Pervasive and Mobile Computing 117 (2026) 102175

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Design and implementation of a platform for stateful agents at the edge

Claudio Cicconetti [a,*], Emanuele Carlini [a], Chen Chen [b], Roman Kolcun [b], Richard Mortier [b]

[a] National Research Council, Pisa, Italy
[b] University of Cambridge, Cambridge, UK

ARTICLE INFO

Keywords:
Serverless computing
Edge computing
Stateful FaaS

ABSTRACT

Edge–cloud computing infrastructures are increasingly widespread as they combine the flexibility of cloud-native development tools with the performance and security of distributed computing environments. Function-as-a-Service has emerged as a powerful abstraction that overcomes the limitations of a micro-service architecture. However, it generally does not support stateful functions, making it unsuitable for many practical applications in, e.g., Internet of Things (IoT) and real-time analytics. In this paper, we explore a novel paradigm, based on stateful asynchronous agents, that goes beyond traditional serverless computing. We focus on several key technical aspects: programming model, deployment procedures, design of a flexible compute node, and state management. We illustrate our paradigm using the *EDGELESS* platform as a concrete implementation of this stateful agents' pattern. We report proof-of-concept experiment results obtained in a testbed with heterogeneous resource-constrained edge nodes that showcase some distinguishing features of our platform: scalable management of lightweight function instances, the advantage of keeping the state local at function instances, and delegated orchestration to enable a third-party agent to make migration decisions in a group of local nodes.

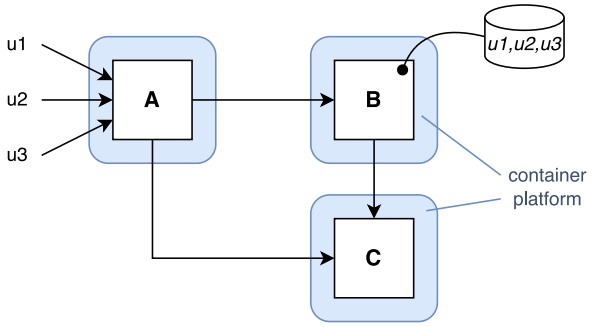
1. Introduction

In recent years, edge computing has emerged as an alternative environment for executing Internet of Things (IoT) applications and services due to several advantages [1]: reduced long-haul network traffic, proximity of data to processing units, easier enforcement of privacy rules, and reduced costs thanks to (partial) use of customer computing infrastructure, particularly for specialized tasks. It is now a mature option offered by all major cloud service providers. However, despite its appeals, edge computing still struggles to define its identity independently from the cloud that fathered it [2]. This is due to the much broader spectrum of application requirements and deployment options covered by edge computing compared to the far more homogeneous cloud. The state of the art is to enable some form of *computational offloading*, delegating some of the processing burden from the cloud to edge nodes, often conveniently realized in a software architecture based on containerized micro-services [3].

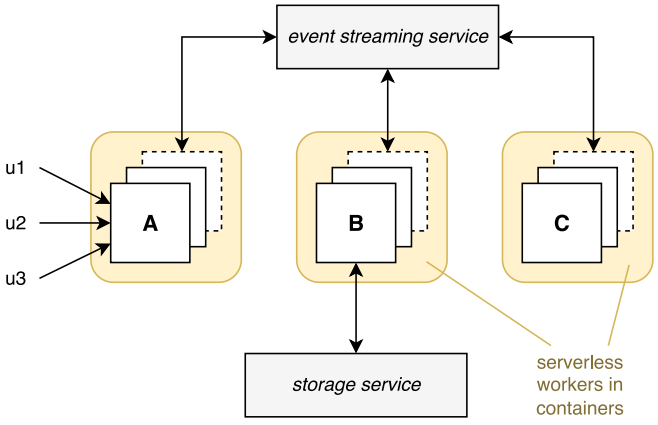
Serverless computing [4] has been proposed as a deployment option for applications on edge nodes, e.g., to enable digital twins [5]. It allows application developers to define services by composing elementary stateless functions that interact through a function

* Corresponding author.
E-mail addresses: c.cicconetti@iit.cnr.it (C. Cicconetti), emanuele.carlini@isti.cnr.it (E. Carlini), cc2181@cam.ac.uk (C. Chen), rk647@cam.ac.uk (R. Kolcun), rmm1002@cam.ac.uk (R. Mortier).

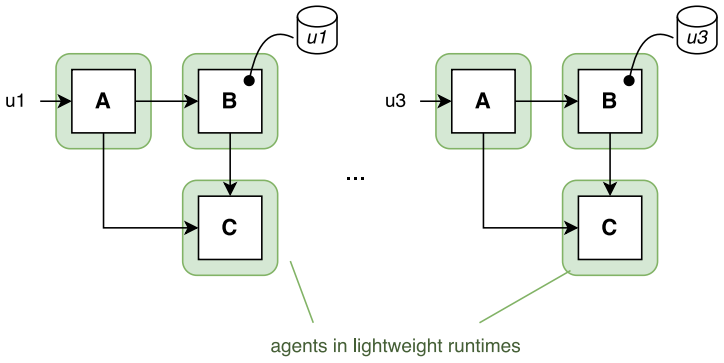
https://doi.org/10.1016/j.pmcj.2026.102175
Received 9 June 2025; Received in revised form 10 January 2026; Accepted 18 January 2026
Available online 28 January 2026



(a) Micro-services running in containers and interacting via dedicated interfaces.



(b) FaaS composition realized through third-party services for event streaming and state management.



(c) Stateful agent approach, where each user is assigned their own set of instances.

Fig. 1. Different models for the composition of an example workflow consisting of three logical functions, *A*, *B*, and *C*, with *B* stateful, accessed by three users.

invocation pattern, Function as a Service (FaaS). Access to application state in the cloud relies on ancillary services such as storage platforms and in-memory key–value stores, which can be cumbersome to provide efficiently in edge computing environments due to limited resources (compute, memory, network, storage) plus the reduced reliability of edge nodes [6]. These challenges make it difficult to realize the advantages of serverless computing platforms at the edge [7].

1.1. A high-level comparison of architectural patterns

To highlight the limitations of currently state-of-the-art models and provide motivation for our approach, we compare three architectural patterns for executing a simple workflow in Fig. 1: micro-service vs. serverless computing vs. stateful agents. The

Table 1
WebAssembly vs. Docker container run-times for a simple function.

Metric	WebAssembly	CT/Rust	CT/Python
Image size (MB)	0.017	86	1060
Memory (MB)	1	20	25
Start-up (ms)	1.5	150	300

figure depicts a *workflow*[1] comprising three *tasks*, A, B, C, where A is the entry-point of external user triggers, B depends on input from A, and C depends on input from both A and B. The example workflow forms a Directed Acyclic Graph (DAG), a common model for both micro-service and serverless applications [9], and we assume that B's application logic depends on some state associated with each user (or session).

The **micro-service** approach is illustrated in Fig. 1(a), where A, B, and C run as containers in an environment such as Kubernetes (K8s) [10] that also manages horizontal and vertical scalability, and user state is maintained in the container running task B. This approach is the state-of-the-art in public and private clouds, but when investigated for edge computing environments, it displays the following limitations: (*i*) edge nodes have limited computation resources and connectivity compared to cloud nodes, which makes container migration relatively heavier and so limits opportunities for dynamic rebalancing workload across cluster nodes; and (*ii*) the strong consistency model of Kubernetes coupled with limited edge resources may increase latency and reduce availability [11].

The **serverless computing** option is illustrated in Fig. 1(b), where the key difference from a micro-service architecture is that the tasks, or *functions* in this context, are all stateless. This makes autoscaling, where the number of replicas is matched to the instantaneous demand, easier as there is no state to be migrated [12]. There are two main means through which the functions interact with one another: function invocation ($A \rightarrow B$, $A \rightarrow C$, etc.), which can be done via an event streaming service, and state access for only those functions that require accessing user state, which can be done via a storage service. Readily available in a cloud datacenter, these ancillary services can be a single point of failure and performance bottleneck at the edge, where the nodes have fewer resources and lower availability than corresponding cloud nodes.

Our contribution is to propose a third way, **stateful agents**, illustrated in Fig. 1(c). In this model, each user is assigned a dedicated workflow, an instance of the generic $A/B/C$ workflow, decomposed into function instances, each running in a separate virtualization abstraction so that each B instance will contain only the state of its user. Workflow instances do not interact with one another, following the principle of user isolation. The agents, or *actors* [13], interact directly with one another as in a micro-service architecture but using a homogeneous API based on events to enable functions to be developed as elementary pieces of application logic as in the FaaS paradigm in serverless computing. This design eliminates the need for centralized state stores or event brokers, reducing latency and improving fault isolation. Furthermore, as each function instance addresses the need of a single user, there is less pressure to replicate it than in a micro-service or serverless platform, which greatly simplifies the orchestration of resources in a cluster of nodes. However, the number of stateful agent instances can be much higher than the number of containers with the other two options. We thus pivoted towards the use of lightweight virtualization, significantly reducing the overhead of function instance lifecycle management and allowing our approach to scale to a large number of concurrent agents. For example, WebAssembly [14] has been promoted in this context in several studies [15–17] and Table 1 gives a basic performance comparison between WebAssembly and Docker containers obtained with our platform. The data were obtained by 100 consecutive deployments of a simple function that doubles its numerical input, implemented in Rust and deployed as both WebAssembly and a Docker container based on the `ubuntu:22.04` image, and implemented in Python and deployed using the `python:3` Docker image. Experiments have been repeated on both a high-end Intel server and an NVIDIA Orin NX device with similar results.

In short, the stateful agents pattern we advocate aims to:

- Support stateful tasks, e.g., micro-service architectures, with reduced overhead because each function only keeps the state of a single user/session.
- Enable streamlined orchestration, as in serverless computing, with a finer granularity because functions execute in lightweight virtualization run-time environments.

We argue that adopting the proposed pattern offers advantages not only for infrastructure operators by improving performance and reducing overhead, but also from the perspective of the effort required to develop services. In a typical micro-service architecture, scalability concerns must be addressed within each component, or at least within those most likely to become bottlenecks. This places a significant burden on developers, preventing them from focusing solely on the internal logic of individual components. Although serverless computing platforms can alleviate some of this burden, they often reintroduce complexity in more subtle ways, particularly due to the challenges of managing user or session state. Moreover, neither container orchestration systems nor serverless platforms were originally designed for concurrent execution across multiple edge–cloud infrastructures. As a result, custom, aftermarket solutions are often required to support such deployments. In contrast, with stateful agents, developers can concentrate on implementing and testing elementary functions in isolation, both stateless and stateful. These functions are then dynamically composed at runtime within a heterogeneous and distributed infrastructure, significantly reducing both engineering complexity and scalability risks.

[1] In the paper we call the user application a *workflow* to emphasize that, in general, it consists of multiple tasks interacting with one another through some inter-process communication interface [8].

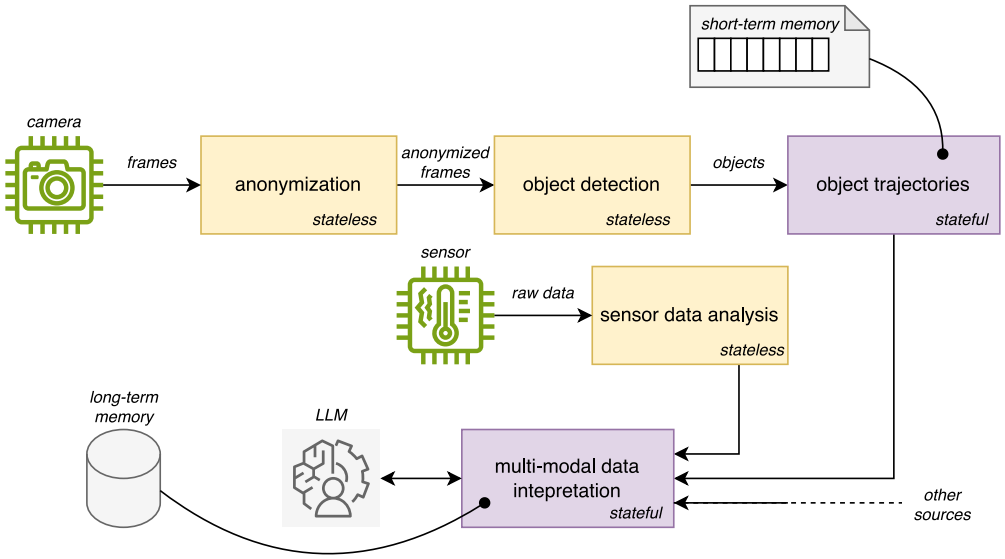


Fig. 2. Intelligent traffic management use case example.

1.2. Work contribution and methodology

The main contribution of this work is the design and analysis of a modular platform, with clear yet flexible specifications, to lay the foundations for supporting future research in the area of stateless/stateful services in the edge–cloud continuum.

To corroborate our design choices, we have run experiments with a reference implementation of such a platform developed in the EDGELESS project [18] and available as open-source software with a permissive license on GitHub [19]. The experiments have been run on a research infrastructure at CNR, the hosting institution of authors C. Cicconetti and E. Carlini, and the results obtained are publicly available, together with the scripts to generate all the plots in this paper in [20]. Furthermore, to ensure reproducibility of the results by third parties, the repository also contains the instructions to set up the system under study and the scripts to run the experiments. In particular, the results will cover three key aspects:

- Experiment 007 in [20] (Section 2.3): efficient migration of stateful agents running in WebAssembly vs. Docker containers.
- Experiment 008 in [20] (Section 2.4): low overhead in accessing local vs. remote state, with stateful vs. stateless functions.
- Experiment 004 in [20] (Section 3.2): opportunity to plug in an external decision maker to orchestrate resources in a group of edge nodes.

Furthermore, in Section 3.3, we compare EDGELESS with wasmCloud, which is a state-of-the-art framework to develop workflows of reusable WebAssembly components and deploy them in the edge–cloud continuum. The comparison is both qualitative, in terms of design and architectural choices, and quantitative, through experiments whose scripts and results can be found as Experiment 012 in [20].

1.3. A motivating practical example

To ground our discussion, we illustrate the key concepts through a domain-specific use case in intelligent traffic management, which serves as an anchor for understanding the proposed architecture and its contributions.

The use case is illustrated with the help of Fig. 2 and envisions cameras and roadside sensors at intersections in a smart city that capture video and telemetry data. Edge nodes run stateful FaaS agents that process these streams in real time to anonymize video frames and detect objects, which are stateless operations that should be performed as close as possible to the raw data source for technical (low latency, reduced network traffic) and non-technical reasons (privacy requirements). The objects detected are sent to agents using lightweight computer vision models to extract and maintain short-term trajectories of vehicles, cyclists, and pedestrians. These trajectories form the local state, enabling predictions such as potential collisions, jaywalking events, or congestion build-up. An LLM, collecting information from multiple object trajectories, augmented with pre-processed sensor data, is then used to interpret multimodal data, which is stored persistently.

