Towards Implementing Distributed Custom Serverless Function Scheduling in FunLess*

Giuseppe De Palma[1,2], Saverio Giallorenzo[1,2], Jacopo Mauro[3], Matteo Trentin[1,2,3], and Gianluigi Zavattaro[1,2]

[1] Università di Bologna, Italy {giuseppe.depalma2,saverio.giallorenzo2, matteo.trentin2,gianluigi.zavattaro}@unibo.it
[2] OLAS research team, INRIA, France
[3] University of Southern Denmark, Denmark mauro@imada.sdu.dk

Abstract. APP is a declarative language for the definition of custom function scheduling on the worker nodes available in serverless Function as a Service (FaaS) platforms. Current APP implementations assume a central control point that users can access to issue the execution of functions. We propose an extension of APP's implementation to allow for multiple control points, tackling both scaling and resilience issues of existing implementations. To substantiate our proposal, we present an implementation of our extension using the FunLess FaaS platform, tailored for private edge-cloud and multi-cloud environments. We show initial experiments that indicate performance improvements in setups where both the platform and function invocations are spread across multiple locations.

Keywords: Serverless, Function Scheduling, Decentralisation

1 Introduction

This paper has been devised for the Jean-Bernard Stefani's festschrift, as it presents research conducted within our group that aligns closely with his interests. Jean-Bernard has long been an advocate for innovative approaches to the development of distributed software systems, particularly those grounded in modular and compositional principles. Recent trends in modular and compositional distributed software developments tend to minimise the complexity of the components and limit (and possibly avoid) component's interdependencies. This property allows components in the system to autonomously scale in order to make the entire system more resilient. However, as these components scale, the complexity in the management of the overall system, including its computing

* This work has been partially supported by the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), RTM&R (CUP: J33C22001170001) funded by the MUR National Recovery and Resilience Plan (European Union - NextGenerationEU) and the French ANR project SmartCloud ANR-23-CE25-0012.

infrastructure, also increases, creating the need for efficient — ideally automatic — management systems. Serverless computing emerged to address these challenges by offering a model that abstracts away the underlying infrastructure, allowing developers to build applications as compositions of stateless (hence, less complex), event-driven functions, determining the paradigm of Function-as-a-Service (FaaS), that a platform can automatically scale up and down based on demand.

The language of Allocation Priority Policies (APP) (and its variants) emerged as a declarative solution allowing developers to define custom function scheduling policies and placement constraints across worker nodes in serverless platforms [16,9,10,8,11,13,7,15]. Thanks to APP, developers can express fine-grained placement preferences, enabling improved execution performance on diverse criteria, including resource requirements and locality principles. This declarative approach abstracts away the complexities of infrastructure management while giving users control over critical aspects of function placement. For instance, developers can ensure compute-intensive functions run on nodes with specific hardware accelerators, co-locate functions that frequently communicate to reduce latency, or implement compliance requirements by restricting certain functions to specific geographical regions.

Being a platform-agnostic solution, APP-based scheduling can be implemented on multiple serverless platforms, like it has been done for Apache OpenWhisk [23,16] and FunLess [12,7]. Similarly to other open-source alternatives, like OpenFaaS [22], and KNative [1], these platforms typically assume a centralised controller that handles the scheduling of functions. Indeed, serverless platform architectures typically consist of two main components: *controllers* and *workers*. Workers constitute the majority of FaaS platform deployment, as these nodes are responsible for executing the functions. The prevalent deployment model features a single central controller that manages the scheduling and execution of functions across the available workers. This controller acts as the brain of the system: it receives invocation requests, keeps track of system state, and assigns work (i.e., the execution of functions) to the available worker nodes.

Problem The centrality of the controller in APP-based implementations is paramount: it is the component that realises the semantics of APP-governed scheduling, working as the single entry point of a serverless architecture and having complete control over the functions executing on the workers — it maintains their status and constantly knows which functions run on which workers at any time. While this design simplifies coordination and is efficient at small and moderate scales, it poses limitations as deployments grow. The central controller can become a performance bottleneck and a single point of failure, especially under heavy load or in geographically distributed environments.

