in the same order in which they occur, ensuring that all (active) replicas receive session changes in the same order.

The strategy for identifying the set of nodes hosting the replicas of a session could be customized depending on the specific requirements of the use case and the characteristics of the nodes. A possible strategy could replicate sessions to the parent node, which is assumed to be more reliable.

In the presence of replication, SCOPE may optimize the migration process. If a session `s` needs to be migrated, SCOPE prioritizes migrating it to one of the nodes storing a replica of session `s`. Indeed, in this case, the migration protocol becomes more lightweight, as the destination node already contains a copy of session `s`.

Likewise, when replication is applied, clients are aware of the replicas for each of the sessions they own. This information is stored in the session token and can be used by clients to autonomously contact one of the session replicas in the case of a failure of the leader replica or a network partition.

This allows them to access the content of the session even in the presence of failures. To do so, clients attach identifiers of their last requests within new invocations. The replica checks these identifiers to see if it has received and applied all the changes related to the requests: in this case, the replica is up-to-date and can process the client request on behalf of the leader.

Table 2: Functions of the SCOPE platform.

	FaaS runtime layer	Storage layer
	Infrastructure management functions	**Infrastructure data API**

Infrastructure manager	`GetParentNodeOf(nodeId)`
	`GetChildrenNodesOf(nodeId)`
	`GetSessionResourceUsage(sessionId)`
	`GetNodeResourcesUsage(nodeId)`
	`UpdateSessionResourcesUsage(sessionID, resourcesUsage)`
	`ResourcesUsageUpdateToParent()`
	`ResourcesUsageUpdateFromChild()`
		`SetClientCoordinates(keys, args)`
		`GetCurrentNodeKey(keys, args)`
		`SetCurrentNodeKey(keys, args)`
		`RegisterNode(keys, args)`
		`RegisterNodeRelation(keys, args)`
		`GetParentNodeOf(keys, args)`
		`GetChildrenNodesOf(keys, args)`
		`FindLookupNode(keys, args)`

	Session management functions	**Session management API**

Session manager	`CreateMigrationRequest(host, sessionId)`
	`IssueMigrationRequest(host, sessionId)`
	`MigrateSession(w, req)`
		`MigrationStart(keys, args)`
		`MigrateData(keys, args)`
		`MigrationFinish(keys, args)`
		`CancelMigration(keys, args)`

	SCOPE API	**Session access API**

Application manager	`AcquireSession(sessionToken, ifAcquired)`
	`CreateAndAcquireSession(ifAcquired)`
	`MaybeCreateAndAcquireSession(sessionToken, ifAcquired)`
	`CreateSession()`
		`CreateSession(keys, args)`
		`AcquireSession(keys, args)`
		`ReleaseSession(keys, args)`

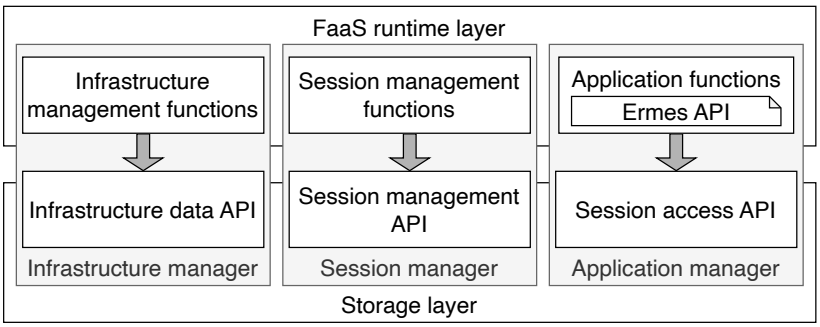


Fig. 3: The SCOPE architecture.

4 API and Implementation

We implemented the SCOPE model into a concrete platform[7]. Fig. 3 shows its high-level architecture, which builds on three main components (gray boxes in Fig. 3), running on each and every node of the distributed infrastructure:

- The `Infrastructure manager` handles information about the topology of the infrastructure and the resources of nodes.
- The `Session manager` handles the deployment and migration of sessions.
- The `Application manager` handles the execution of the applications running on top of SCOPE and persists their state.

The implementation of each component spans two layers:

- The `FaaS runtime` is responsible for executing the logic of each component in the form of serverless functions.
- The `Storage layer` stores the state of each component within nodes.

The current prototype implements the `FaaS runtime` using faasd[8], a lightweight single-node distribution of the OpenFaaS framework suited for edge nodes, and the `storage layer` using Redis[9]. The `infrastructure management functions` and the `session management functions` implement the logic of the `infrastructure manager` and of the `session manager`, respectively. They are implemented as serverless functions written in Go and run on top of the `FaaS runtime`.

Nodes communicate by invoking these functions via HTTP requests, thus implementing the protocols for managing resources depicted in Sec. 3.

Developers can implement their applications as a set of serverless `application functions` that invoke the `SCOPE API` to acquire, read, and modify sessions. The `SCOPE API` represent a standardized interface for applications. We currently provide an implementation of this

[7] The source code and docs are available at https://anonymous.4open.science/r/SCOPE_framework/README.md
[8] https://github.com/openfaas/faasd/
[9] https://redis.io/

interface for the Go language, but we plan to offer bindings for other common languages for Web applications such as JavaScript in the future.