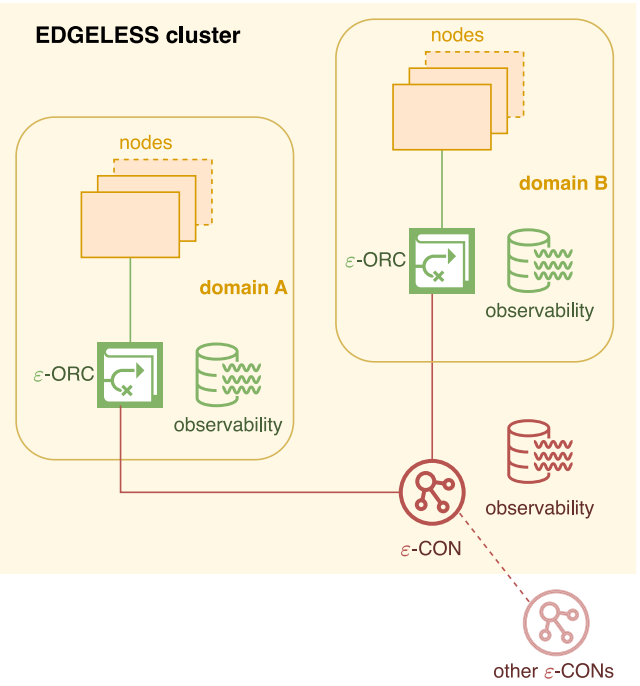


Fig. 3. High-level architecture of an EDGELESS cluster showcased by means of an example deployment where an ε-CON manages two orchestration domains, each managed, in turn, by an ε-ORC. An orchestration domain consists of a set of nodes that execute the EDGELESS applications. The orchestration domains and the cluster as a whole have their separate observability layers, which maintain an up-to-date status of the run-time system performance.

1.4. Paper structure

The rest of this paper is structured as follows. In Section 2, we highlight the key design issues of a platform supporting stateful agents, for which the EDGELESS concrete implementation is illustrated in Section 3. We discuss related work in Section 4. Finally, in Section 5 we conclude and discuss future work directions.

Some concepts described in this work have been anticipated in two previous short papers about EDGELESS: the project's objectives, high-level architecture, and roles of the main orchestration components in [21]; and the fundamental characteristics of the programming model and node run-times in [22].

2. Platform design

We next illustrate the high-level design choices of the EDGELESS platform to realize the stateful agents' pattern. At the time of writing, EDGELESS is a work-in-progress initiative. We describe here some fundamental properties and core architecture elements that are distinctive characteristics and are expected to remain stable in the near-medium term. The description in this work closely matches the terminology currently adopted in the open-source code base [19] as of release `v1.0.0`, but is subject to change in future releases.

Fig. 3 depicts the high-level architecture of an EDGELESS cluster. To improve scalability, we divide the cluster into *orchestration domains*, each comprising nodes managed by an EDGELESS orchestrator (ε-ORC), which oversees the domain formation and intermediates with the cluster-level controller (ε-CON). This two-tier structure follows the results obtained in prior works related to efficient orchestration of edge–cloud systems such as Serverledge [23] and Oakestra [24]. Serverledge demonstrated that decentralized control is well suited to computation offloading in geographically distributed edge computing infrastructures, as it can decrease response times and increase throughput by exploiting resources on local nodes more efficiently. We implement this pattern via ε-ORCs, which enable local operations within their respective orchestration domains. Oakestra envisioned significantly reduced management complexity and overhead through a delegated scheduling mechanism with coarse-grained resource allocation choices made by the root and fine-grained placement of tasks by leaves. We adopt this concept and fuse it with workflows managed coarsely by the ε-CON while ε-ORCs allocate tasks, called function and resource instances (Section 2.1). Observability, i.e., the systematic collection of run-time measures from nodes, is described in more detail later in this section. It adheres to the same hierarchical structure and is divided between the domain level, storing fine-grained and detailed metrics, and the cluster level, keeping a partial and aggregated view of the overall system.

We now present details of the programming model (Section 2.1), the deployment model (Section 2.2), the inner structure of nodes (Section 2.3), and state management (Section 2.4), key contributions of EDGELESS.

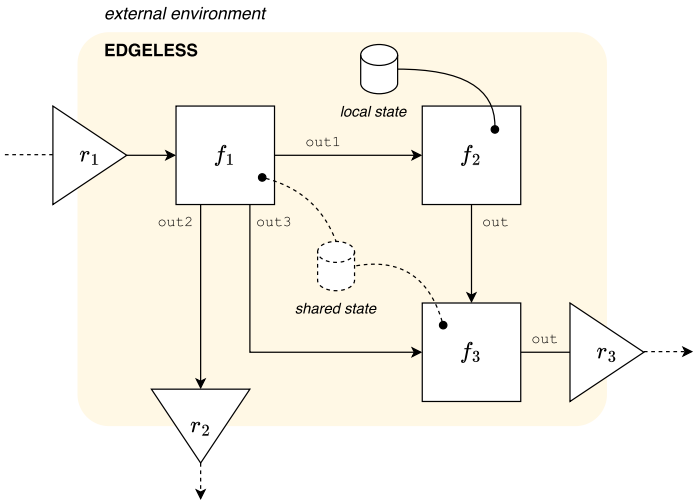


Fig. 4. An EDGELESS workflow comprising three functions (squares, f_i) that process events, and three resources (triangles, r_i) that interact with the external environment. Function f_2 has a local state that is private to that instance, while f_1 and f_3 both access shared state that is private to the workflow. Function f_1 has multiple output channels, while the others have just one. Details of the state models are discussed in Section 2.4.

2.1. Programming model

As briefly introduced in Section 1, EDGELESS applications are deployed as *workflows*, each associated with a specific user or session. Fig. 4 illustrates an example workflow comprising six stateful agents of different types: three functions and three resources. A *function* is a reactive agent that consumes received events and generates further events that flow to functions or resources of that workflow. The only side effect a function is allowed is to update the local state associated with a specific function instance or shared among multiple function instances of the same workflow. Functions thus can have no interaction with external services or devices, allowing them to have a consistent I/O interface based on event passing. In contrast, a *resource* is a stateful agent that interfaces between EDGELESS and the external world. It generates events as a *source* (or trigger) and consumes events as a *sink*, but it also offers/consumes APIs of external services and can have side effects such as writing files or updating a database.

Functions and resources may have multiple *channels*, which are the possible outputs through which events can be dispatched. Each channel is associated with a label, part of the function/resource specification provided by the developer, allowing the workflow developer to build a service by interconnecting known functions and resources in arbitrary graphs.[2] An edge between two elements in the workflow graph implies simply that the dataplane will ensure any event generated on the given output channel will reach the successor's given input port, and not that events will be generated every time the workflow is triggered.

For both functions and resources, events can be either synchronous or asynchronous:

- *Synchronous events* are akin to function calls or remote procedure calls in that the caller blocks after issuing a synchronous event towards an output channel until a response from the callee or an error is received. This mechanism can be used to transfer a message from the callee to the caller,
- *Asynchronous events* are simply dispatched from the caller to the callee, with the caller continuing immediately after event generation. No indication is given to the caller as to whether the event was properly received by the callee, and the latter cannot send callback data to the former.

Asynchronous events are preferred as: (*i*) Synchronous events can lead to deadlock when calls occur in a cycle (e.g., f synchronously calls g, which calls back to f); many systems avoid this problem by constraining workflows to be DAGs, but this does add an otherwise unnecessary constraint on how services can be composed. (*ii*) synchronous events may reduce system utilization and increase end-to-end latency, e.g., if multiple events are received in a batch by a function issuing synchronous events, the function misses the opportunity to start working on pending events each time it blocks waiting for a response. This problem arises for the same underlying reason as the *double billing* issue, well known in cloud serverless computing [25]: whenever a function blocks waiting for a response, both the caller and the callee are considered "in use" and so billed to the customer, even if only the callee is actually doing work.

Fig. 5 illustrates the Finite State Machine (FSM) of a function instance to assist understanding the pieces of logic the function developer must implement. As shown, the lifetime of a function instance begins when it is started by the ϵ-ORC following the

[2] Channels are not currently *typed*, so the workflow developer must rely on function/resource documentation to understand what type of event to expect on an output channel, including the encoding and the semantics of data contained.

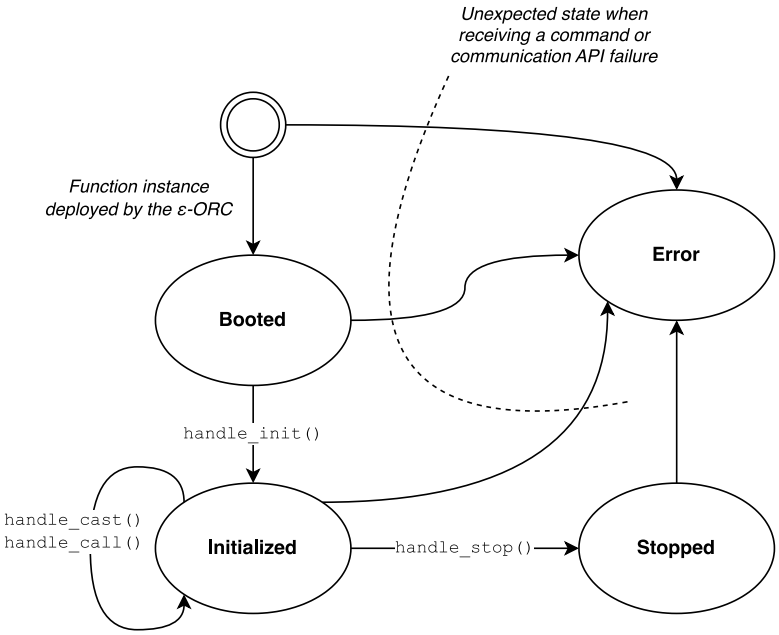


Fig. 5. Finite state machine of a function instance.

Table 2
Function APIs.

Method	Description
APIs to be implemented for a function	
`handle_init()`	Called when a function instance starts to set up necessary state and configuration
`handle_cast()`	User for asynchronous requests that may not respond to the caller
`handle_call()`	Used for synchronous requests to handle processes expecting a return value
`handle_stop()`	Invoked to clean-up when a function instance is terminated
APIs that function methods can use	
`cast(alias, args)`	Asynchronously call the function/resource instance with given `alias` in the workflow, providing any desired arguments `args`
`delayed_cast(delay, alias, args)`	As `cast()` but the invocation is delayed as requested
`call(alias, args)`	Invoke the function/resource instance with the given `alias` in the workflow, providing any arguments `args`, and waiting for a response
`log(level, msg)`	Log a message at the given severity level
`sync(state)`	Persist the current version of the state

deployment model described in detail in Section 2.2. After the booting phase, it enters an Initialized state by invoking the user-supplied `handle_init()` method, which prepares the function instance's internal data structures. In this state, the function instance reacts to asynchronous events via the `handle_cast()` method and to synchronous events via `handle_call()`. As the instance runs in a single-thread main loop, the developer does not need to synchronize concurrent access to data structures or other operations, needing only to focus on application logic following this reactive/functional pattern. The function instance remains operational until it is terminated by the ϵ-ORC, triggering execution of the `handle_stop()` method for optional clean-up purposes.

The functions APIs that a function must implement are summarized in Table 2, which also reports the APIs that EDGELESS offers to functions. Developers use the latter to interact with the rest of the EDGELESS system from within a function, chiefly to invoke other functions or resources via `cast()`/`delayed_cast()` and `call()` methods. The supplementary material includes as an example a complete implementation of a Rust function that computes a moving average of input values received. Invoking the cast/call methods requires providing an *alias*, a label that identifies a specific function or resource within the workflow specification, as seen in the example JSON workflow the supplementary material. Such aliases are mapped at run-time by the ϵ-CON and ϵ-ORC to

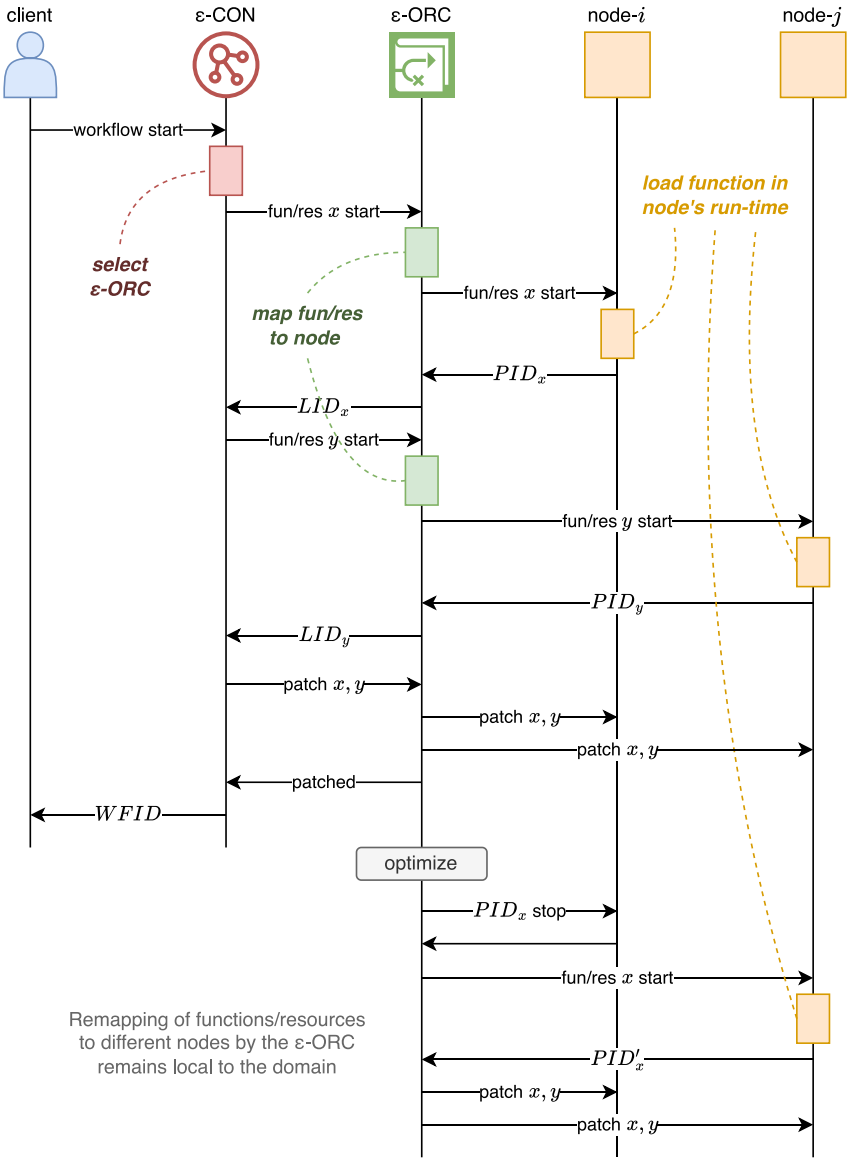


Fig. 6. Example sequence diagram of deployment of a workflow consisting of two functions (or resources) x and y.

internal Universally Unique Identifiers (UUIDs), which are distinct for every workflow and function/resource instance, as illustrated below.