The need for multiple controllers arises primarily from scalability, locality, and fault-tolerance concerns. As serverless deployments increase in size — both in terms of number of functions and the geographic spread of infrastructure — a single controller can hardly satisfy the number and quality of service of user requests. The controller must handle a growing volume of requests, maintaining a global view of a large and dynamic system and managing communication with

all nodes. This scalability issue can lead to degraded performance and increased latency, particularly when the controller becomes overwhelmed or is physically distant from parts of the system it manages. Moreover, the larger the geographical distribution of users that access the system, the more relevant the problem of having a controller at a single location becomes — e.g., the farther the users are from the controller, the higher is their experienced interaction latency with the system. Regarding faults, having a single coordinator for a given deployment means that all user interactions can fail due to a controller's failure.

Distributing the control logic across multiple controllers addresses these challenges, enhancing scalability, reducing latency, and increasing the overall robustness of the platform. Decentralisation enhances scalability by providing multiple access points to the platform and improves fault-tolerance by eliminating a single point of failure — if one controller fails, only the function requests managed by that controller fail, while users of other, healthy controllers remain unaffected and can continue their operations. In geographically-distributed scenarios, users have the option of interacting with the controller closest to them, reducing latency and improving responsiveness. These aspects are especially critical in IoT and edge computing environments, where constraints such as geographic locality, limited resources, and intermittent connectivity are common.

Contribution In this paper, we propose an extension of APP's architectural implementation to allow for multiple control points, thereby addressing both scaling, locality, and resilience issues of existing implementations. By enabling decentralised, APP-based scheduling, each controller shall operate autonomously while still respecting the application-level constraints expressed via APP. Indeed, although having multiple controllers solves the mentioned issues, introducing decentralised control in APP-based platforms brings in a new set of challenges, particularly in maintaining distributed state consistency. When scheduling, each controller must maintain an accurate and coherent view of the system's state while respecting APP policies. We explore the architectural and protocol-level modifications needed to make decentralised control possible, and we discuss the key trade-offs between maintaining consistency and achieving responsiveness.

To substantiate our proposal, we present an implementation based on FunLess — a serverless platform tailored for private edge-cloud and multi-cloud environments. Through our implementation and experiments, we demonstrate performance improvements in setups where both the platform and function invocations are spread across multiple locations. Our results indicate that decentralising APP control points improves overall system performance, particularly in geographically distributed deployments.

2 Background

Before presenting our proposal, we briefly overview the FaaS paradigm, the typical architecture of FaaS platforms, APP, and the design of FunLess.

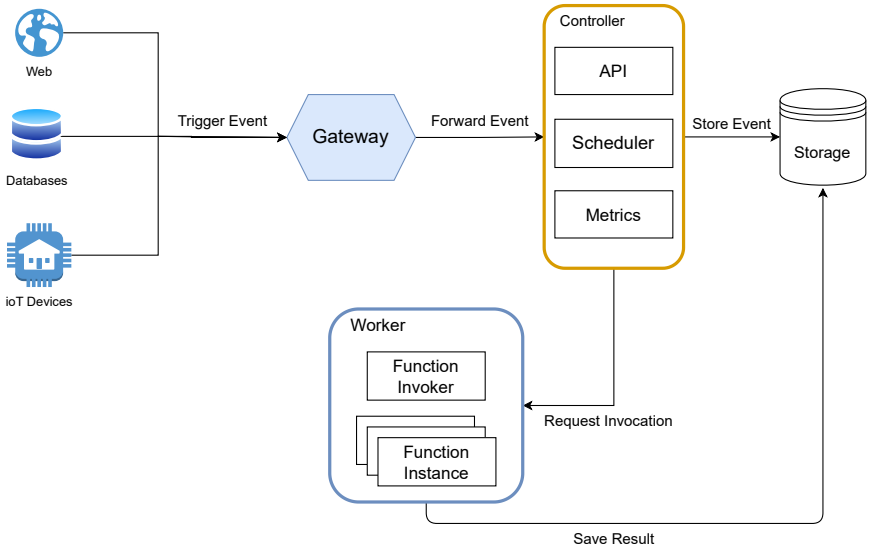

Fig. 1. A typical serverless platform architecture.