Likewise, serverless functions running in the `infrastructure manager`, `session manager`, and `application manager` interact with the `storage layer` via standardized API. Specifically, the `infrastructure data API` enables storing and retrieving topological data about the infrastructure and the current use of resources within nodes. The `session management API` provide access to information and data about sessions. The `session access API` enable applications to acquire, access, and modify application state in the form of sessions. The current SCOPE prototype implements these interfaces within an adapter for Redis. Adopting different storage engines would only require providing custom adapters to expose the same interfaces.

Table 2 lists the interfaces defined by the components of the SCOPE architecture within each layer. Next, we define each component in detail, summarizing the role of these interfaces.

4.1 Infrastructure manager

The `infrastructure manager` implements the protocol to exchange information about the topology of nodes within the infrastructure and the amount of resources made available within each node.

Listing 2: JSON file of the infrastructure

```
{
"areaName": "<name>",
"host": "<ip>" ,
"geoCoordinates": {"latitude": <lat>, "longitude":<long>},
"tags": {<tag>: <value>},
"resources": {<resource>: <value>},
"areas": [...]
}
```

In the current implementation, we assume the network topology to be immutable. Infrastructure administrators define it in JSON format within a file that is distributed to all nodes. We plan to implement dynamic changes of topology in the future.

Listing 2 depicts the schema of such JSON file. The infrastructure consists of a hierarchy of `areas`, each containing a main node that manages the area and the set of nodes within its subtree. The `areaName` field represents the unique identifier of the area managed by the `host` node, which has fixed `geoCoordinates`. Optional key-value `tags` may be associated with the area. Field `resources` contains the resources made available by the nodes in the area. Field `areas` contains the sub-areas in the hierarchy: every element in this list has the same structure as the enclosing object, thus enabling a recursive definition of the network hierarchy.

The `infrastructure management functions` allow nodes to (i) exchange information about their relations within the hierarchy (`GetParentNodeOf`, `GetChildrenNodesOf`), (ii) exchange information about the actual usage of resources (`GetSessionResourceUsage`, `GetNodeResourcesUsage`), and to (iii) update and propagate resource usages (`UpdateSessionResourcesUsage`, `ResourcesUsageUpdateToParent`, `ResourcesUsageUpdateFromChild`).

The `infrastructure data API` allows the `infrastructure management functions` to access the storage to read and modify infrastructure parameters (`SetClientCoordinates`,

`GetCurrentNodeKey`, `SetCurrentNodeKey`, `RegisterNode`, `RegisterNodeRelation`, `GetParentNodeOf`, `GetChildrenNodesOf`, `FindLookupNode`).

4.2 Session manager

The `session manager` implements the logic to manage the deployment and migration of sessions. To do so, nodes invoke `session management functions` to request and handle the migration of groups of sessions. Specifically, these functions are used to: (i) prepare a migration request (`CreateMigrationRequest`); (ii) initiate the migration process (`IssueMigrationRequest`); (iii) execute the migration process (`MigrateSession`); (iv) handle and complete requests through the `Handle` function.

The `session management API` enables the `session management functions` to access storage during the migration, to retrieve and update information about current sessions, including their frequency of access by application functions and the resources they consume (`MigrationStart`, `MigrateData`, `MigrationFinish`, `CancelMigration`).

4.3 Application manager

The `application manager` handles the execution of the serverless `application functions` that implement the application logic. These functions use the `SCOPE API` to create, acquire, read and modify sessions. Specifically, (i) the `CreateSession` function create a new session; (ii) the `AcquireSession` function acquire the session within the `sessionToken` and run the `ifAcquired` function; (iii) the `CreateAndAcquireSession` function create a new session and acquire it to perform the `ifAcquired` function; (iv) the `MaybeCreateAndAcquireSession` acquire the session within the `session token` and perform the `ifAcquired` function – if there is no `sessionToken`, it first creates a new session.

To save and access the sessions from the storage layer, these functions invokes the `session access API`, which includes a function to create sessions (`CreateSession`), a function to acquire sessions (`AcquireSession`), and one to release sessions (`ReleaseSession`).

5 Evaluation

We conducted an empirical evaluation of SCOPE, aiming to answer the following research questions.

1. How efficient is the SCOPE migration protocol? In particular, we investigate how fast SCOPE reacts to changes in workload and how long it takes to complete a migration.
2. What is the benefit of migrating sessions? In particular, we investigate how a suitable positioning of sessions improves the response time for clients.
3. How does SCOPE react to changes in workload and client mobility?

To answer the first two questions, we rely on a real distributed deployment of the SCOPE platform. To answer the third question, we implemented a simulator of the SCOPE migration strategies, which enables us to study larger scale and dynamic edge-to-cloud infrastructures.

Sec. 5.1 presents the setup we use for our experiments. Sec. 5.2, Sec. 5.3, Sec. 5.4 describe the experiments we performed to answer the three research questions, and discuss the results we measured.

5.1 Experiment setup

Deployment infrastructure.

To answer the first two research questions, we deployed SCOPE in a virtualized infrastructure hosted on a single physical machine. It consists of a client node generating the application load, one edge node, and one cloud node. Specifically, the nodes have the following configurations and geographical location.