2.2. Deployment model

An EDGELESS workflow contains the following information (a full example with two functions and two resources chained together is included in the supplementary material):

- The set of the workflow's function instances, each identified by a user-provided alias used to reference it within the workflow, and specifying the data used by the node to deploy it, plus any function-specific configuration. Details vary depending on the specific run-time used (see Section 2.3), e.g., the WebAssembly client includes the bytecode of the function itself in the workflow specification, which can be loaded from the client's filesystem or retrieved from a function repository via an APIs.
- The set of the workflow's resource instances identified, as are functions, by its alias plus the *type* of resource provider, e.g., file logging, or HTTP ingress/egress. The set of resource providers available in an EDGELESS cluster can be obtained by the client from the ε-CON at run-time. The client specifies the resource-specific configuration parameters for each resource instance.

- The mapping of output channels of all functions/resources to other functions/resources in the same workflow, via their aliases. This allows the ε-CON and ε-ORC to configure at run-time the EDGELESS dataplane so that asynchronous/synchronous events are dispatched to the intended recipients.
- The workflow and function annotations, discussed below.

We illustrate workflow deployment with the help, in Fig. 6, of a sequence diagram of the management plane interactions for an example workflow consisting of two functions (or resources) x and y. A client makes a request to the ε-CON, which selects the orchestration domain to handle the workflow and then interacts with that domain's ε-ORC to start the two functions/resources. The ε-CON also indicates how to map the output channels to the respective logical functions, as requested in the workflow, via a *patch* command. The ε-ORC decides the mapping between functions/resources and nodes in its domain, and may modify that mapping at any time following events such as node failure or when optimizing domain resources. After all functions/resources instances have been successfully created, the ε-ORC configures the nodes' local components of the dataplane via patch commands, depending on the output channel mappings received. The client receives an identifier for the whole workflow ($WFID$), the ε-CON is provided with logical identifiers for individual functions/resources (LID_x and LID_y), and the ε-ORC manages physical identifiers with nodes (PID_x and PID_y). Only the latter are subject to change during the workflow's lifetime.

For the ε-ORC and ε-CON to make appropriate decisions about the allocation/relocation of functions to edge nodes during the workflow lifecycle, the workflow and its functions are given *annotations* specifying *Service Level Objectives (SLOs)*, characteristics, and deployment constraints. The SLOs specify performance targets to be achieved (e.g., bounded latency for the execution of a given function) and workflow characteristics relevant to orchestration (e.g., the average or peak transaction rate). The characteristics provide optional information about, e.g., the expected invocation and transfer rates, to guide the decisions made by the ε-ORC and ε-CON. The deployment constraints determine which nodes may or may not be used based on their hardware characteristics (e.g., availability of Intel SGX to run functions in a Trusted Execution Environment (TEE) [26]), software features (e.g., availability of a specific run-time, see Section 2.3), or configuration parameters (e.g., free-text labels associated with each node by the system administrator). For completeness, the supplementary material lists the workflow and function annotations defined in EDGELESS so far.

A note on autoscaling. Serverless computing has been very successful in cloud computing because it entices three stakeholders: the programmers, who benefit from the FaaS programming model, the customers, because of its flexible billing, and the platform operators, thanks to the efficient usage of resources. Autoscaling plays a decisive role in the latter: by adapting the number of workers to the instantaneous load, it makes the platform parsimonious at zero/low load — no resource wastage with a scale-to-zero policy — but, at the same time, it copes well with high loads, building on the ability of underlying virtualization platforms to spawn multiple instances of the same container relatively fast. While the autoscaling feature is intended to match a load of aggregate requests from multiple users, in principle it can also be exploited to scale a single instance of certain applications to obtain the desired result faster; this opportunity is studied, for instance, in [27] for highly parallel linear algebra operations.

As introduced in Section 1.1, the stateful agent pattern that we advocate in this work is fundamentally different from serverless computing, even if we reuse (and extend) its FaaS programming model. Following our pattern, there is no aggregation of requests from multiple users. Rather, each user is associated with an instance of the workflow requested, which allows us to keep the user's state local within the function instance. This novel approach does not require autoscaling for the optimization of resources, like in a serverless computing platform: the role of the autoscaler is played by the ε-ORC and ε-CON, at each respective time scale and orchestration level, by re-arranging the mapping of logical function instances to physical nodes based on the telemetry data.

For example, let us compare what happens when there is a flash-crowd effect of applications consisting of a single function $f()$. In a serverless platform, a flash crowd occurs when there is a sudden increase in $f()$ execution calls due to many users, say N, requesting access to a given service simultaneously. Ideally, the autoscaler should be able to detect such a trend, based on measured function execution time or ingress queue length/rate, and perform corrective actions, mainly consisting of the creation of M more containers able to serve the new load of $f()$ calls. Deciding the specific value of M may not be trivial, and also depends on how the autoscaler prefers to explore the trade-off between the risk of overallocation of resources vs. underperforming of the application. With stateful agents, a flash-crowd happens when many users, again say N, create new workflows in a short amount of time. For every workflow creation request, the ε-CON assigns it to an orchestration domain, where the ε-ORC selects the node that will serve $f()$ for that specific user. So, in the end, there will always be N function instances; the only choice would be on which domains/nodes to run them.

Clearly, we cannot argue that the stateful agents pattern is superior to serverless computing, in general. For instance, highly parallel applications, where each instance has a massive load and requires running on many workers, cannot be served under this model, precisely because of the lack of autoscaling, like in serverless computing. However, in this work, we elaborate on the statement that the stateful agents pattern is a sensible choice for scenarios where each application has modest computing requirements and the infrastructure is decentralized, which suits very well typical IoT requirements.

2.3. Flexible node design

Unlike cloud computing counterparts, EDGELESS nodes must run on a wide range of devices with heterogeneous characteristics, including far-edge elements with limited computational resources and connectivity. We thus designed the EDGELESS node as a lightweight component comprising multiple function run-times and resource providers, depicted in Fig. 7. The *node agent* is the interface between the ε-ORC and the function run-times and resource providers, allowing the former to manage instance lifecycles via `start` and `stop` commands. Currently, three types of function run-times are supported, but the flexible node design allows more to be added without changing the APIs:

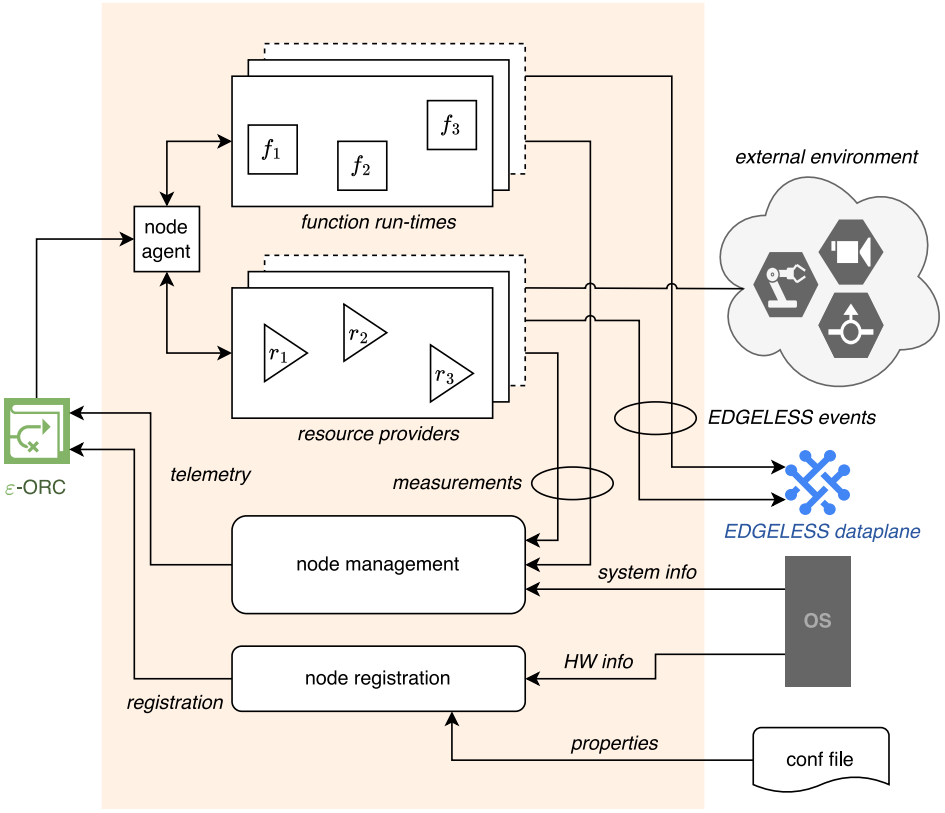


Fig. 7. Blueprint of an EDGELESS node, including the main elements with which the node interacts. A node may support multiple function run-times, each with its own virtualization abstraction, and many resource providers enabling EDGELESS applications to interact with the external world (triggers, sources, sinks). Function and resource instances interact via a dataplane interconnecting all nodes in an orchestration domain. The node collects telemetry data about the execution of function/resource instances, and from system information (e.g., CPU load and network usage) provided by the OS. These data are delivered to the ε-ORC to make appropriate resource management decisions. Nodes announce themselves to the ε-ORC through an orchestration domain formation procedure, providing their hardware capabilities and the values of parameters read from a configuration file.

- The **WebAssembly** run-time supports new functions developed specifically for EDGELESS to carry out simple operations like data conversion and semantic checking. WebAssembly is portable, which allows the source code to be compiled into bytecode once for any target platform (hardware, OS) but does preclude use of hardware-specific optimizations and specialized libraries.
- The **container** run-time enables a programming model similar to that of traditional FaaS, permitting reuse of existing code to perform more complex operations, compared to anything typically available in a WebAssembly environment.
- Finally, **native execution** allows: loading of arbitrary functions, including third-party libraries and tools, linking with the EDGELESS node application itself. Unlike the other run-times, it does not provide strong isolation or safe execution (sandboxing) in favor of higher performance.

The set of resource providers available at a node depends on the real-world interactions possible for that node. A node may offer resources of a type if it is hosted on a device that is physically connected to suitable sensors/actuators or is logically connected to a relevant external service, e.g., a Redis server or a Kafka cluster. Both function and resource instances are connected to the EDGELESS dataplane, allowing the exchange of events following the composition patterns specified in each active workflow.

Furthermore, a node includes housekeeping components for node management and registration. **Node management** refers to the ability of a node to provide telemetry data about the node's run-time information (e.g., CPU load, memory occupation, network traffic data) and function/resource measurements (e.g., execution times) to the ε-ORC. **Node registration** allows nodes to announce themselves to the ε-ORC, contributing to the orchestration domain formation process. During the registration, the node also includes information about the node's static characteristics (e.g., total number of cores, available memory, availability of special hardware) and configuration. Static and dynamic information held by the ε-ORC allows it to make decisions about reconfiguration/optimization of the function and resource instances.

Finally, we discuss how our design interacts with the well-known **cold-start** problem in serverless computing: the extra latency an incoming request experiences when assigned to a newly-started container rather than one that has already been started (i.e., is warm). As a result, cold-start primarily impacts tail latencies, and some of the many approaches to mitigating it are discussed in

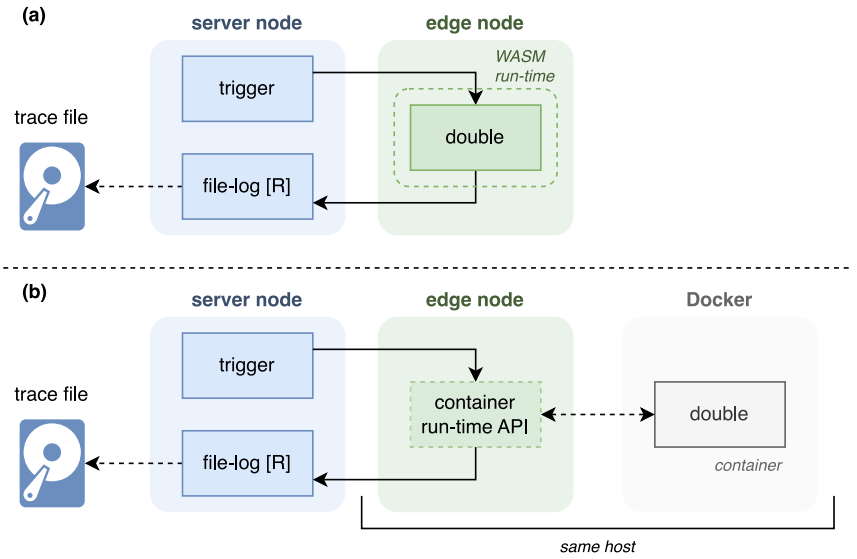


Fig. 8. Workflows used in the experiments in Section 2.3, with the function under test running in the (a) WebAssembly vs. (b) container run-time environment of the node.

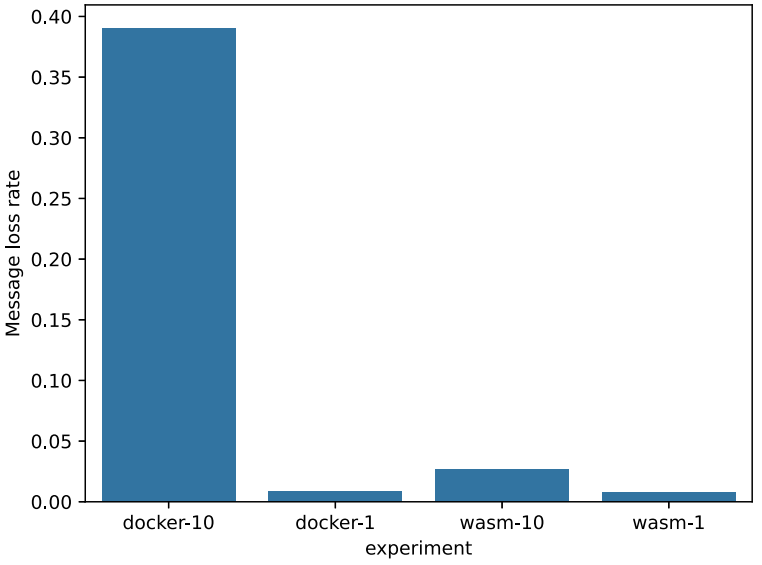


Fig. 9. Loss rate with WebAssembly vs. Docker functions, with 1 and 10 workflows, respectively, when migrating all workflows every 1 s.