2.1 Serverless and Functions-as-a-Service

A serverless application emerges from the combination of software units called functions, which run in short-lived environments, triggered by events — such as HTTP requests, database updates, file uploads, and scheduled intervals. When an event triggers a function execution, the FaaS platform runs the code after initialising an execution environment — a secure, isolated context that provides the resources for the function lifecycle. FaaS platforms mainly use virtual machines and containers to implement portable, isolated function execution environments.

Serverless platforms generally follow the (simplified) architecture design reported in Fig. 1. As visible in the figure, the main components of a FaaS platform's architecture are the controllers and the workers. The controller receives requests from external sources, such as users or other systems, and it orchestrates the allocation of functions on the available worker nodes. In particular, the scheduler controller's component determines which worker should execute a function, based on factors such as the worker's current load, the function requirements, and resource availability. Upon receiving a function-execution request, the targeted worker executes the function, handling the function's execution environment lifecycle, including provisioning, scaling, and teardown.

Serverless platforms usually adopt a communication layer that facilitates communication between the controller node and worker nodes, handling messages and data transfer between components — omitted, in Fig. 1, for clarity. In particular, message queues or event brokers (e.g., RabbitMQ [5], Kafka [2])

support asynchronous communication between components, allowing decoupling and scalability. Internal APIs support synchronous communication for tasks such as function deployment, status updates, and resource allocation. Monitoring tools also appear in these architectures to collect metrics on resource usage, function execution times, and error rates. Metrics provide visibility into system performance, function execution, and overall health and enable debugging, troubleshooting, and performance optimisation.

Among the leading providers of serverless computing platforms, Amazon Web Services (AWS) Lambda [24] stands out as a pioneer in the field. AWS Lambda was the first publicly available serverless platform, allowing developers to pay only for the compute time consumed by their functions. Briefly thereafter, other platforms followed suit, such as Microsoft Azure Cloud Functions [3] and Google Cloud Platform (GCP) Cloud Functions [6]. A number of open-source serverless platforms have also emerged, such as OpenWhisk [23], Knative [1], and OpenFaaS [22]. One can deploy these platforms on-premises or on the cloud, as a more flexible and customisable solution compared to the proprietary ones.

2.2 Allocation Priority Policies (APP)

The scheduling of functions, i.e., which worker, among the available ones, executes a given function, can substantially influence their performance. Indeed, effects like *code locality* [18] — due to latencies in loading function code and runtimes — or *session locality* [18] — due to the need to authenticate and open new sessions to interact with other services — can substantially increase the run time of functions. Usually, serverless platforms implement opinionated policies that favour some performance principle tailored for one or more of these locality principles. This shortcoming motivated De Palma et al. [17] to introduce a YAML-like declarative language used to specify scheduling policies to govern the allocation of serverless functions on the nodes that make up a cluster, called APP. Thanks to APP, the same platform can support different scheduling policies, each tailored to meet the specific needs of a set of related functions.

To define function-specific policies, APP assumes the association of each function with a tag. In our examples, we directly use the function's reference name as the tag, but the relation can be one-to-many, to specify a policy shared among a set of functions. Then, APP associates a tag to a policy, so that, at runtime, the scheduler of the platform can pair each function with its APP policy and follow the latter's scheduling logic.