- `client`. The client node generating the application load. It corresponds to the host machine, a MacBook Pro equipped with an Apple M3 processor (8 cores) and 24GB RAM, running macOS 26.2. It is located in a European region (EU1) anonymized for double-blind review.
- `edge`. An edge node located in the same region as the `client`. It is a virtual machine managed by UTM, equipped with 4 vCPUs (aarch64) and 4GB RAM.
- `cloud`. A cloud node located in a region far from the `client`. It is a virtual machine managed by UTM, equipped with 4 vCPUs (aarch64) and 4GB RAM. It is logically located in Northern Virginia (USA).

To simulate the geographical distance between the nodes, we introduced network delays using the Linux Traffic Control (`tc`) subsystem. We configured a fixed delay of 100ms to simulate the trans-Atlantic latency between EU1 and Northern Virginia (approx. 200ms round-trip time - RTT). Crucially, to reflect the topology where the `client` and `edge` are co-located while the `cloud` is remote, we applied the delay as follows: (i) the `cloud` node adds 100ms delay to *all* egress traffic, affecting communication towards both the `edge` node and the `client`; (ii) the `edge` node adds 100ms delay *selectively* only towards the `cloud` node, maintaining negligible latency towards the co-located `client`.

Use Cases. To evaluate the platform, we present two example use cases implemented in our functions: (1) a video storage application that provides simulated upload and download of content, and (2) a speech-to-text converter that simulates transcribing audio files.

The video storage application simulates the behavior of uploading and accessing large files. Varying the size of files emphasizes the benefits of accessing content from closer nodes that are connected with higher bandwidth. The function for uploading videos (`cdn-upload`) simulates a write-intensive task. It accepts a file (or uses a dummy payload - 5KB) and saves it into the session storage. Internally, it uses the `AcquireSession` function with specific options to allow offloading, while performing a busy-wait loop to simulate the processing time of a real upload (approximately 440ms + size-dependent delay). We used this function to test the platform's ability to handle resource-intensive write operations and to trigger migration based on I/O usage. The function for downloading videos (`cdn-download`) simulates a read-intensive task. It retrieves the requested file from the session storage using `AcquireSession`. Like the upload function, it includes a busy-wait loop to simulate the latency of serving a file. We used this function to verify that the platform can properly migrate sessions even when they are being actively read, maintaining data consistency.

The speech-to-text converter application represents a CPU-intensive workload. Executing this function close to the user (at the edge) is critical to reduce the perceived response time. The function (`speech-to-text`) receives an audio file, simulates the transcription process (using a busy-wait of 220ms), and stores the result in the session. It uses `AcquireSession` to lock the session during the processing and updates the session's CPU usage metrics (UpdateSessionResourcesUsage) to reflect the computational load. We used this function to test the platform's efficiency in managing complex, CPU-bound tasks and its ability to make offloading decisions based on CPU saturation.

Finally, we used two lightweight functions (`lf1`, `lf2`), one to measure platform overhead without SCOPE, and one to measure the migration benefits with SCOPE's session management but without application logic overhead. Function `lf1` does not interact with the session

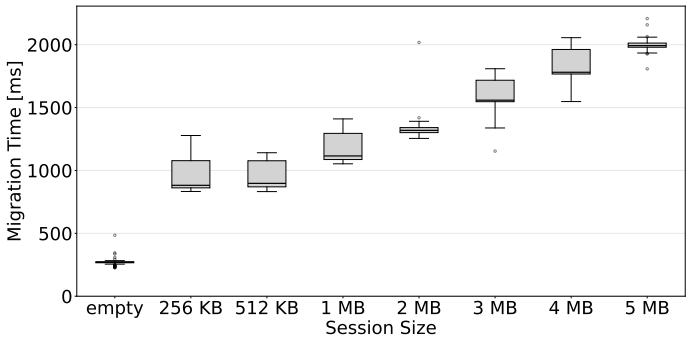


Fig. 4: Migration time with increasing session sizes.

management layer. We use it to measure the inherent latency and overhead of the FaaS platform (container cold starts, networking, routing), largely isolated from the costs of SCOPE's session acquisition and state management logic. Function `lf2` fully participates in SCOPE's session lifecycle and can trigger and respond to migration events, but does *not* execute any computational workload or I/O-intensive tasks. We used it to measure the benefits of session migration while isolating the effects from application-specific processing overhead.

5.2 Efficiency of the migration protocol

The first research question aims to evaluate the efficiency of the migration protocol, which directly impacts system availability. We define *migration time* as the total elapsed time from when the source node is triggered to migrate a session until the destination node confirms it has successfully received and stored the session state. In our experiments, this is measured as the execution time of a `POST` request to the `EU1` (edge) node, which initiates the transfer toward the `north-virginia` (cloud) node.

To perform this test, we used the `lf2` function to create sessions with varying state sizes—from empty sessions up to 5 MB. For each size, we triggered 200 migrations and measured the completion time. Fig. 4 presents these results as box-and-whisker plots, where each box represents the interquartile range (IQR). The experiment was conducted between the EU1 and North Virginia nodes with a simulated RTT of 200ms.

The results (Fig. 4) show that the average migration time for an empty session is about 270ms, which closely aligns with the baseline network RTT (200ms) (plus a small overhead for the HTTP interaction and session management logic). As the session size increases, the migration time scales linearly. Indeed, transferring 1 MB takes approximately 1.2s, and the time reaches about 2.0s for 5 MB. This corresponds to an average growth of roughly 200ms per additional MB, a rate primarily determined by the available network bandwidth between the nodes.