Section 4. These typically rely on the fact that user requests are not associated with workers and functions are stateless, to either pre-warm containers in anticipation of future needs or to reduce setup time. With stateful agents, each worker is assigned to a specific stream of requests associated with a given user or session, thus belonging to the same workflow in EDGELESS terminology. Therefore, except for the very first request that a client issues after the workflow is created, there should never be a cold-start effect, although it could still occur if the ϵ-ORC were to terminate a function instance in an active workflow to reclaim unused resources such as allocated memory. Except for the Docker run-time, intended primarily for legacy functions, the footprint of an EDGELESS function instance is very small (see Table 1 in Section 1) and so such an optimization is less appealing than in traditional serverless computing platforms built around autoscaling.

We now report the results obtained with an experiment aimed at assessing the scalability of managing functions running in WebAssembly vs. Docker containers. The workflow used is illustrated in Fig. 8 for the two cases. The testbed consists of 5 NVIDIA AGX Orin nodes, which are assigned a toy function `double` that reads the number received in the incoming message and produces as output a message with the value doubled, and a VM on an Intel server running the ϵ-CON, ϵ-ORC, and a node with ancillary components to drive the experiment, i.e., the `trigger` function that emits an incremental number every 10 ms and a `file-log`

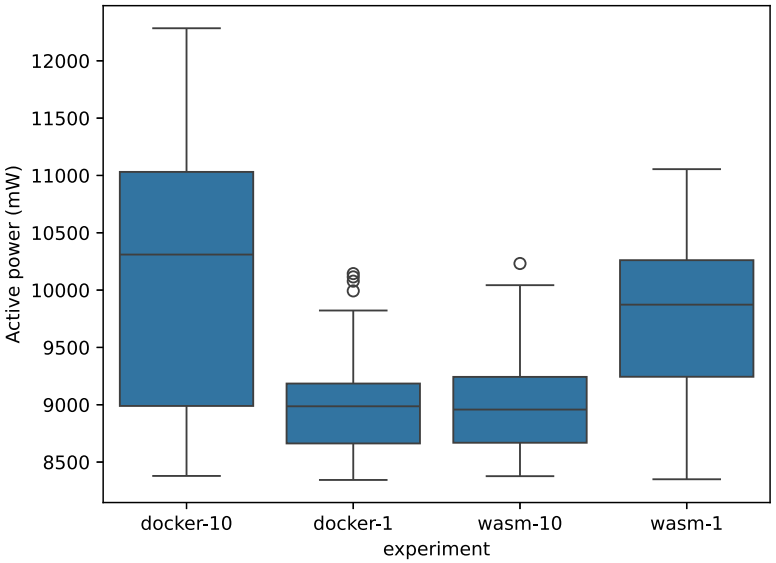


Fig. 10. Active power of nodes with WebAssembly vs. Docker functions, with 1 and 10 workflows, respectively, when migrating all workflows every 1 s.

Table 3
Comparison of EDGELESS state management models.

Model	Pros	Cons	Use cases
Function-state	Simple to implement No need to sync Fast in-memory access	Lost on termination Not shareable	FIFO queues Moving averages Stateless analytics
Sync-state	Allows state reuse across instances Supports state updates	No reloading of updated state at runtime	ML data initialization Periodic retraining
Shared-state	Enables coordination across functions Supports complex workflows	Requires synchronization Risk of consistency issues Rely on underlying solutions	Collaborative workflows

resource to save timestamps for post-processing. The experiment lasts 100 s and is repeated with 1 workflow and 10 workflows. *During the experiment, we force the migration of all the* `double` *function instances from the current node to another one at random every 1 s.*

In Fig. 9 we show the loss rate, that is, the ratio between the number of messages that are lost due to the migration of the function instance and the total number of messages triggered. As can be seen, while with a single workflow the loss rate is comparable between WebAssembly and Docker, already with 10 workflows, corresponding to just two `double` functions per node on average, the Docker run-time exhibits a significant loss. We note that the system is operating in extreme conditions, with a migration rate that exceeds a reasonable operation level in the field, but still, the results show that the lightweight nature of WebAssembly makes it suitable for fast and scalable orchestration.

This is confirmed by the active power of nodes, reported in Fig. 10, which shows a sharp increase in energy consumption from 1 to 10 workflows with Docker. It is interesting to note that, with WebAssembly, the energy consumption with 10 workflows is generally *lower* than that with a single workflow. We speculate that such a behavior is due to the scaling policies implemented by the NVIDIA AGX Orin hosts, which are beyond the scope of this work and currently under separate investigation.

2.4. State management

In general, serverless functions are stateless and ephemeral, and do not maintain any data or state between invocations. However, the ability to maintain and manage state in the short lifespan of a function's execution is crucial for some applications. For this reason, in EDGELESS, we propose that the developer can select from several models of state management according to their needs, illustrated in Fig. 11 and Table 3.

With **function-state** (Fig. 11a), a function keeps its state inside its instance, e.g., container or WebAssembly worker, and thus needs no synchronization mechanisms as the function is single threaded by design. The state is created when the function instance starts, possibly with a procedure executed in the `handle_init()` function's method (see Section 2.1), and persists only until the

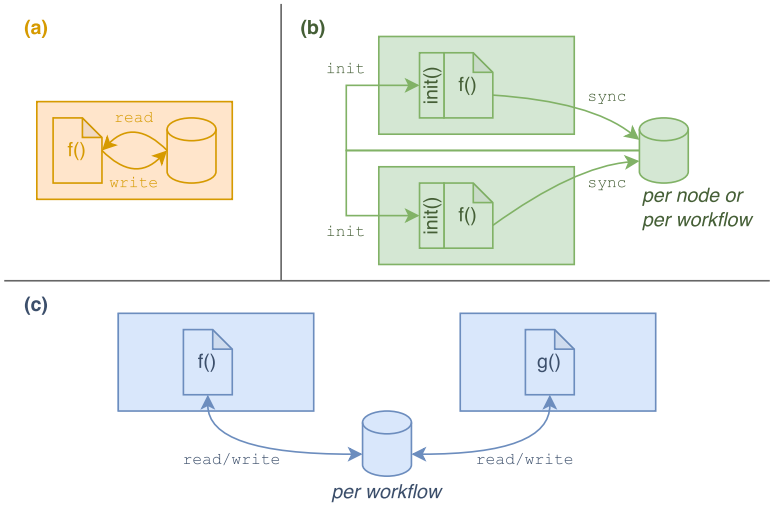


Fig. 11. EDGELESS state management models: (a) *function-state* where state is maintained at each function instance and can only be accessed from that instance; (b) *sync-state* where state is maintained in the node or for the whole workflow, and is used to initialize new function instances of a given type, which can update it at run-time; (c) *shared-state* where state is associated with the workflow and function instances can access it concurrently.

function instance is terminated either by migration to another node or by termination of the workflow. At that point, the state vanishes with the function instance unless specific measures are taken by the function developer in `handle_stop()`. This model is thus suitable for functions for which an ephemeral state suffices, such as that reported in the supplementary material, where the state is a FIFO queue of received values used to compute a moving average at each new function invocation.

With **sync-state** (Fig. 11b), a function's initial state is passed to its `handle_init()` from outside the instance. During the function's lifetime, it cannot again retrieve such an initial state, but it can update the initial state that will be used in the future by invoking the `sync()` method. This model is to support functions for which the preparation of the initial state is an expensive process, e.g., the training phase of an Machine Learning (ML) pipeline. With sync-state, training can be done once by the first function instance of a workflow, and then reused by subsequent instances. Furthermore, any function wishing to re-train, e.g., in a continual learning paradigm, can persist the new state at any time.

Finally, with **shared-state** (Fig. 11c), multiple functions in a workflow can share some state. This model is more complex than the others and requires special care. In fact, some have explored approaches that transform DAG workflows with shared states into equivalent ones using only local state plus state propagated through function invocation arguments, precisely to avoid the burden and complexity of this model, e.g. [28]. Due to the many possible and diverse needs of applications, we believe a one-size-fits-all solution for this model does not exist, but we have identified three categories of solutions based on the use of a centralized database, distributed state management, and distributed consensus, briefly described below.

The *centralized database* approach relies on a database existing in the orchestration domain to provide consistency and persistence guarantees. We implement this model via a dedicated resource provider in EDGELESS, `sqlx`, that uses the workflow identifier ($WFID$ in Fig. 6) to partition the database space and allow functions to access only data associated with their workflow. Developers send SQL commands directly to the database to retrieve or serialize all or part of their shared state, providing read/write commands.

Another approach is to use *distributed state management* to provide synchronized multi-reader and multi-writer access to state, supporting heterogeneous and widely distributed devices. In this context, this suffers from three challenges specific to edge computing. First, nodes may be highly unreliable, failing and recovering to re-join the system while still out-of-date. Second, network communication can be unreliable, with messages being lost or duplicated. Third, network communication is asynchronous so messages may take an arbitrary time to be delivered and may be delivered out of order.

Finally, *distributed consensus* protocols are even more complex and can be distinguished based on the type of consistency they offer: causal, session, and strong consistency.

- Causal consistency guarantees that the order of operations reflects their causal relationships using techniques such as vector clocks or timestamps to keep track of the order of operations. Causal consistency is relatively simple to implement and allows high concurrency.
- Session consistency is weaker, allowing monotonic read operations, ensuring old data is never read after new data, and monotonic writes, ensuring writes are totally ordered. One benefit of session consistency is that writes are immediately accessible to reads, but an extra cost is incurred by creating new session tokens after every write.
- Strong consistency ensures all parallel processes are in the same order, i.e., each read operation returns the latest result of a write operation, irrespective of the node on which the read operation executes. This guarantees that all nodes receive the same

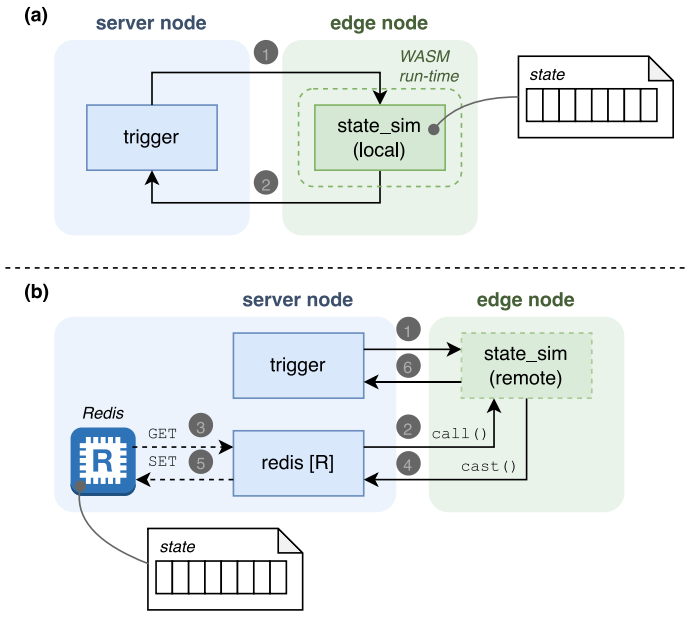


Fig. 12. Workflows used in the experiments in Section 2.4, with the function under test (`state_sim`) operating in two different modes: (a) with the state kept local in the instance, following the stateful agent pattern vs. (b) with the state in an external service, in particular a Redis server accessed via a `redis` resource, which is the traditional approach in serverless computing.

data at the cost of extra coordination and communication between nodes, and is usually achieved using a consensus system such as Raft [29] or Paxos [30].

Designing and implementing suitable techniques for EDGELESS to go beyond a centralized database will be a focus of our future work, for which we have two promising directions. First, in [31] the authors have put forward a serverless platform, called Pheronome, based on a data bucket abstraction that can be accessed by functions for read/write operations: even if the bucket is intended to hold intermediate data generated by functions (events in EDGELESS), we believe that its scalability and low-latency properties make it a good candidate also for durable state storing. Second, StructMesh [32] has some interesting properties that can be relevant to shared state management in EDGELESS: it is a serverless storage framework designed for uniform data access through multiple infrastructures in the edge–cloud, supporting load balancing for efficient resource usage and reliability/security for applications with such specific non-functional requirements.

We now report the results obtained with an experiment aimed at assessing the cost of running a stateful function with the state stored in an external service, which is the typical way in which stateful applications are deployed in serverless computing platforms. The workflow used is illustrated in Fig. 12 for the two cases, where the state is a vector of 32-bit floating point numbers. Following the stateful agent pattern (Fig. 12a), the state is kept in the local memory of the function instance, whereas a stateless function (Fig. 12b) needs to retrieve and then save back the state at every invocation. The operation performed by the function under test `state_sim` is an increment by 1 of all the elements in the vector, which are initialized to random numbers when the function instance starts. Like in Section 2.3, the testbed consists of 5 NVIDIA AGX Orin nodes, which are assigned `state_sim` instances, and a VM on an Intel server running the ϵ-CON, ϵ-ORC, a node that triggers execution every 100 ms, and the Redis server to keep the state, when needed. The experiment lasts 60 s and is repeated with different state sizes in $\{1, 100, 100\,000\}$, as well as with 1 vs. 20 workflows.

In Fig. 13 we report the latency measured in the experiments. As can be seen, when the state is local, the latency increase due to a bigger state or a higher number of workflows is modest (note the logarithmic scale on the y-axis), where all samples remain below a few ms. On the other hand, with a remote state, the latency increases significantly already with 1 workflow, and a small state of 1 or 100 vector elements, due to the need to perform the `GET`/`SET` Redis operations. We note that the NVIDIA AGX Orin nodes are connected with the Intel server hosting the VM with Redis via a fast 10 Gb/s Ethernet LAN, which was underloaded during the experiments. This cannot prevent the latency from becoming significant with a size of 100k elements. Moreover, with 20 workflows, due to the contention on the Redis and other shared components, such as the dataplane interconnecting the function instances, the latency is affected by a high jitter, which would degrade further the Quality of Service (QoS) perceived by the application.