We show an example APP script, illustrating one policy in Listing 1.1. In an APP script, users can specify a sequence of blocks — each identified by YAML's list unit `-` — associated with a tag. In the example, the policy tag is `f`, at line 1. Each block indicates on which workers the scheduler can allocate the function. At function invocation, the scheduler tries to allocate the function following the logic in the first block, passing to the next only if none of the workers specified in that block can host the function, and so on. Exhausting all blocks causes the invocation's failure. In APP, `workers` is the keyword used in the scripts to specify the label of the worker nodes available to that block; in Listing 1.1,

```
1 f:
2   - workers: [ w1, w2, w3 ]
3     strategy: best_first
4     invalidate: max_concurrent_invocations: 10
5   - workers: *
6     strategy: random
7     invalidate: capacity_used: 80
```

Listing 1.1. APP script used for the tests.

the workers labelled `w1`, `w2`, and `w3` are the ones specified in the first block, at line 2; the universal `*` at line 5 indicates the selection of all available workers in a given deployment. Besides `workers`, APP lets users specify the `strategy` the scheduler shall follow to select among the indicated workers and when to `invalidate` a worker, which would not be able to execute the function under scheduling. Examples of strategies are `random`, to chose uniformly at random among the workers in a block, e.g., for load-balancing (line 6) and `best_first`, to follow a top-to-bottom ordering (line 3), e.g., to indicate the workers from the most to the least powerful. The invalidation constraints can, for example, set a maximal threshold of concurrent functions running on a worker (line 4) or define a maximum number of resources (cpu, memory) occupied by other functions on the worker (line 7).

2.3 The FunLess Serverless Platform

FunLess [12,14] is a serverless platform for private edge-cloud and multi-cloud environments that consists of mainly two components: the Core and the Worker. The Core corresponds to the controller and acts as a user-facing API to *i*) create, fetch, update, and delete functions and *ii*) schedule functions on workers. The Worker is the component deployed on every node tasked to run the functions. Besides Core and Workers, FunLess includes a Postgres database, to store functions and metadata, and Prometheus, to manage the metrics of the platform.[4] Both main components are written in Elixir [19], and take advantage of the BEAM [25] message passing model to communicate with each other.

The Core controls the platform, exposing an HTTP REST API for user interaction, handling authentication and authorisation, and managing functions' lifecycle and invocations. The Core can automatically discover Workers within the same network employing the Multicast UDP Gossip algorithm for bare-metal deployments and Kubernetes' service discovery for containerised environments. Functionality-wise, users create functions by compiling source code to WebAssembly and uploading the binary to the Core, which stores it in the database with a name. Users can group functions in modules and specify memory requirements for function execution. When an invocation request is received, the Core selects a

[4] Resp. at https://www.postgresql.org/ and https://prometheus.io.

suitable Worker to run the function, to which it then sends the request parameters. In case the Worker does not have a function's binary, it returns an error to the Core, which send said binary alongside the request.

The Worker runs functions via Wasmtime, a security-oriented runtime for WebAssembly. Workers have a local cache with a configurable size limit, where they store recently run binaries, to minimise network traffic during function invocations. Upon reception of an invocation request, the Worker checks its cache for the function's binary, returning an error as mentioned above. After an invocation, the Worker returns its result (successful or not) to the Core.

3 From Centralised to Decentralised APP Scheduling

Usually, open-source serverless platforms employ centralised function scheduling, where a single controller manages the scheduling of functions across worker nodes. In this model, the controller maintains comprehensive knowledge of all workers' states — the only divergence between the status of a worker and the knowledge the controller has is that the worker can become unreachable — and workers are primarily passive components that execute assigned functions and report results or timeouts. APP-based platform implementations particularly rely on the centrality of the controller to ensure the satisfaction of the specified function scheduling constraints found in APP scripts.

While effective at smaller scales, this centralised approach faces significant challenges when scaling to larger, geographically distributed deployments. This section explores the transition from centralised to decentralised function scheduling, examining different decentralisation strategies, their underlying challenges, and their respective trade-offs.