SCOPE minimizes the computing overhead of the migration process by ensuring there is always a warm function available to handle migration, thus avoiding cold start penalties. Moreover, the design of the Redis storage interface enables session data to be streamed simultaneously as it is being read from storage, and written into the new storage as soon as the data stream arrives, thus leaving the network as the only significant bottleneck.

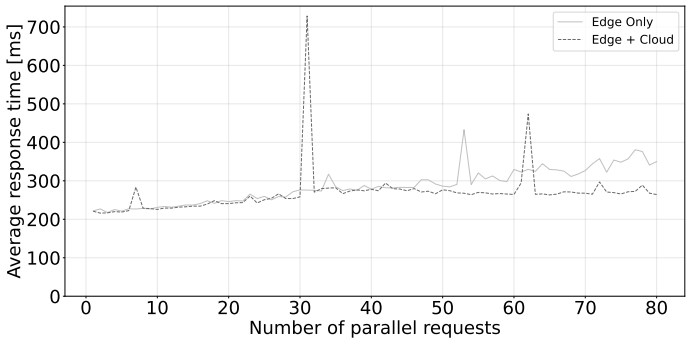


Fig. 5: Response time with increasing number of invocations.

5.3 Benefits of migrations

The second research question aims to evaluate the benefits of migration for system performance. First, we measure how effectively SCOPE handles growing rates of stateless requests by distributing loads to a better-equipped node, thereby optimizing throughput and maintaining system performance under high-traffic conditions. To examine how the average response time scales with increasing workload, we sent a progressively growing number of concurrent requests (from 1 to 80) to the `lf1` function, which allows us to focus on the cost of handling requests without additional application-specific computational load.

We measured the average response time under two configurations. In the first configuration, all traffic was forwarded exclusively to the `EU1` (edge) node. In the second configuration, we implemented a load-balancing strategy where traffic exceeding the edge node's capacity was redirected. The first 42 concurrent requests were handled by the edge node, while any additional requests were forwarded to the `cloud` node.

Fig. 5 illustrates the average response time (Y-axis) as the number of parallel requests (X-axis) increases for both configurations. Both scenarios start with a baseline response time of approximately 220ms, which is influenced by the network configuration and the minimum overhead of the serverless platform. In the edge-only scenario (light gray line), the system maintains this baseline until it reaches approximately 42 parallel requests. Beyond this threshold, the edge node begins to saturate, leading to a steady increase in average response time, which reaches nearly 350ms at 80 concurrent requests. In the edge-cloud scenario (dark gray line), the system identifies the saturation point and redirects the overflow traffic to the `cloud` node. As a result, the average response time increases much more gradually, settling around 265ms at the maximum tested load of 80 requests. Although the cloud responses are slightly slower due to the additional network distance (averaging ≈310ms individually), the overall system performance remains significantly more stable compared to allowing the edge node to reach total saturation. This emphasizes the importance of proper load balancing within the system: when nearby nodes are overloaded, forwarding parts of the requests to remote but more available nodes prevents performance degradation and keeps response times within acceptable limits.

Having proved the benefits of load balancing, we now evaluate the improvements in response times after:

1. The migration of a session to a node closer to the client, when such a node exists and has enough resources to accommodate the session.

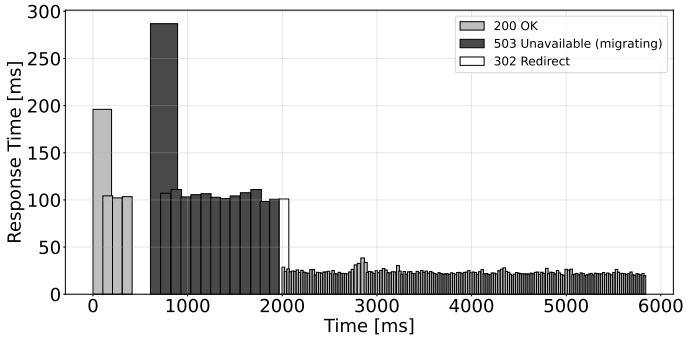


Fig. 6: Response times during a migration process.

2. The migration of pre-existing active sessions from an edge node to a cloud node, triggered when existing sessions with established state generate simultaneous load that exceeds the edge node's limits, requiring the transfer of session state during migration.
3. The redirection of new session requests toward a remote cloud node, executed when the edge node becomes saturated during a surge of new traffic, allowing the system to scale gracefully by placing new sessions directly on the cloud without the overhead of migrating existing state.

By simulating these three scenarios, we aim to demonstrate that SCOPE effectively manages session placement across heterogeneous infrastructures, improving response times and system utilization through context-aware migration decisions that adapt to dynamic load conditions.

The goal of the first experiment is to quantify the improvements in response times from a client's perspective by demonstrating the benefits of moving a computation closer to the user through the onloading process.

To evaluate this, a client repeatedly sends the same `lf2` function request to the `cloud` node, sequentially for 6 seconds. The request involves a 2MB session state in read-write mode, ensuring that data consistency is maintained during the migration.

Initially, the client contacts the `cloud` node. After exactly four requests, we manually trigger a migration from the `cloud` node to the `EU1` (edge) node. During the migration protocol, the session becomes temporarily unavailable as the state is transferred. This results in the client receiving a series of `503 Session not available` responses (13 consecutive failures in our test), followed by a single `302 Redirect` that points the client to the new `edge` node.