3. Implementation

We provide a reference implementation of EDGELESS under an MIT license on GitHub [19]. It is written in Rust [33], which is emerging as a memory/thread-safe alternative to traditional high-performance strongly-typed alternatives, such as C++. The

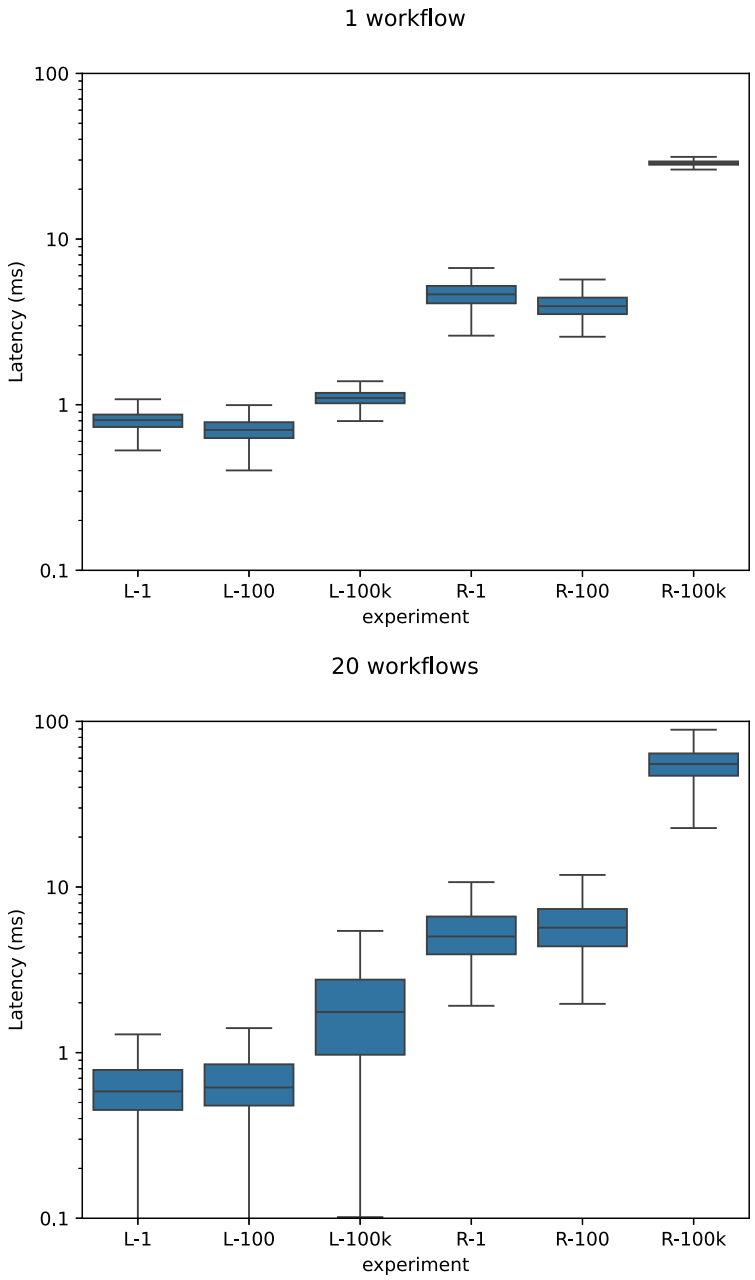


Fig. 13. Service latency of local (L) vs. remote (R) state, with 1 and 20 workflows, respectively, and variable size of the state.

APIs is defined and implemented using Google's gRPC [34], which is widely employed for inter-process communication in many cloud-native production-grade applications and services.

EDGELESS provides a holistic framework to implement, deploy, manage, and optimize services adopting the stateful agents pattern in a decentralized computing infrastructure, without relying on an external cloud or underlying services such as a container management system or an event streaming platform. The repository contains numerous scripts and examples allowing adopters to experiment with several out-of-the-box features: multi-domain EDGELESS clusters, HTTP triggers/sinks, notifications to external systems such as Apache Kafka and Redis, local log files, asynchronous and synchronous events handling, workflows with DAGs as well as circular function dependencies and mixed run-time workflows.

Among these features, in Section 3.1, we linger on the *serverless resource*, which introduces the possibility to interconnect EDGELESS workflows, made of stateful agents, with traditional stateless functions running in an external platform. Furthermore, in Section 3.2, we focus on the ϵ-ORC implementation because it plays a critical role in achieving high performance by keeping

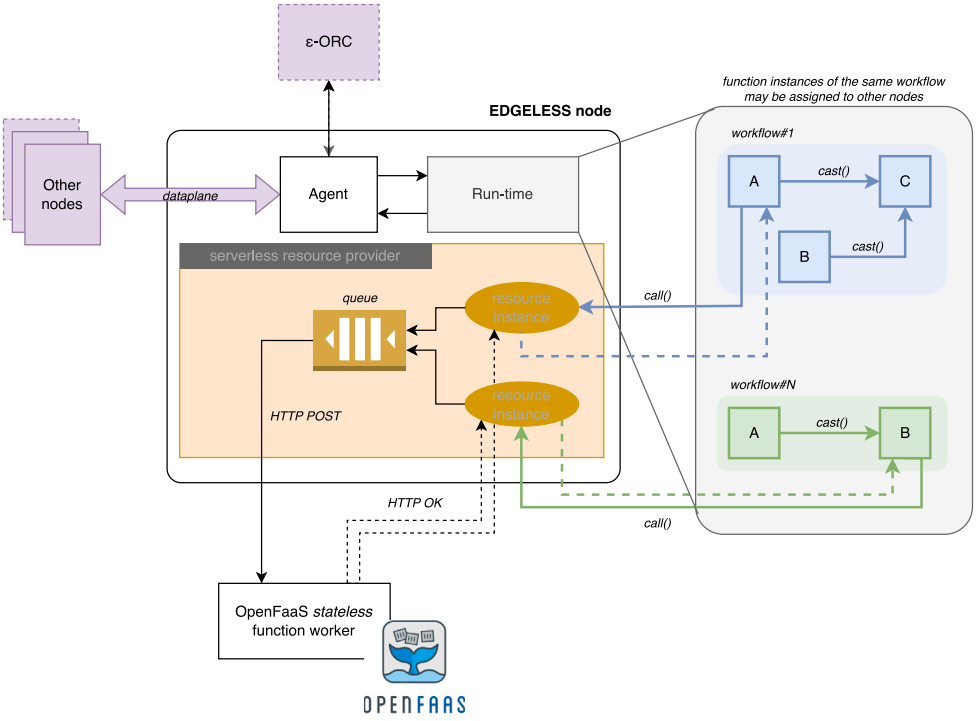


Fig. 14. Software architecture of the serverless resource in EDGELESS, which acts as a gateway for regular function/resource instances to call traditional stateless functions running in a serverless platform.

system utilization high while still meeting the application SLOs. Numerous other works have addressed the orchestration challenge in a wide range of settings, but a single conclusive solution has yet to be found. The interested reader can find relevant pointers to the state-of-the-art orchestration in the edge–cloud continuum in Section 4.

Finally, in Section 3.3, we analyze the differences between EDGELESS and wasmCloud, which is a mature technology aiming to enable the development of WebAssembly components and their seamless execution across cloud and edge infrastructures. The analysis also includes experimental results obtained in a testbed of nodes with limited resources, highlighting the advantages of a key distinguishing feature of EDGELESS, i.e., native support of stateful agents.

3.1. Serverless resource

Despite our enthusiasm about stateful agents, we recognize that this pattern may not adapt efficiently to the application logic of functions that are, by their nature, truly stateless. For this reason, in EDGELESS, we have provided a way for the workflow application developer to seamlessly mix-and-match stateful and stateless functions: the former are executed in one of the run-times available, as illustrated in Section 2.3, while the latter can be deployed as regular serverless functions and called by EDGELESS function/resource instances.

In practice, this is realized through the *serverless resource*, according to the architecture illustrated in Fig. 14. Remember that resources are special functions that, unlike functions, can access services available outside of EDGELESS. In this particular case, the service is a traditional serverless computing function, which may be running in a full-fledged serverless platform or directly as a stand-alone application running on the node itself. A serverless resource offered by a node is identified by the name assigned in the node's configuration and advertised to the ϵ-ORC to make it available to the active workflows. The function/resource instances invoke the stateless function by means of the `call()` primitive (see Table 2), which is synchronous and returns the result of the function invocation, with arguments passed in the called event, when completed. Multiple workflows may be associated with the same serverless resource: call events may queue at the resource, which will ensure non-overlapping invocations. In the current version of EDGELESS, we support the OpenFaaS function invocation APIs, but more can be easily added, as needed.

In addition to adding flexibility in designing a workflow with mixed stateless/stateful functions, the serverless resource also provides a smooth path for the migration of an application from traditional serverless, e.g., with OpenFaaS, to EDGELESS.

3.2. Delegated orchestration

Orchestration in the reference implementation of EDGELESS has been implemented through the concept of *delegated orchestration*, inspired by the Kubernetes Operator pattern [35], which allows the addition of automated policies to a cluster without modifying the Kubernetes code itself.

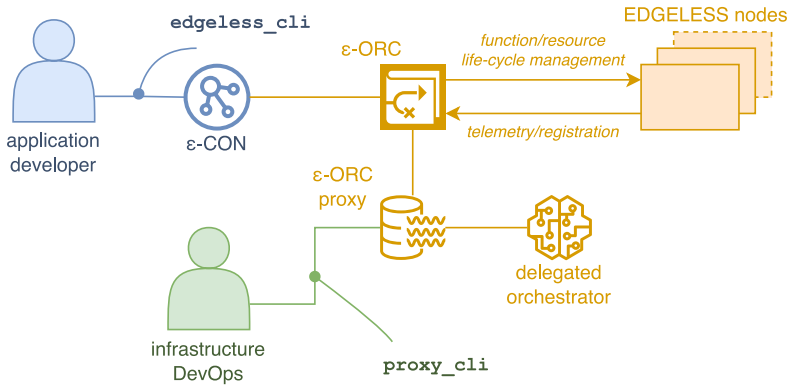


Fig. 15. High-level view of the delegated orchestrator concept implemented in the EDGELESS reference implementation.

Table 4
Health information provided by the nodes to the ϵ-ORC, which is saved in the domain observability layer and by the delegated orchestrator to make optimization decisions.

Short name	gRPC type	Description	Unit
`mem_free`	`int32`	Free memory	kbytes
`mem_used`	`int32`	Used memory	kbytes
`proc_cpu_usage`	`int32`	CPU usage of the current process	%
`proc_memory`	`int32`	Memory occupied by this process	kbytes
`proc_vmemory`	`int32`	Virtual memory occupied by this process	kbytes
`load_avg_1`	`int32`	Load, one minute average	%
`load_avg_5`	`int32`	Load, five minutes average	%
`load_avg_15`	`int32`	Load, fifteen minutes average	%
`tot_rx_bytes`	`int64`	Traffic received on the network interfaces	bytes
`tot_rx_pkts`	`int64`	Traffic received on the network interfaces	packets
`tot_rx_errs`	`int64`	Total rx errors on the network interfaces	
`tot_tx_bytes`	`int64`	Traffic transmitted on the network interfaces	bytes
`tot_tx_pkts`	`int64`	Traffic transmitted on the network interfaces	packets
`tot_tx_errs`	`int64`	Total tx errors on the network interfaces	
`disk_free_space`	`int64`	Total disk space	bytes
`disk_tot_reads`	`int64`	Total disk reads	bytes
`disk_tot_writes`	`int64`	Total disk writes	bytes
`gpu_load_perc`	`int32`	GPU load	%
`gpu_temp_cels`	`int32`	GPU temperature	1/1000 °C

Delegated orchestration is illustrated in Fig. 15. As described in Section 2.3, the ϵ-ORC manages the lifecycle of function and resource instances on nodes and receives static registration information from the latter, plus dynamic telemetry measures consisting of function execution latencies associated with the handling of event and transfer times including network and queueing delays, and the health information reported in Table 4. These data are kept by the ϵ-ORC in internal data structures, which in principle would require updating the ϵ-ORC every time its policies must change. This is impractical for two reasons: (*i*) Modifying the source code of the ϵ-ORC requires Rust programming language experience and familiarity with not only EDGELESS design but also the codebase of its reference implementation, creating an entry barrier for new adopters. (*ii*) Orchestration policies are usually updated dynamically in an edge cluster to adapt to the changing conditions of both the computing infrastructure and the workload, requiring a reboot of the ϵ-ORC, with possible service disruption. To overcome both, the ϵ-ORC keeps a mirror of its internal data structures in an external in-memory database, currently realized as a KVS with Redis that can be inspected by third parties, whether human agents via a command-line interface utility (`proxy_cli`), or automated agents, possibly augmented by Artificial Intelligence (AI). Such agents can analyze the data on their own timescales, reacting appropriately by proposing relocation of function or resource instances from one node to another. The ϵ-ORC then enforces the request only if it is consistent with its model, e.g., if the deployment constraints remain valid.

Some applications of the delegated orchestration concept are:

- Manual or semi-automatic intervention by an operator, e.g., to remove all load from an edge node that is misbehaving or planned to undergo maintenance.
- Automatic orchestration based on an algorithm running at the edge or in the cloud, which can either run periodically to optimize the system operation based on epochs or be activated based on internal/external triggers (e.g., when a ticket is opened by a customer or a node becomes overloaded).

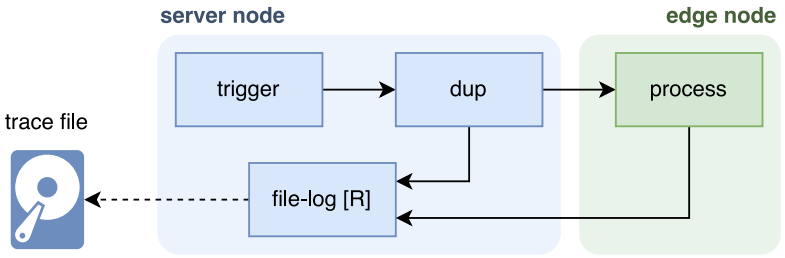


Fig. 16. Logical view of the workflow deployed in proof-of-concept experiments. The trigger function sends messages containing an incremental counter following a Poisson distribution, with an average of 100 ms. As its name implies, the `dup` function duplicates the received input to the two output channels. One copy is serialized to a log file immediately by the file-log resource, while another is dispatched to a process function. The latter computes the 10,000th element of the Fibonacci sequence before forwarding the received message to the file-log. The trigger, dup, and file-log instances are ancillary and must be deployed on the same server hosting the ϵ-ORC and ϵ-CON. In contrast, the workflow annotations specify that the process function must be allocated on a resource-constrained edge node.

- Identification of performance or security anomalies, notified to an external system for manual intervention or used internally as triggers for automatic reactions. Performance anomalies might arise due to uneven use of resources or violation of application SLOs, whereas security anomalies might include Denial of Service (DoS) attacks or unfair use of resources.
- Long-term storage of run-time measurements: an agent can stream the updates from the proxy to long-term storage, possibly through aggregation techniques to reduce the volume of data. The resulting dataset can then be used to train AI models, e.g., for cognitive orchestration, as well as for Business Support System (BSS)/Operations Support System (OSS) processes.

The delegated orchestrator is optional and runs in parallel with a baseline orchestrator policy embedded in the ϵ-ORC that takes care of immediate actions upon creation/termination of function/resource instances, plus other relevant events such as node failures. Such a baseline policy guarantees consistency through simple, yet robust, algorithms, but does not attempt to optimize system performance, a challenge delegated to third-party agents. In the current implementation, we assume that there is just one agent making decisions at a time. In this context, ensuring consistency and achieving consensus among multiple agents, avoiding race conditions, and pursuing contrasting objectives is an open research challenge that we leave for future study.