3.1 Challenges in Decentralised Scheduling

Moving from a centralised to a decentralised scheduling architecture introduces several fundamental challenges. First, maintaining state consistency becomes complex as multiple controllers must hold a coherent view of the system state, including worker availability, resource utilisation, and function execution status. Second, coordination overhead emerges when controllers must synchronise scheduling decisions to avoid conflicts, introducing additional communication and potential latency. Third, network limitations, particularly in edge computing scenarios, mean the bandwidth and latency costs of coordination messages can significantly impact performance.

We identify two primary approaches to decentralising serverless scheduling architectures: transaction-based decentralisation and optimistic decentralisation. Each approach offers distinct advantages and limitations.

Transaction-based Decentralisation In transaction-based centralisation, each controller maintains a local view of the global system state, and distributed transactions are used to ensure consistency across controllers. Every scheduling

decision involves a coordinated update to these local views, guaranteeing that all controllers operate on a consistent snapshot of the system. While this method provides strong consistency guarantees, it is also communication-intensive, as each scheduling operation incurs the overhead of a distributed coordination protocol.

Within this context, two main coordination strategies emerge: distributed transactions and leader election. In both cases, the systems in place ensure that controllers maintain a consistent view of worker states and avoid race conditions between controllers during function scheduling, which could generate a misalignment between the knowledge of the controllers and the state of the workers. Thus, before proceeding with the actual scheduling, controllers must reach consensus on the scheduling decisions. In this case, workers maintain their passive role as in centralised architectures, as transactions entail global state management.

Distributed transactions provide a mechanism to ensure consistent state across controllers. When applied to function scheduling, these transactions enable multiple controllers to coordinate their view of worker availability and resource allocation. One foundational protocol in this domain is two-phase commit [4], which operates in phases: first, a controller sends a "proposal" request to the other controllers; each controller validates the proposed scheduling action against its local state and votes to commit or abort; finally, the initial controller collects all votes and issues either a global commit or abort instruction. This approach guarantees that all controllers either collectively apply or collectively reject a scheduling decision.

As an alternative, leader-election protocols, such as PAXOS [20] and RAFT [21], can reduce the number of messages needed to issue a scheduling instruction by having an elected leader among the controllers govern the global state and regulate scheduling requests from the other controllers, usually called "followers". In practice, followers inform the leader when they receive a request from a user (since the leader maintains the state of the system, requests to the leader can proceed directly, informing the followers of the scheduling decision it takes) to schedule a function, and the leader directs the follower on how to proceed (where to schedule the function or whether the scheduling fails, e.g., because no worker has enough capacity to run that function).

Both approaches become problematic with large-distance topologies, where transaction-based scheduling, leader election and coordination may incur high latency — imagine that a controller with high communication latency becomes the leader, with which all other controllers have to coordinate to schedule their functions. This aspect is particularly relevant in constrained network environments, such as edge computing, where the costs of these messages can be significant — in terms of energy (edge devices may rely on batteries to power them), bandwidth, latency and, ultimately, scheduling time.

Another limitation of this approach arises when a controller is already a bottleneck for incoming scheduling requests. In such cases, enforcing coordination through distributed transactions or relying on a single leader to govern scheduling decisions may exacerbate the problem. The overhead introduced by coordination

protocols can further delay responses, compounding the controller's inability to handle requests in a timely manner. This aspect is especially critical in high-load scenarios, where even minor delays can lead to significant degradation in system responsiveness or throughput. Practically, the additional communication and synchronisation steps required by this approach can undermine scalability and reactivity.

Optimistic Decentralisation To avoid the overhead of distributed transactions, one can pursue an "optimistic" approach, where controllers attempt to schedule functions on workers and delegate to the workers themselves the task of verifying whether they can effectively execute the scheduled functions.

Thus, controllers can maintain partial knowledge of worker states because the workers themselves perform the verification whether their current status is compatible with the constraints to run a function. This approach is safe from a scheduling point of view, because despite the fact that controllers have an under-approximated view of the load on the workers (the one they issued and not the one issued by the other controllers), workers deny the execution functions targeted at them if their state does not satisfy the function's scheduling constraints (e.g., minimal amount of resources, too many concurrent invocations).