Fig. 6 illustrates the client's perspective on the effects of migration. The X-axis represents time, while the Y-axis shows the observed response times. The results highlight the immediate performance gains: the response time drops from an average of 126ms (cloud) to just 21ms (edge) after the migration is completed, proving that SCOPE efficiently moved sessions closer to the client. The short period of session unavailability is represented by the 503 error responses. For a 2MB session, the migration causes a total disruption of approximately 1.5 seconds (calculated as the sum of failure durations). As seen in Fig. 4, the transfer time for 2MB is roughly 1.3s, which matches the observed window of unavailability plus small protocol overheads. This demonstrates the effectiveness of SCOPE's onloading strategy, as the 1.5 seconds only of disruptions are compensated by the drastic and permanent improvement in response times for all subsequent requests.

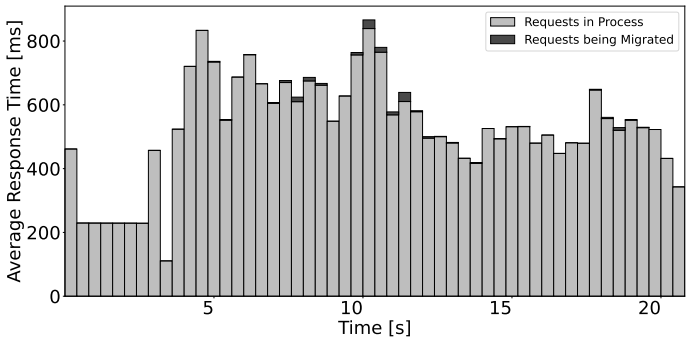


Fig. 7: Increase in client traffic and average response time during session migration.

While the previous experiment focused on stateless function invocations, we performed additional tests to evaluate the benefits of relocating sessions toward nodes with higher availability when the edge node reaches saturation. We consider the `EU1` (edge) node and the `north-virginia` (cloud) node, utilizing the functions described in Sec. 5.1: `speech-to-text` (CPU-intensive), `cdn-upload`, and `cdn-download` (I/O-intensive). At the start of the experiment, the edge node has 60 pre-registered sessions, each containing 2 MB of state data. Clients are configured to submit requests at their maximum rate, with each client targeting a different session. For the first 3.2 seconds of the test, only one client generates traffic for a single session, establishing a baseline response time of approximately 230ms. Subsequently, additional clients start generating traffic in a staggered fashion (one every 15ms) until all 60 sessions are active. Half of these sessions trigger the CPU-bound `speech-to-text` function, while the rest are divided between upload and download I/O-intensive tasks. Fig. 7 depicts the system's response to this traffic peak from the perspective of the clients. The X-axis represents the experiment timeline (in seconds), while the Y-axis shows the average response time (in milliseconds). The vertical bars indicate the average response time partitioned within each 400ms time slice. As more clients connect, the edge node quickly reaches its capacity limits, causing the average response time to peak at approximately 866ms. At this point, SCOPE's offloading monitor detects the saturation and triggers the migration of sessions toward the cloud node. In the graph, each bar highlights two contributions: the light gray segment (`Requests in Process`) represents the successful execution time, while the dark gray segment (`Requests being Migrated`) accounts for the session unavailability period during state transfer. As the migration process progresses, we observe a temporary increase in total response time due to the 2MB session state being moved over the 200ms RTT link. However, as more sessions are successfully relocated to the cloud, the overhead on the edge node decreases. The dark gray "migration penalty" gradually vanishes as sessions stabilize on the new host. Ultimately, the system reaches a new equilibrium with an average response time of approximately 520ms. While this is higher than the initial edge-only baseline (due to the inherent latency of the North Virginia cloud node), it remains stable and significantly lower than the peak saturation levels observed before the offloading took effect, demonstrating how reactive migration prevents system collapse under heavy, stateful workloads.

We conducted a final experiment to evaluate a scenario where traffic is generated by adding new sessions to the system rather than increasing traffic to existing ones. This experiment is separate from the previous one to evaluate how does SCOPE handle proactive redirection. In this test, 60 concurrent clients are introduced with a 15ms staggering, each requesting a unique session and executing high-load functions (`speech-to-text`,

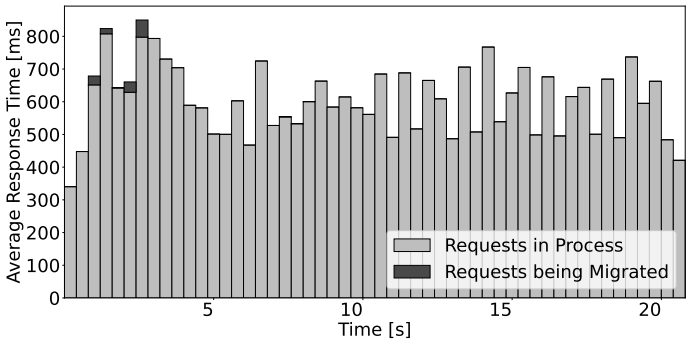


Fig. 8: Increase in active clients and response time, including the redirection overhead for new sessions on a saturated node.