In Fig. 15, we also show the interaction with the application developer, in charge of creating/terminating workflows, via another command-line interface tool called `edgeless_cli`, which is invoked on an API offered by ϵ-CON to the users.

In the remainder of the section, we present the results of proof-of-concept experiments involving delegated orchestration to answer the following questions: *Is the platform able to enforce the indications from the delegated orchestrator?* and *Can the platform support workflow execution even when nodes leave ungracefully?* The delegated orchestrator used in the experiments is developed in Rust and periodically checks the system status to balance the number of allocated functions among nodes. The source code is available in `delegated_orc` in [20].

The experiment involved multiple instances of a workflow (Fig. 16) that computes the Fibonacci sequence, a CPU-intensive task. Additionally, edge nodes were shut down during the workflow execution to observe the behavior of the delegated orchestrator when nodes fail. The testbed used consisted of four nodes: one server node hosting the ϵ-ORC, the ϵ-CON, and the ancillary functions of the workflow, plus three heterogeneous edge nodes (a Raspberry Pi 3 Model 3B, an NVIDIA Jetson AGX Orin 64 GB, and an NVIDIA Jetson Xavier NX). All nodes used the WebAssembly run-time.

The experiment lasts for approximately 470 s (around 8 min). Fig. 17 shows the evolution of two metrics for each of the nodes: the number of functions allocated (solid line) and the CPU usage (dotted line). At 50 s into the experiment, we added one workflow every 5 s until 20 workflows were in the system, taking up to one minute for workflow creation. Initially, the ϵ-ORC assigns the workflows randomly to the nodes, which are in turn periodically balanced among nodes by the delegated orchestrator. This behavior is evident from the changes in the number of allocated functions in the plot. We then kept the system stable for one minute. Around the 220 s mark, we shut down the Orin node. The 7 functions allocated to Orin at that point were reallocated to the Xavier and RPi nodes by the ϵ-ORC and further rearranged by the delegated orchestrator shortly after (for example, the Xavier node's functions decreased from 11 to 10 due to this reallocation). Around the 280 s mark, we resumed the Orin node. After another minute (around the 350 s mark), during which the delegated orchestrator reallocated the functions again, we shut down the RPi node. Once again, the functions running on the killed node were reassigned to the others. Finally, around the 410, s mark, we restarted the RPi node, which was immediately assigned 6 functions by the delegated orchestrator, balancing load among nodes.

From these proof-of-concept experiments, we make two observations: (*i*) the platform is capable of accepting more advanced orchestration decisions from the delegated orchestrator, as demonstrated by the continuous rebalancing of functions among the nodes; and (*ii*) the platform can continue executing multiple workflows even in the event of node shutdowns, as evidenced by the fact that the number of functions concurrently being executed never drops below 20.

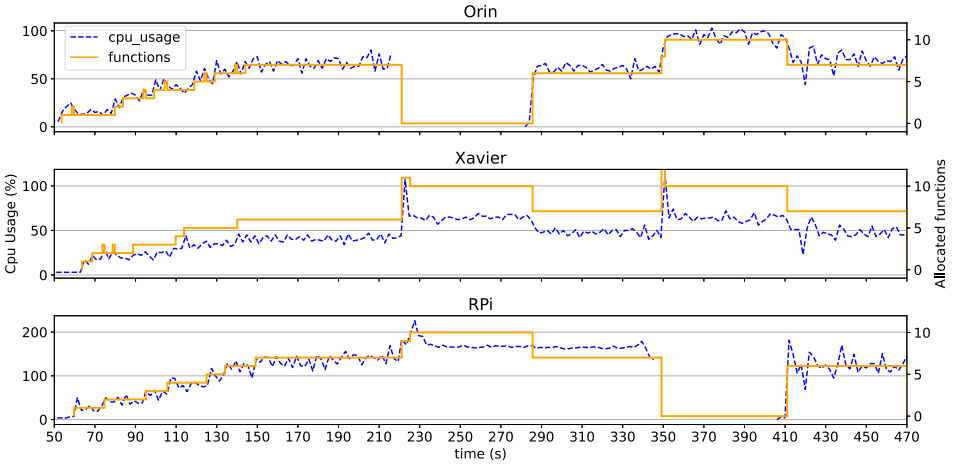


Fig. 17. Workflow allocation and CPU usage of the testbed during the experiment. The solid line represents the number of workflows allocated to the node, while the dotted line indicates CPU usage in percentage. The empty spots in CPU usage correspond to node shutdowns. The initial 50 s are cut off due to space constraints.

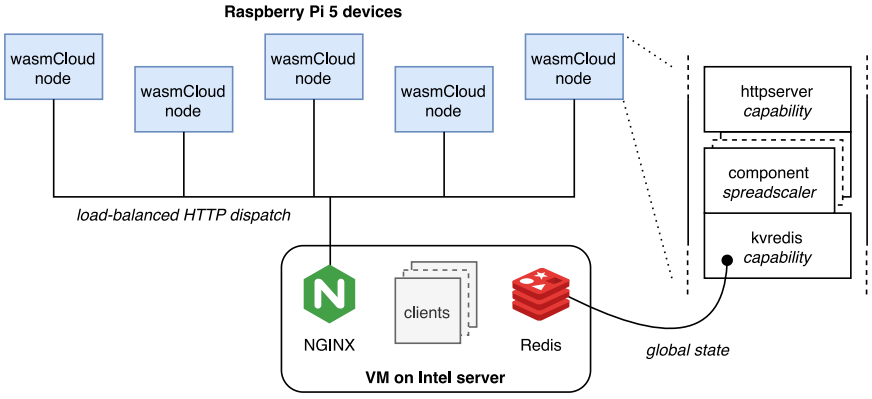


Fig. 18. Deployment of a stateful application as a wasmCloud component on five Raspberry PI 5 devices. The state is stored in a Redis KVS server, which is accessed by the components via the wasmCloud `kvredis` capability. The clients run in a separate host on a VM and reach the components via an NGINX load balancer.

3.3. Comparison with wasmCloud

We conclude the section with a comparison of EDGELESS with wasmCloud [36] as a possible alternative to develop and run FaaS applications in the edge–cloud continuum. First of all, we note that EDGELESS is an experimental platform developed as part of the activities of a collaborative project of the same name to explore and validate specific scientific challenges, whereas wasmCloud is high-maturity software currently maintained as a project of the Cloud Native Computing Foundation (CNCF). Therefore, they cannot be compared in terms of the complexity of features they offer, stability, or integration with tools and platforms in the edge–cloud ecosystem. We limit ourselves to reviewing the architectural differences and then showing how they can have a significant practical impact for some applications with testbed experiments.

wasmCloud is centered on the concept of *components*. A component is a binary piece of software implementing stateless logic, which can be composed with others to implement a more complex application logic and is reusable across different applications. A component is very similar to a function in EDGELESS: WebAssembly is adopted to make functions/components portable and lightweight; they are reactive; they are internally single-threaded; they are composable. Likewise, wasmCloud offers a concept similar to that of resources for long-lived services (called *providers*) offering common functionalities (called *capabilities*) to components, such as logging and persistence. However, there are two important differences between wasmCloud components and EDGELESS functions, which have motivated the development of our own software from scratch in the project. First, wasmCloud components are stateless, like in serverless computing, and need to rely on providers if their operation depends on a user's state. Second, components are arranged in a flat hierarchy, though a lattice offering the abstraction of a unifying mesh network that logically interconnects them with all the hosts and providers. On the other hand, EDGELESS conforms to the model in Section 2, which envisions stateful functions

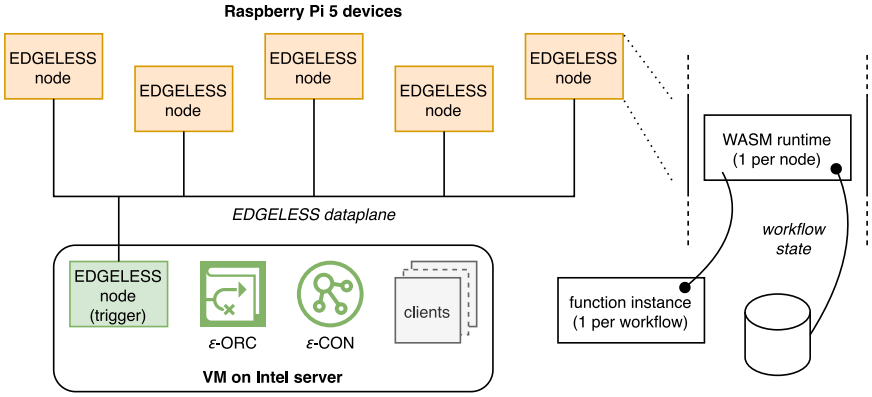


Fig. 19. EDGELESS version of the deployment in Fig. 18 on the same physical resources. The application is the same as that used in the experiments analyzed in Section 2.4.

to natively support the stateful agent pattern, and a two-level hierarchy to react faster to local changes happening in an orchestration domain. We believe that wasmCloud is the natural next step in the evolution ladder of serverless computing platforms, moving forward towards a more granular and distributed infrastructure, while EDGELESS, as a research sandbox platform, aspires to an even more forward-looking goal that completely detaches from serverless computing and fully embraces stateful agents.

In the following, we analyze the experimental results obtained to highlight precisely how this can be an advantage under some circumstances. We have used a testbed of five Raspberry PI 5 devices and a VM running on an Intel server, which is illustrated in Fig. 18. The application is the same in Section 2.4, where the client triggers a trigonometric operation on a vector of floating-point numbers that represent the user's state. Since the wasmCloud components are stateless by design, the state is kept on a Redis provider running in the VM accessed via the `kvredis` built-in capability. Each component is allowed to scale up to 10 instances per node, via the `spreadscaler` feature implemented by wasmCloud. The clients run in the VM and access the components, which offer an HTTP server, via a local load balancer implemented via NGINX [37].

The corresponding EDGELESS deployment is illustrated in Fig. 19, which shows that the user's state is kept local to each function instance. For every new client, a new workflow is created on the ε-CON. The workload is the same for wasmCloud and EDGELESS, and it consists of an increasing number of clients. Specifically, every minute we start 5 new clients in a batch, for a maximum of 10 batches. Each client triggers the execution of the function and then waits for 100 ms. Each experiment is repeated with different vector sizes: 1000 (1k), 100 000 (100k), and 1 000 000 (1M) elements.

In Fig. 20, we report the results obtained with wasmCloud (left column of plots) and EDGELESS (right column). First, the *application latency* with EDGELESS is largely independent from the load and increases only slightly with the state size, i.e., from 1–2 ms to 8–10 ms. On the other hand, wasmCloud suffers dramatically with increased load and larger state sizes, reaching 2 s (>200× EDGELESS) with 50 clients and 1M state size. This is due to the need for stateless components to access the state through their `kvredis` capability, which also impacts *throughput*: the latter decreases significantly with increasing state size with wasmCloud, while it is unaffected by it with EDGELESS.

The *average load*, as the average across nodes of the 1-minute average reported by each, shows that a significant fraction of the CPU resources are needed for state access. In fact, with EDGELESS, the load is very small and independent from the load with state size 1000 and 100k, and becomes non-negligible only with a larger state size of 1M elements, which exhibits a linear trend with the time/load, up to about 100%, i.e., one CPU core on average. With wasmCloud, the load increases significantly already from 1000 to 100k elements, and it is always much higher than that with EDGELESS. Finally, we report the total network traffic, which has a strong dependence on the state size and load with wasmCloud. Note that, with state size 1M, the traffic peaks after about 450 s and then decreases: this is because with that load the components cannot process fast enough the incoming requests, i.e., they become CPU-bound, as can be seen from the load plot. Instead, the network traffic with EDGELESS is absolutely negligible, because the state remains local to the corresponding function instances.

4. Related work

Our work describes an innovative pattern, *stateful agents*, implemented in the open-source platform EDGELESS, designed to overcome the limitations of both micro-services and serverless computing in edge–cloud deployments. In this section, we review the state-of-the-art in four areas relevant to specific aspects of EDGELESS, which can be considered for integration in future developments of stateful agents' frameworks: state management, cold-start, resource orchestration, and serverless at the edge.

Before delving into the literature review, we mention that several concepts in this manuscript have been inspired by the seminal work [8] by Burckhardt et al. which defines in a formal manner a programming model for *durable functions*, i.e., stateful actors whose lifetime spans over multiple executions of the application logic, called *stateful agents* here. In [8], the authors discuss the fundamental reasons why a pure serverless approach may be too limited for some applications or programming patterns, which have led Microsoft

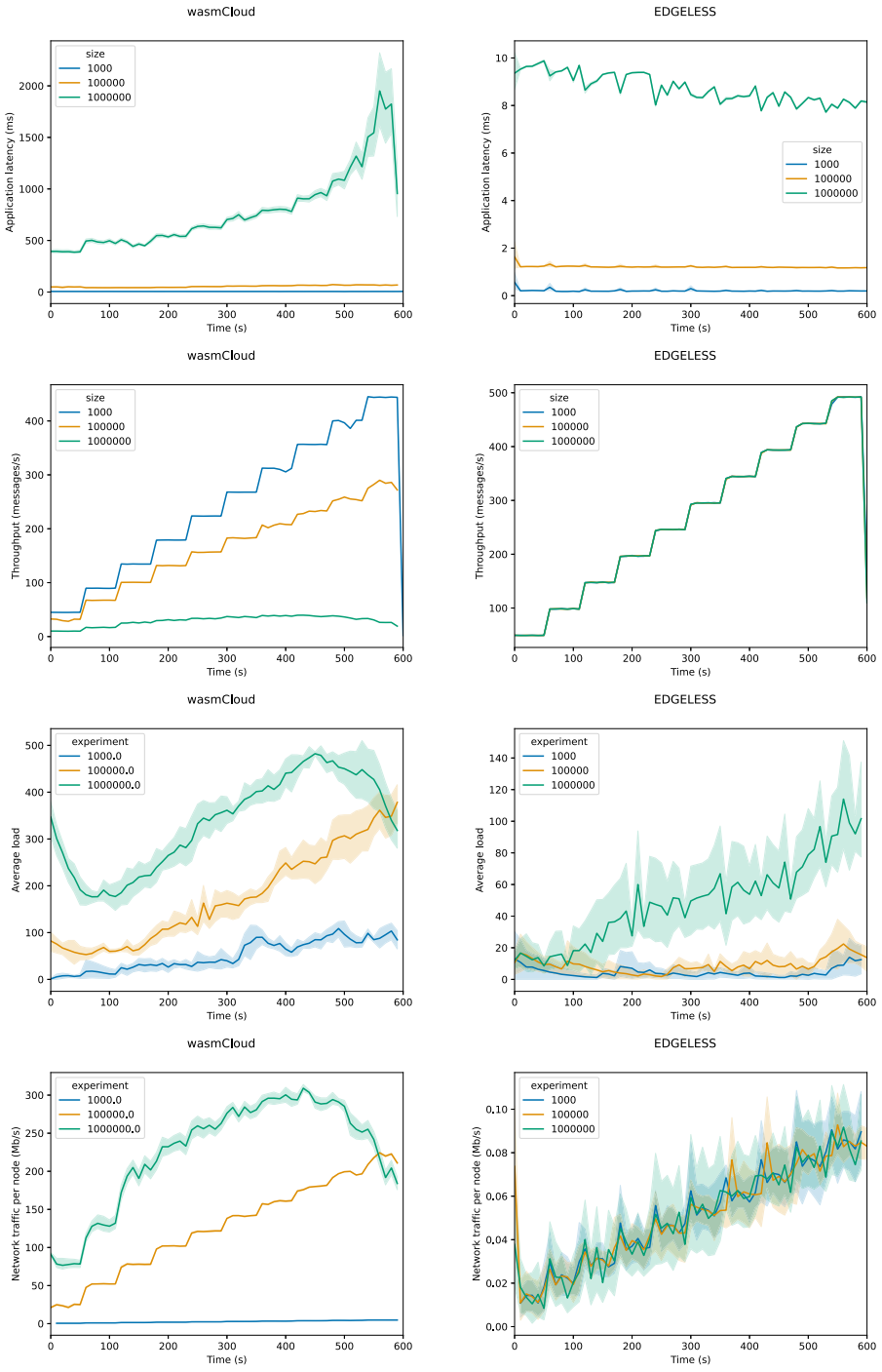


Fig. 20. wasmCloud (left) vs. EDGELESS (right) results. From top to bottom: application latency (ms), throughput (messages/s), node load (1-minute average), total network traffic on nodes (Mb/s). The bands show 95% confidence intervals. *For better readability, the top range of the y-axis in wasmCloud vs. EDGELESS plots is different.*

to implement and offer durable functions as part of their portfolio of cloud computing technologies. For example, durable functions are explicitly allowed to call one another without causing the infamous double-billing issue [38]. Under the hood, durable functions are supported by means of the Durable Task Framework, which has been released by Microsoft as open-source [39], still actively maintained, which supports different storage backends to keep the durable functions' state. The programming model is very flexible and the framework is production-ready, but they have been designed for cloud computing applications and, therefore, do not address specifically the needs and constraints of edge–cloud environments, as we do in this work.