Hence, under the optimistic decentralisation approach, workers become active components. To ensure the satisfaction of the scheduling constraints (the invalidation properties) of a policy, controllers must send in their scheduling request both the (reference to the) function to execute (as done in the centralised case) and the invalidation constraints.

Generally, at scheduling time, this modality saves messages compared to the transaction-based version. In the best case, we have at most two messages: one between the controller and worker for function scheduling and, depending on the architecture, possibly a response message to the controller (alternatively, the worker could store the response in a database). In the worst case, for each block of the policy and for each worker in the block, two messages are exchanged: one for scheduling and one to signal the failure of scheduling from the worker.

Since controllers ignore the current status of the worker, it can happen that multiple requests to execute a family of functions related to the same scheduling policy could target the same workers in a small timeframe, leading to many failures and the relative messages. In these contexts, one can introduce an exponential back-off system whereby, after the failure of scheduling with respect to a certain policy, the controller waits for an exponentially increasing time before forwarding scheduling requests, allowing the interested workers to regain some capacity to run new functions by waiting for the termination of the ones they are running.

We conclude our discussion about the optimistic decentralisation approach by observing that, given that controllers have no need to store the current status of the workers, we can consider controllers almost as stateless services — the only stateful element is the management of the response back to the user, which one can delegate to a dedicated, lightweight service that users can receive their responses from. Having stateless controllers allows one to implement standard scale-in and scale-out techniques to dynamically increase/decrease the number of

instances of controllers. In this way, the system can elastically adapt to possible modifications of the traffic of incoming function execution requests.

4 Implementation

We proceed to present an APP-based FunLess implementation that uses decentralised function scheduling and supports the execution of multiple controllers.

We remark that since FunLess' main focus is the edge-cloud continuum and multi-cloud scenarios, we have a context in which the communication delays between system's nodes is not negligible. Using a transaction- or leader-based coordination policy among several cores possibly deployed on edge and cloud nodes in different geographical zones can significantly slow down the system due to coordination messages — even in the "lighter" case of leader-based coordination, the leader and followers can experience latencies that would slow down scheduling. For instance, imagine a configuration where the leader is in the cloud and requests for the edge must coordinate with the cloud for scheduling on the edge itself. This configuration represents a clear antipattern, because executing the functions at the edge entails bringing computation closer to the consumer/producer of the response/computation and going through the cloud for coordinating the execution of these functions would diminish the benefits of the cloud-edge approach. The opposite configuration is even more inconvenient: the leader is on the edge and the traffic from the cloud has to reach out to the edge, leaving the Cloud data centre, making the communication substantially slower. For these reasons, we chose to implement the distributed version of FunLess following the optimistic approach.

4.1 A Decentralised Variant of FunLess's Architecture

We draw our proposal for a FunLess' decentralised architecture variant in Fig. 2, including the flow of function creation and execution therein. The architecture includes several Workers and, differently from previous Funless architectures [12,14], several Cores and a distributed database, which allows for multi-instance deployments. First, we discuss the workflow followed by Cores and Workers in the FunLess standard implementation. Then, we comment on the modifications to these workflows necessary to implement the optimistic decentralisation approach.

Core Upon receiving a function creation request (as shown in Fig. 2, step 1. Upload), a Core stores the binary in the database by accessing the closest instance (2. Store) and notifies the Workers (3. Broadcast) to cache a local copy (4. Cache), to reduce cold-start overheads. The components communicate via BEAM's distributed inter-process messaging system. When a function invocation reaches a Core (5. Invoke), it retrieves it (if any) from the database (6. Retrieve). Using the latest collected metrics, the Core selects a Worker with enough memory to execute the function (7. Request) if no APP policy is specified, otherwise it follows the policy instruction. Once it selects the Worker, the Core issues the