`cdn-upload/download`). Fig. 8 shows that as the `EU1` (edge) node approach its saturation point, SCOPE detects the load and proactively redirects any *new* session-creation requests directly to the `north-virginia` (cloud) node using HTTP 302 redirects. In this scenario, the `edge` node is not burdened with migrating existing states, as sessions are immediately redirected and generated directly on the cloud host. This prevents the "stop-and-move" overhead seen in the reactive migration experiment (Exp A). As shown in Fig. 8, while there is an increase in response time as more clients connect—rising from an initial 340ms to a peak of approximately 850ms (Slice 3)—the transition is considerably smoother. The dark gray bars in the graph (peaking at 52ms in Slice 6) represent the small redirection overhead incurred only during the initial session establishment handshake. Unlike the previous experiment, there is no performance penalty for transferring 2MB of active state, as the session is assigned directly to the node capable of accepting it without interrupting existing executions. After this brief setup phase, the average response time stabilizes between 500ms and 750ms, reflecting the steady-state execution on the higher-latency cloud node.

This demonstrates that SCOPE handles dynamic workloads optimally by prioritizing placement decisions at session creation time, reserving reactive migration only for unexpected load shifts in active sessions. In this experiment, the system can react earlier to incoming load because redirecting new session requests incurs minimal overhead compared to migrating existing sessions with state (previous experiment). Indeed, the framework adopt a more aggressive, preemptive policy in this case, as it can begin redirecting new traffic before reaching full saturation, preventing performance degradation. In contrast, the previous experiment must wait for actual saturation before triggering costly state migrations, as premature migration would waste resources.

Summing up, these results show the benefits of SCOPE's multi-tiered strategy, that is, SCOPE migration saves the system from collapse under unexpected load, while proactive placement ensures optimal performance for new workloads.

5.4 Adaptation to changes

The third research question aims to explore the behavior of SCOPE in response to changes in the operating conditions. Specifically, we aim to demonstrate that the session migration protocol: (1) maintains sessions as close as possible to their clients, provided that nearby nodes have enough resources; (2) responds to client mobility to preserve low response time.

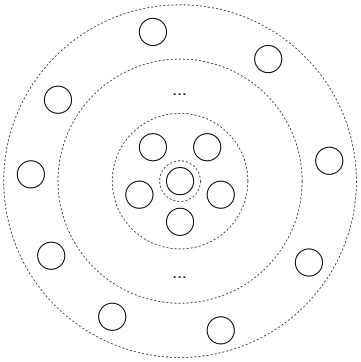
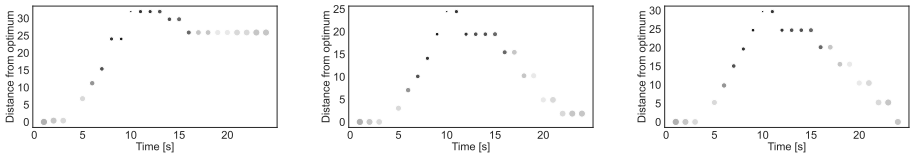


Fig. 9: Topology of the network used in the experiments.



(a) Network with 10 nodes. (b) Network with 100 nodes. (c) Network with 1000 nodes.

Fig. 10: Deviation between current and optimal position of sessions while varying resource usage.

For these experiments, we simulated networks with a variable number of nodes, generating traffic within them and implementing the SCOPE migration protocol in response to the events generated by the traffic. The simulated networks consist of nodes ($n_i \in N$) that are positioned on a Cartesian plane using geographical coordinates. The resources of each node are modeled using a single scalar number (r_i). Fig. 9 illustrates the network topology. Nodes are arranged in concentric circles, with a root node located in the center. As we move outward, each successive circle contains an increasing number of nodes, evenly distributed along the circumference of the circle. The nodes in each circle are connected to the nearest nodes in the immediately preceding circle, creating a hierarchical and well-connected network. The root node is the one with the highest resource value r_i, and resource values gradually decrease in the successive circles as the distance from the center increases.

To illustrate that the session migration protocol keeps sessions as close as possible to their clients, we simulated five geographically positioned clients ($c_j \in C$), interacting with the system to access their own sessions. For each client c_j, we consider a single session s_j stored in one of the node of the network and consuming a resource w_j on the node where it is stored. With each interaction between a client and the network, the session resource may change, potentially causing SCOPE to migrate the sessions.

Specifically, our simulated traffic begins by assigning each client c_j a session s_j with $w_j = 1$. Then, we perform an increasing phase, in which we simulated a series of invocations from each client c_j that increases the value w_j of s_j. Finally, we perform a decreasing phase,

in which we simulated a series of invocations from each client c_j that decreases the value w_j of s_j.

We simulated networks of different size (10, 100, and 1000 nodes), and we triggered the SCOPE migration protocol to analyze how effectively SCOPE keeps each session close to the client that owns it.