State management. One of the first scientific papers addressing stateful applications in serverless (cloud) platforms is [40], where the authors have proposed a programming model that enriches FaaS with shared-memory primitives; a prototype has been realized with state and synchronization data stored in an Infinispan data grid, which is a high-performance in-memory database. The framework proposed, called Crucial, was designed to assist the migration to the cloud of massively parallel operations, e.g., for big data analysis and scientific applications, without considering edge computing requirements. More recent solutions, instead, have adopted shared logs, e.g., Boki [41] uses a metalog that orders shared log records with high throughput and provides read consistency while allowing serverless platforms to optimize *read* and *read* operations. Halfmoon [42] suggests that it is sufficient to log only *read* or *write* operations, reducing the overheads for serverless applications. Ding et al. [43] propose a toolkit that verifies the idempotence of serverless functions with an improvement in latency and throughput. Beldi [44] is a stateful and transactional runtime for serverless computing, providing fault tolerance and high concurrency. FUYAO [45] enables direct data transfer between functions by decoupling the control flow from data flow, achieving transfer latency at the sub-millisecond level. Wei et al. [46] uses Remote Data Memory Access (RDMA) to enable fast *read* operation and partial state transfer, reducing tail latency by 89%. RDMA is used in the Distributed Shared-Memory Queue (DSMQueue) [47] that builds on Derecho [48] to offer a pub/sub Message-oriented Middleware (MoM) suitable for composition of workflows as interconnected functions to realize business logic and triggers. Nardelli et al. [49] propose a heuristic for function offloading and use an ILP problem for state migration, reducing the response time of serverless functions. *While the function-state and sync-state models (see Section 2.4) are well-understood and easy to realize efficiently, the implementation of the more general shared-state model requires tighter integration between the programming model and the orchestration process. Some important steps have been taken in this direction, but a conclusive solution that suits all cases has not yet been found. In our future work, we will investigate how to integrate individual findings and technology from the literature to maintain their benefits unscathed in an environment of stateful agents.*

Cold-start. Sabre [50], Pronghorn [51], and FaaSnap [52] use snapshots or checkpoints to accelerate the restoration of containers or virtual machines by recording part of the physical memory of a container or microVM into a file for later restoration, reducing cold start-up time. Some other works use function caching to reuse idle containers, mitigating cold start issues. For instance, RainbowCake [53] uses function caching and layer-wise sharing to avoid cold starts. S-Cache [54] and Chen et al. [55] adopt priority-based and opportunistic function caching to reduce the occurrence of cold starts, reducing latency and cost. Pagurus [56] uses an intra-function manager for assigning an idle warm container to be a container that other functions can use without introducing additional security issues, an inter-function scheduler for placing containers between functions, and a load balancer at the cluster level for balancing the workload across different nodes. FaasCache [57] proposes a greedy-dual caching algorithm to reduce the occurrence of cold starts. Recently, Huang et al. have proposed TrEnv [58], which offers repurposable sandboxes that can be shared across different functions to reduce the sandbox setup penalties (latency and overheads). *For our proposed platform, cold-start is less critical than in traditional serverless computing because of the adoption of lightweight virtualization abstractions such as WebAssembly, alongside the workflow-based deployment model. However, frequent migrations and future optimizations might require the adoption of cold-start reducing solutions, which can be plugged in dynamically using the delegated orchestration feature.*

Resource orchestration. Efficient utilization of the available resource while optimizing metrics such as latency, monetary cost, and energy is key to successful adoption. For this reason, the topic has received huge attention in the scientific community, especially through the definition of mathematical optimization models and solvers, which complement our work. Demeter [59] reduces the deployment costs of serverless jobs while considering user-specific SLOs. Shang et al. [60] present a function scheduler that considers edge node heterogeneity and the serverless platform overheads. Pan et al. [61] present a retention-aware scheduler that optimizes request distribution and function retention. Jiagu [62] uses invocation scheduling and a dual-stage scaling technique to improve deployment density while guaranteeing the quality of service. Shahrad et al. [63] characterize serverless workloads and use a histogram-based approach to decide the keep-alive policy for functions. Golgi [64] uses nine low-level metrics to infer resource utilization and function performance in serverless systems. *All these works can be implemented, in principle, as decision-making elements in our platform, fed by the telemetry system and using a delegated orchestration concept to make decisions.*

Serverless at the edge. The emergence of edge computing has fostered discussion among researchers about how to exploit the serverless advantages in this much more diverse and fragmented context compared to cloud computing. Kjorveziroski et al. [65] is an extension to an existing WebAssembly software shim for containerd and a new Kubernetes WebAssembly orchestrator that halves deployment times and reduces the size of artifacts by an order of magnitude. ESMA [66] uses the Egalitarian Stable Matching Algorithm (ESMA) for faster data processing while also considering server resources to create a decentralized serverless edge environment. Cvetkovic et al. [67] report that the cluster manager in Knative incurs over 65% end-to-end latency for cold-start invocations. NEPTUNE [68] is a serverless orchestrator for function placement, CPU and GPU allocation, and resource contention in Multi-access Edge Computing. EneA-FL [69] optimizes the energy consumption in the training process of federated learning while providing seamless interactions between Internet of Things devices and edge nodes. Hudson et al. [70] propose a serverless orchestrator for edge AI service placement and request scheduling, maximizing the quality of service. CodeCrunch [71] uses function compression to alleviate the memory pressure incurred by function retention, reducing the service time by 32% with the same retention budget. Fluidity [72] provides service placement for cloud, edge, and mobile nodes. Fluidity enables developers to use their own deployment and adaptation policies as well as to switch between different policies while the applications are executing. Mahdizadeh et al. [73] propose a serverless workflow orchestrator by constructing a directed acyclic graph to represent a serverless workflow, jointly considering task priorities and workflow execution time. Sun et al. [74] study the problem of task dispatching and bandwidth allocation and use a Pointer Network model to infer task priorities in distributed serverless edge computing. Moakhar et al. [75] model the function orchestration problem as a *School Choice* problem, using Edge FaaS-Top Trading Cycles policy to deploy serverless functions in the edge. Lin et al. [76] offer a targeted heterogeneous runtime unified orchestration solution for

micro-services, predicting runtime types of heterogeneous micro-service with graph neural networks. *With stateful agents, we aim to progress beyond traditional serverless computing and its FaaS model, but research in this area will still provide useful insights on how to drive future evolution of our platform.*

5. Conclusions

In this paper, we have studied the problem of deploying stateful applications on edge–cloud infrastructures leveraging the Function-as-a-Service programming model. We have elaborated on the most critical design issues in this context in the practical context of the EDGELESS open-source platform, where computation nodes can host multiple types of run-time environments (containers, WebAssembly lightweight virtual machines) and resources interconnecting with real-world sources and sinks (sensors, actuators, logs, notification systems, etc.). In particular, we have analyzed the state management aspects and the orchestration problem, which is performed in a hierarchical and distributed manner to maximize scalability and flexibility. Proof-of-concept experiments with heterogeneous resource-constrained edge nodes have been reported to showcase the scalability of management of lightweight stateful agents, the benefits of keeping the state local, and the delegated orchestrator concept with self-healing capabilities.

Despite the volume of work in this space, there remain several open challenges, including: (*i*) investigation of end-to-end latency and reliability with applications sensitive to these metrics; (*ii*) analysis of scalability in terms of workflow throughput and management complexity in diverse edge–cloud deployments; (*iii*) exploitation of the multi-dimensional nature of run-time information collected by the ϵ-ORC to enable sophisticated orchestration policies at both local and cluster levels, potentially using ML/AI; (*iv*) integration of distributed data platforms to enable efficient implementation of shared state management policies for practical use cases; and (*v*) exploration of autoscaling policies for unloading function instances of active but currently idle workflows, trading scalability for cold-start latency.

CRediT authorship contribution statement

Claudio Cicconetti: Writing – original draft, Software, Funding acquisition, Conceptualization. **Emanuele Carlini:** Writing – review & editing, Visualization, Methodology, Investigation, Funding acquisition. **Chen Chen:** Writing – review & editing, Software, Investigation. **Roman Kolcun:** Writing – review & editing, Investigation. **Richard Mortier:** Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Claudio Cicconetti reports financial support was provided by European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU. Emanuele Carlini reports financial support was provided by European Union under the projects EDGELESS. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We thank all the individuals who have joined forces in the EDGELESS project and contributed to defining the operating principles of stateful agents at the edge, and in particular, the most active contributors of the open-source reference implementation, Raphael Hetzel, Francisco Vicente Parra, and Lukasz Zalewski. The work was partially funded by the European Union under the project EDGELESS (GA no. 101092950). The work of C. Cicconetti was funded by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 – program "RESTART", Spoke 1 on "Pervasive and photonic network technologies and infrastructures", project "PESCO – Pervasive communications").

Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.pmcj.2026.102175.

Data availability

We have shared our data in a public GitHub repository.

References

[1] U. Ramachandran, H. Gupta, A. Hall, E. Saurez, Z. Xu, A case for elevating the edge to be a peer of the cloud, GetMobile: Mob. Comput. Commun. 24 (3) (2021) 14–19, http://dx.doi.org/10.1145/3447853.3447859.

[2] IoT Edge Working Group, Edge native application principles white paper, 2023, URL https://www.cncf.io/blog/2023/03/09/introducing-the-edge-native-whitepaper/.

[3] A. Hazra, A. Kalita, M. Gurusamy, Meeting the requirements of internet of things: The promise of edge computing, IEEE Internet Things J. 11 (5) (2024) 7474–7498, http://dx.doi.org/10.1109/JIOT.2023.3339492.

[4] P. Raith, S. Nastic, S. Dustdar, Serverless edge computing—where we are and what Lies ahead, IEEE Internet Comput. 27 (3) (2023) 50–64, http://dx.doi.org/10.1109/MIC.2023.3260939.

[5] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, M. Picone, Exploiting microservices and serverless for digital twins in the cloud-to-edge continuum, Future Gener. Comput. Syst. 157 (2024) 275–287, http://dx.doi.org/10.1016/j.future.2024.03.052, URL https://www.sciencedirect.com/science/article/pii/S0167739X24001249.

[6] M.S. Aslanpour, A.N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S.S. Gill, R. Gaire, S. Dustdar, Serverless edge computing: Vision and challenges, in: Proceedings of the 2021 Australasian Computer Science Week Multiconference, ACSW'21, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3437378.3444367.

[7] J.J.L. Escobar, F. Gil-Castiñeira, R.P. Díaz Redondo, Decentralized serverless IoT dataflow architecture for the cloud-to-edge continuum, in: 2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN, 2023, pp. 42–49, http://dx.doi.org/10.1109/ICIN56760.2023.10073502.

[8] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, C.S. Meiklejohn, Durable functions: semantics for stateful serverless, Proc. ACM Program. Lang. 5 (OOPSLA) (2021) http://dx.doi.org/10.1145/3485510.

[9] V.M. Bhasi, J.R. Gunasekaran, P. Thinakaran, C.S. Mishra, M.T. Kandemir, C. Das, Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms, in: Proceedings of the ACM Symposium on Cloud Computing, ACM, New York, NY, USA, 2021, pp. 153–167, http://dx.doi.org/10.1145/3472883.3486992, URL https://dl.acm.org/doi/10.1145/3472883.3486992.

[10] Kubernetes, 2026, https://kubernetes.io/. (Accessed 09 January 2026).

[11] A. Jeffery, H. Howard, R. Mortier, Rearchitecting kubernetes for the edge, in: Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, EdgeSys'21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 7–12, http://dx.doi.org/10.1145/3434770.3459730.

[12] S. Risco, C. Alarcón, S. Langarita, M. Caballer, G. Moltó, Rescheduling serverless workloads across the cloud-to-edge continuum, Future Gener. Comput. Syst. 153 (2024) 457–466, http://dx.doi.org/10.1016/j.future.2023.12.015, URL https://www.sciencedirect.com/science/article/pii/S0167739X23004764.

[13] J. Spenger, P. Carbone, P. Haller, A survey of actor-like programming models for serverless computing, in: F. De Boer, F. Damiani, R. Hähnle, E. Broch Johnsen, E. Kamburjan (Eds.), Active Object Languages: Current Research Trends, vol. 14360, Springer Nature Switzerland, Cham, 2024, pp. 123–146.

[14] WebAssembly, 2026, https://webassembly.org/. (Accessed 09 January 2026).

[15] S. Shillaker, P. Pietzuch, FAASM: Lightweight isolation for efficient stateful serverless computing, in: Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020, 2020, pp. 419–433.

[16] P.K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, G. Parmer, Sledge: a serverless-first, light-weight wasm runtime for the edge, in: Proceedings of the 21st International Middleware Conference, Middleware'20, 2020, pp. 265–279, http://dx.doi.org/10.1145/3423211.3425680, URL https://dl.acm.org/doi/10.1145/3423211.3425680.

[17] P. Gackstatter, P.A. Frangoudis, S. Dustdar, Pushing serverless to the edge with WebAssembly runtimes, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid, 2022, pp. 140–149, http://dx.doi.org/10.1109/CCGrid54584.2022.00023, URL https://ieeexplore.ieee.org/document/9826054.

[18] EDGELESS project website, 2026, https://edgeless-project.eu/. (Accessed 09 January 2026).

[19] EDGELESS reference implementation – Core modules, 2026, https://github.com/edgeless-project/edgeless/. (Accessed 09 January 2026).

[20] Repository of experiment scripts and artifacts, 2026, https://github.com/edgeless-project/cnr-experiments/. (Accessed 09 January 2026).

[21] C. Cicconetti, E. Carlini, A. Paradell, EDGELESS project: On the road to serverless edge AI, in: Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge, FRAME'23, 2023, pp. 41–43, http://dx.doi.org/10.1145/3589010.3594890.

[22] C. Cicconetti, E. Carlini, R. Hetzel, R. Mortier, A. Paradell, M. Sauer, EDGELESS: A software architecture for stateful FaaS at the edge, in: Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'24, 2024, pp. 393–396, http://dx.doi.org/10.1145/3625549.3658817.

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

[24] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, J. Ott, Oakestra: A lightweight hierarchical orchestration framework for edge computing, in: Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC'23, 2023, pp. 215–231, URL https://www.usenix.org/conference/atc23/presentation/bartolomeo.

[25] I. Baldini, P. Cheng, S.J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu, The serverless trilemma: Function composition for serverless computing, in: Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2017, 2017, pp. 89–103, http://dx.doi.org/10.1145/3133850.3133855.

[26] J. Park, S. Kang, S. Lee, T. Kim, J. Park, Y. Kwon, J. Huh, Hardware-hardened sandbox enclaves for trusted serverless computing, ACM Trans. Arch. Code Optim. 21 (1) (2024) http://dx.doi.org/10.1145/3632954.

[27] K. Krauth, U.C. Berkeley, B. Recht, U.C. Berkeley, J. Ragan-kelley, Serverless linear algebra, in: ACM SoCC, 2020, pp. 281–295.

[28] C. Cicconetti, M. Conti, A. Passarella, FaaS execution models for edge applications, Pervasive Mob. Comput. 86 (2022) 101689, http://dx.doi.org/10.1016/j.pmcj.2022.101689, URL https://www.sciencedirect.com/science/article/pii/S157411922200102X.

[29] D. Ongaro, J. Ousterhout, The raft consensus algorithm, Lect. Not. CS 190 (2015) 2022.

[30] L. Lamport, Generalized consensus and paxos, 2005, https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/.

[31] M. Yu, T. Cao, W. Wang, R. Chen, Pheromone: Restructuring serverless computing with data-centric function orchestration, IEEE Trans. Netw. 33 (1) (2025) 226–240, http://dx.doi.org/10.1109/TNET.2024.3480031, URL https://ieeexplore.ieee.org/document/10770575/.

[32] D. Carrizales-Espinoza, D.D. Sanchez-Gallegos, J. Gonzalez-Compean, J. Carretero, StructMesh: A storage framework for serverless computing continuum, Future Gener. Comput. Syst. 159 (2024) 353–369, http://dx.doi.org/10.1016/j.future.2024.05.033, URL https://linkinghub.elsevier.com/retrieve/pii/S0167739X24002401.

[33] Rust programming language, 2026, https://www.rust-lang.org/. (Accessed 09 January 2026).

[34] gRPC – A high performance, open source universal RPC framework, 2026, https://grpc.io/. (Accessed 09 January 2026).

[35] J. Dobies, J. Wood, Kubernetes Operators: Automating the Container Orchestration Platform, first ed., O'Reilly, Beijing Boston Farnham Sebastopol Tokyo, 2020.

[36] wasmCloud – wasm-native orchestration, 2026, https://wasmcloud.com/. (Accessed 09 January 2026).

[37] nginx, 2026, https://nginx.org/. (Accessed 09 January 2026).

[38] Durable functions billing, 2026, https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-billing. (Accessed 09 January 2026).

[39] Durable task framework, 2026, https://github.com/Azure/durabletask. (Accessed 09 January 2026).

[40] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, P. García-López, On the FaaS track: Building stateful distributed applications with serverless architectures, in: Proceedings of the 20th International Middleware Conference, ACM, Davis CA USA, 2019, pp. 41–54, http://dx.doi.org/10.1145/3361525.3361535, URL https://dl.acm.org/doi/10.1145/3361525.3361535.

[41] Z. Jia, E. Witchel, Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices, in: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'21, 2021, pp. 152–166, http://dx.doi.org/10.1145/3445814.3446701.

[42] S. Qi, X. Liu, X. Jin, Halfmoon: Log-optimal fault-tolerant stateful serverless computing, in: Proceedings of the 29th Symposium on Operating Systems Principles, SOSP'23, 2023, pp. 314–330, http://dx.doi.org/10.1145/3600006.3613154.

[43] H. Ding, Z. Wang, Z. Shen, R. Chen, H. Chen, Automated verification of idempotence for stateful serverless applications, in: 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 23, 2023, pp. 887–910, URL https://www.usenix.org/conference/osdi23/presentation/ding.

[44] C. Zhang, H. Tan, H. Huang, Z. Han, S.H. Jiang, N. Freris, X.Y. Li, Online dispatching and scheduling of jobs with heterogeneous utilities in edge computing, in: Proceedings of the international symposium on mobile ad hoc networking and computing, MobiHoc, 2020, pp. 101–110, http://dx.doi.org/10.1145/3397166.3409122.

[45] G. Liu, L. Zhao, Y. Li, Z. Duan, S. Chen, Y. Hu, Z. Su, W. Qu, FUYAO: DPU-enabled direct data transfer for serverless computing, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 3, ASPLOS'24, 2024, pp. 431–447, http://dx.doi.org/10.1145/3620666.3651327.

[46] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, H. Chen, No provisioned concurrency: Fast RDMA-co-designed remote fork for serverless computing, in: 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 23, 2023, pp. 497–517, URL https://www.usenix.org/conference/osdi23/presentation/wei-rdma.

[47] A. Sabbioni, L. Rosa, A. Bujari, L. Foschini, A. Corradi, DIFFUSE: A distributed and decentralized platform enabling function composition in serverless environments, Comput. Netw. 210 (2022) 108993, http://dx.doi.org/10.1016/j.comnet.2022.108993, URL https://www.sciencedirect.com/science/article/pii/S138912862200161X.

[48] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R.V. Renesse, S. Zink, K.P. Birman, Derecho: Fast state machine replication for cloud services, ACM Trans. Comput. Syst. 36 (2) (2019) http://dx.doi.org/10.1145/3302258.

[49] M. Nardelli, G. Russo Russo, Function offloading and data migration for stateful serverless edge computing, in: Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, 2024, pp. 247–257, http://dx.doi.org/10.1145/3629526.3649293, URL https://dl.acm.org/doi/10.1145/3629526.3649293.

[50] N. Lazarev, V. Gohil, J. Tsai, A. Anderson, B. Chitlur, Z. Zhang, C. Delimitrou, Sabre: Hardware-accelerated snapshot compression for serverless MicroVMs, in: 18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 24, USENIX Association, Santa Clara, CA, 2024, pp. 1–18, URL https://www.usenix.org/conference/osdi24/presentation/lazarev.

[51] S. Kohli, S. Kharbanda, R. Bruno, J. Carreira, P. Fonseca, Pronghorn: Effective checkpoint orchestration for serverless hot-starts, in: Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys'24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 298–316.

[52] L. Ao, G. Porter, G.M. Voelker, FaaSnap: FaaS made fast using snapshot-based VMs, in: Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys'22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 730–746.

[53] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, S.-J. Park, RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 1, ASPLOS'24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 335–350.

[54] C. Chen, L. Nagel, L. Cui, F.P. Tso, S-Cache: Function caching for serverless edge computing, in: Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking, EdgeSys'23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–6.

[55] C. Chen, M. Herrera, G. Zheng, L. Xia, Z. Ling, J. Wang, Cross-edge orchestration of serverless functions with probabilistic caching, IEEE Trans. Serv. Comput. 17 (5) (2024) 2139–2150, http://dx.doi.org/10.1109/TSC.2024.3399651.

[56] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, M. Guo, Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing, in: 2022 USENIX Annual Technical Conference, USENIX ATC 22, USENIX Association, Carlsbad, CA, 2022, pp. 69–84, URL https://www.usenix.org/conference/atc22/presentation/li-zijun-help.

[57] A. Fuerst, P. Sharma, FaasCache: Keeping serverless computing alive with greedy-dual caching, in: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'21, 2021.

[58] J. Huang, M. Zhang, T. Ma, Z. Liu, S. Lin, K. Chen, J. Jiang, X. Liao, Y. Shan, N. Zhang, M. Lu, T. Ma, H. Gong, Y. Wu, TrEnv: Transparently share serverless execution environments across different functions and nodes, in: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, ACM, Austin TX USA, 2024, pp. 421–437, http://dx.doi.org/10.1145/3694715.3695967, URL https://dl.acm.org/doi/10.1145/3694715.3695967.

[59] X. Yue, S. Yang, L. Zhu, S. Trajanovski, X. Fu, Demeter: Fine-grained function orchestration for geo-distributed serverless analytics, in: IEEE INFOCOM 2024 - IEEE Conference on Computer Communications, 2024, pp. 2498–2507, http://dx.doi.org/10.1109/INFOCOM52122.2024.10621303.

[60] X. Shang, Y. Mao, Y. Liu, Y. Huang, Z. Liu, Y. Yang, Online container scheduling for data-intensive applications in serverless edge computing, in: IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, 2023, pp. 1–10, http://dx.doi.org/10.1109/INFOCOM53939.2023.10229034.

[61] L. Pan, L. Wang, S. Chen, F. Liu, Retention-aware container caching for serverless edge computing, in: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, 2022, pp. 1069–1078.

[62] Q. Liu, Y. Yang, D. Du, Y. Xia, P. Zhang, J. Feng, J.R. Larus, H. Chen, Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with jiagu, in: 2024 USENIX Annual Technical Conference, USENIX ATC 24, USENIX Association, Santa Clara, CA, 2024, pp. 1–17, URL https://www.usenix.org/conference/atc24/presentation/liu-qingyuan.

[63] M. Shahrad, R. Fonseca, Í.n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, in: Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020, 2020, pp. 205–218.

[64] S. Li, W. Wang, J. Yang, G. Chen, D. Lu, Golgi: Performance-aware, resource-efficient function scheduling for serverless computing, in: Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC'23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 32–47.

[65] V. Kjorveziroski, S. Filiposka, WebAssembly orchestration in the context of serverless computing, J. Netw. Syst. Manage. 31 (3) (2023) 62, http://dx.doi.org/10.1007/s10922-023-09753-0.

[66] S. Datta, S.K. Addya, S.K. Ghosh, ESMA: Towards elevating system happiness in a decentralized serverless edge computing framework, J. Parallel Distrib. Comput. (2023) 104762, http://dx.doi.org/10.1016/j.jpdc.2023.104762, URL https://www.sciencedirect.com/science/article/pii/S0743731523001326.

[67] L. Cvetković, R. Fonseca, A. Klimovic, Understanding the neglected cost of serverless cluster management, in: Proceedings of the 4th Workshop on Resource Disaggregation and Serverless, ACM, Koblenz Germany, 2023, pp. 22–28, http://dx.doi.org/10.1145/3605181.3626286, URL https://dl.acm.org/doi/10.1145/3605181.3626286.

[68] L. Baresi, D.Y.X. Hu, G. Quattrocchi, L. Terracciano, NEPTUNE: a comprehensive framework for managing serverless functions at the edge, ACM Trans. Auton. Adapt. Syst. (2023) http://dx.doi.org/10.1145/3634750, URL https://dl.acm.org/doi/10.1145/3634750.

[69] A. Agiollo, P. Bellavista, M. Mendula, A. Omicini, EneA-FL: Energy-aware orchestration for serverless federated learning, Future Gener. Comput. Syst. 154 (2024) 219–234, http://dx.doi.org/10.1016/j.future.2024.01.007, URL https://www.sciencedirect.com/science/article/pii/S0167739X24000074.

[70] N. Hudson, H. Khamfroush, M. Baughman, D.E. Lucani, K. Chard, I. Foster, QoS-aware edge AI placement and scheduling with multiple implementations in FaaS-based edge computing, Future Gener. Comput. Syst. 157 (2024) 250–263, http://dx.doi.org/10.1016/j.future.2024.03.035, URL https://www.sciencedirect.com/science/article/pii/S0167739X24001067.

[71] R. Basu Roy, T. Patel, R. Garg, D. Tiwari, CodeCrunch: Improving serverless performance via function compression and cost-aware warmup location optimization, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 1, ACM, La Jolla CA USA, 2024, pp. 85–101, http://dx.doi.org/10.1145/3617232.3624866, URL https://dl.acm.org/doi/10.1145/3617232.3624866.

[72] F. Pournaropoulos, A. Patras, C.D. Antonopoulos, N. Bellas, S. Lalis, Fluidity: Providing flexible deployment and adaptation policy experimentation for serverless and distributed applications spanning cloud-edge-mobile environments, Future Gener. Comput. Syst. (2024) http://dx.doi.org/10.1016/j.future.2024.03.031, URL https://www.sciencedirect.com/science/article/pii/S0167739X24000980.

[73] S.H. Mahdizadeh, S. Abrishami, An assignment mechanism for workflow scheduling in function as a service edge environment, Future Gener. Comput. Syst. 157 (2024) 543–557, http://dx.doi.org/10.1016/j.future.2024.04.003, URL https://linkinghub.elsevier.com/retrieve/pii/S0167739X24001328.

[74] Y. Sun, C. Zhang, T. Huang, Joint task dispatching and bandwidth allocation with hard deadlines in distributed serverless edge computing systems, J. Grid Comput. 22 (2) (2024) 51, http://dx.doi.org/10.1007/s10723-024-09770-6.

[75] S.P. Moakhar, S. Abrishami, An efficient mechanism for function scheduling and placement in function as a service edge environment, J. Netw. Comput. Appl. 226 (2024) 103890, http://dx.doi.org/10.1016/j.jnca.2024.103890, URL https://linkinghub.elsevier.com/retrieve/pii/S1084804524000675.

[76] Y. Lin, B. Feng, Z. Ding, Context-aware runtime type prediction for heterogeneous microservices, in: J. Carretero, S. Shende, J. Garcia-Blas, I. Brandic, K. Olcoz, M. Schreiber (Eds.), Euro-Par 2024: Parallel Processing, vol. 14801, Springer Nature Switzerland, Cham, 2024, pp. 329–342.