To evaluate the results, for each client c_i we denote its *optimal* node as the node that is the geographically closest to the client. In practice, SCOPE would always store a client's session in its optimal node if the choice of node was based only on proximity. However, since a node must have sufficient resources to assign a session to it, the client's session is not always located in the optimal node. We define the *client's distance from optimum* (d_j) for client c_j as the geographical distance between the node where s_j is currently located and c_i's optimal node. To evaluate the overall network, we define the *distance from optimum* as $D = \sum_{c_i \in C} d_i$

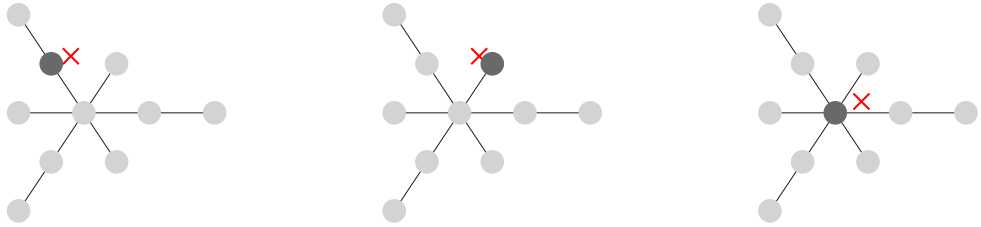
Similarly, we define for each node of the network n_i its *nodes available resource* ar_i as the actual quantity of resource available for node n_i Considering S_i as the set of sessions stored in n_i, the available resource of node n_i is $ar_i = r_i - \sum_{s_x \in S_i} w_x$. We compute the *network available resource* AR as the sum of the *nodes available resources* $AR = \sum_{n_i \in N} ar_i$

Fig. 10 shows the results of our analysis. The X-axis represents the time at which the clients performed their requests, while the Y-axis shows the *distance from optimum* D at different points in time. The size and brightness of the dots reflect the amount of *available resources* AR at each time point: larger and brighter dots indicate higher availability of resources, while smaller and darker dots signal that the network resources are more utilized.

At the beginning of the experiment, all nodes have available resources, and the distance from optimum is kept minimal. This shows that the algorithm implemented by SCOPE allocates sessions effectively and favors proximity to the client when resources are available. When the network resources are more utilized, the distance to the optimum increases as the nearest nodes are already occupied, forcing SCOPE to move sessions to more distant and larger nodes. Later, we progressively reduce the resources used by each session. Ideally, session should return to the optimal node. However, SCOPE adopts a decentralized approach to migrate sessions, where nodes have partial knowledge of the status of the network. In the 10-node network, the simulations reveal that SCOPE struggles to bring the sessions back closer to the optimal. This is due to the network's topology, where nodes are at most two levels away from the root node and have only cumulative knowledge of their child nodes by design. As soon as a session is moved to another child node, it can no longer retrieve information about nodes in another subtree. However, the problem becomes less and less critical when the number of nodes in the network increases, showing that SCOPE successfully moves sessions closer to the optimum when possible, i.e. when nearby nodes have enough resources. In conclusion, the distributed algorithm for session placement adopted by SCOPE successfully places sessions as close as possible to clients, even with networks of modest size.

To test SCOPE' ability to respond to client mobility and keep response times low, we simulated a mobile client that regularly updates its geolocation to reflect movement. In this experiment, we used a network of 10 nodes with a client that owns a session and frequently changes its position. The client continuously sends position updates to the system. As the client moves, the system dynamically migrates its session to maintain minimal distance between the client and the node hosting the session.

We simulated the client's movements by changing its geographic coordinates and triggered the migration protocol based on the client's displacements. Fig. 11 illustrates examples of optimal nodes selected during our simulations according to the client's position. The circles represent the network nodes that were positioned based on their geographic coordinates, while the red "X" marks the client's location. For each network, the dark gray node highlights



(a) (b) (c)

Fig. 11: Examples of optimal node chosen (dark gray circle) based on the position of the client (red X).

where the session will be migrated to, according to the client's movement. In conclusion, SCOPE effectively updates the location of sessions in response to clients' mobility.

6 Conclusions

In this work, we introduced the SCOPE platform to enable stateful serverless computing across the edge-to-cloud continuum. SCOPE combines the convenience of the serverless programming paradigm with the performance benefits of edge computing. It leverages the concept of sessions as units of state accessed by clients and automatically migrates them across the edge-to-cloud continuum to optimize performance based on data locality and resource availability.

Our evaluation demonstrates the benefits of SCOPE and its ability to adapt to dynamic conditions, such as varying workloads and client mobility. Encouraged by these results, we plan to extend the platform by generalizing its programming model to support sessions shared by groups of clients and enable access to replicated data. We also plan to study alternative scheduling strategies, to prioritize data locality, potentially offloading computation to nodes holding the data rather than solely relying on migration.

Acknowledgments

Generative AI tools were used solely for language editing; all content was reviewed and verified by the authors, who assume full responsibility for the manuscript.

The authors declare that no funding was received for this research.

References

[1] Shi, W., Dustdar, S.: The promise of edge computing. Computer **49**(5), 78–81 (2016) https://doi.org/10.1109/MC.2016.145

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53**, 50–58 (2010) https://doi.org/10.1145/1721654.1721672

[3] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE Internet of Things Journal **3**(5), 637–646 (2016) https://doi.org/10.1109/JIOT.2016.2579198

[4] Ren, J., Zhang, D., He, S., Zhang, Y., Li, T.: A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. ACM Comput. Surv. **52**(6) (2019) https://doi.org/10.1145/3362031

[5] Xie, R., Tang, Q., Qiao, S., Zhu, H., Yu, F.R., Huang, T.: When serverless computing meets edge computing: Architecture, challenges, and open issues. IEEE Wireless Communications **28**(5), 126–133 (2021) https://doi.org/10.1109/MWC.001.2000466

[6] Nardelli, M., Russo Russo, G.: Function offloading and data migration for stateful serverless edge computing. In: Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering. ICPE '24, pp. 247–257. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3629526.3649293

[7] Kuhlenkamp, J., Werner, S., Borges, M.C., El Tal, K., Tai, S.: An evaluation of faas platforms as a foundation for serverless big data processing. In: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing. UCC'19, pp. 1–9. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3344341.3368796

[8] Gadepalli, P.K., McBride, S., Peach, G., Cherkasova, L., Parmer, G.: Sledge: a serverless-first, light-weight wasm runtime for the edge. In: Proceedings of the 21st International Middleware Conference. Middleware '20, pp. 265–279. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3423211.3425680

[9] Pfandzelter, T., Bermbach, D.: tinyfaas: A lightweight faas platform for edge environments. In: IEEE International Conference on Fog Computing. ICFC 2020, pp. 17–24. IEEE, ??? (2020). https://doi.org/10.1109/ICFC49376.2020.00011

[10] Shillaker, S., Pietzuch, P.: Faasm: Lightweight isolation for efficient stateful serverless computing. In: USENIX Annual Technical Conference. USENIX ATC 20. USENIX Association, ??? (2020)

[11] Varda, K.: Introducing Cloudflare Workers: Run JavaScript Service Workers at the Edge. https://blog.cloudflare.com/introducing-cloudflare-workers/. Accessed: 2026-02-25 (2017)

[12] Varda, K.: Workers Durable Objects Beta: A New Approach to Stateful Serverless. https://blog.cloudflare.com/introducing-workers-durable-objects. Accessed: 2026-02-25 (2020)

[13] Amazon: AWS IoT Greengrass. https://aws.amazon.com/greengrass/. Accessed:

2026-02-25 (2024)

[14] Russo, G.R., Mannucci, T., Cardellini, V., Presti, F.L.: Serverledge: Decentralized function-as-a-service for the edge-cloud continuum. In: IEEE International Conference on Pervasive Computing and Communications. PerCom 2023, pp. 131–140. IEEE, ??? (2023). https://doi.org/10.1109/PERCOM56429.2023.10099372

[15] Pfandzelter, T., Bermbach, D.: Enoki: Stateful distributed faas from edge to cloud. In: Proceedings of the 2nd International Workshop on Middleware for the Edge. MiddleWEdge '23, pp. 19–24. ACM, New York, NY, USA (2023). https://doi.org/10.1145/3630180.3631203

[16] Vahabi, S., Righetti, F., Vallati, C., Tonellotto, N.: 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. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3603166.3632240

[17] Cicconetti, C., Conti, M., Passarella, A.: A decentralized framework for serverless edge computing in the internet of things. IEEE Transactions on Network and Service Management **18**(2), 2166–2180 (2021) https://doi.org/10.1109/TNSM.2020.3023305

[18] Baresi, L., Hu, D.Y.X., Quattrocchi, G., Terracciano, L.: Neptune: A comprehensive framework for managing serverless functions at the edge. ACM Trans. Auton. Adapt. Syst. **19**(1) (2024) https://doi.org/10.1145/3634750

[19] Guo, S., Xiao, B., Yang, Y., Yang, Y.: Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In: IEEE International Conference on Computer Communications. INFOCOM 2016, pp. 1–9. IEEE Press, ??? (2016). https://doi.org/10.1109/INFOCOM.2016.7524497

[20] Raith, P., Rausch, T., Dustdar, S., Rossi, F., Cardellini, V., Ranjan, R.: Mobility-aware serverless function adaptations across the edge-cloud continuum. In: International Conference on Utility and Cloud Computing. UCC 2022, pp. 123–132 (2022). https://doi.org/10.1109/UCC56403.2022.00023

[21] Lai, P., He, Q., Grundy, J., Chen, F., Abdelrazek, M., Hosking, J., Yang, Y.: Cost-effective app user allocation in an edge computing environment. IEEE Transactions on Cloud Computing **10**(3), 1701–1713 (2022) https://doi.org/10.1109/TCC.2020.3001570

[22] Ma, Z., Zhang, S., Chen, Z., Han, T., Qian, Z., Xiao, M., Chen, N., Wu, J., Lu, S.: Towards revenue-driven multi-user online task offloading in edge computing. IEEE Trans. Parallel Distrib. Syst. **33**(5), 1185–1198 (2022) https://doi.org/10.1109/TPDS.2021.3105325

[23] Tan, H., Han, Z., Li, X.-Y., Lau, F.C.M.: Online job dispatching and scheduling in edge-clouds. In: IEEE Conference on Computer Communications. INFOCOM 2017, pp. 1–9 (2017). https://doi.org/10.1109/INFOCOM.2017.8057116

[24] Bellendorf, J., Mann, Z.A.: Classification of optimization problems in fog computing. Future Gener. Comput. Syst. **107**(C), 158–176 (2020) https://doi.org/10.1016/j.future.2020.01.036

[25] Bernstein, P., Bykov, S., Geller, A., Kliot, G., Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability. MSRTR2014 **41** (2014)

[26] Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. SIGCOMM Comput. Commun. Rev. **34**(4), 15–26 (2004) https://doi.org/10.1145/1030194.1015471

[27] Chen, Y., Wang, X., Song, X., Lua, E.K., Shi, C., Zhao, X., Deng, B., Li, X.: Phoenix: Towards an accurate, practical and decentralized network coordinate system. In: International IFIP-TC 6 Networking Conference. NETWORKING 2009, pp. 313–325 (2009). Springer

[28] Kleppmann, M.: Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, ??? (2017)