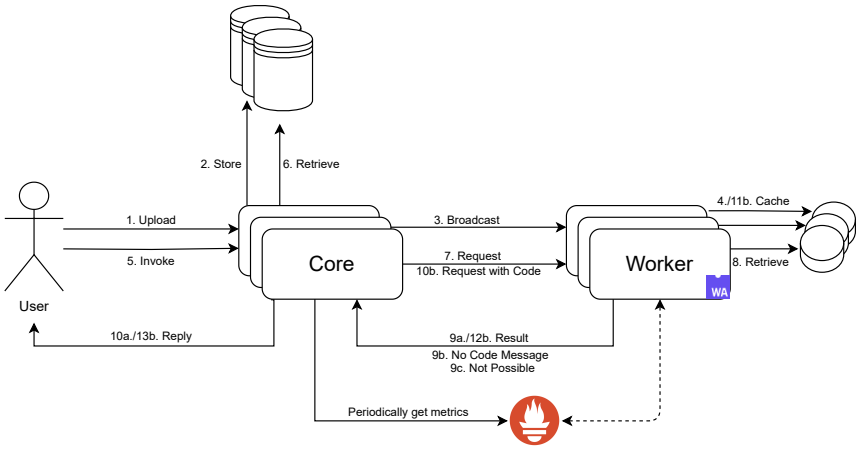

Fig. 2. FunLess decentralised architecture variant, with multiple controllers.

execution of the function therein, waiting to receive the result back, which it relays to the user (10a/13b. reply).

Worker The Worker runs functions via Wasmtime, a runtime for WebAssembly. When a Worker receives the request to run a function (7. Request), it checks its cache for the function's binary (8. Retrieve). If it finds the binary, it runs the function and returns the result to the Core (9a. Result) that contacted it. Otherwise, the Worker informs the Core (9b. No Code Message), which re-sends the request with the code (10b. Request with Code) for caching (11b. Cache) and execution (12b. Result). Alternatively, if the Worker is unable to execute the function, it notifies the Core (9c. Not Possible). The above logic supports efficient function fetching and execution w.r.t. containers, thanks to the small size of Wasm binaries compared to the larger/heavier container images. For caching and eviction, Workers have a configurable cache memory threshold.

Modifications From a practical standpoint, extending the implementation to support the optimistic approach involved modifying the Worker's control flow to recognise the violation of APP invalidation constraints. In particular, when a Worker receives a function allocation request, it checks its current state, namely the number of functions currently scheduled and its actually used capacity (unknown to the Core that sent the request). In case, the current state invalidates the constraints contained in the APP script, the Worker sends a dedicated notification back to the Core, as a new response type from the Worker. On the Core side, the new implementation adds logic to interpret this message. In case such failure message is received, the Core proceeds with the scheduling of the

function by selecting an alternative Worker (if any) following the APP policy of that function, starting a new message exchange with the selected Worker.

5 Experiments

In this section, we show how our FunLess decentralised architectural variant can improve system performance in multi-controllers deployments. We conduct our evaluation with a series of tests involving function invocations from different locations, specifically from EU and US regions, considering two scenarios where the platform has: *i*) one Core in the US region and two Workers, one in the EU and one in the US region, and *ii*) two Cores, one in the EU and one in the US, each with one Worker (as in scenario *i*). In both scenarios, we send 1000 consecutive invocations from the EU and US regions to the (nearest) Core, first without applying any APP script, in order to have "vanilla" invocations, then, repeating the same tests with an APP script that specifies an ad-hoc policy for Worker assignment.

We designed these experiments to evaluate the impact of geographical distribution and scheduling policies on serverless function execution performance. To this aim, we consider a cross-regional deployment with nodes in both the US and EU regions so that we can draw observations from a realistic multi-region deployment scenario, common in production environments to support global user bases and implement resilient service strategies. By placing nodes in geographically distant regions (US and EU), we can clearly measure the effects of network latency on function execution times, particularly the effect of round-trips that occur when requests cross regional boundaries multiple times. The scenarios are particularly suited to measure the impact of scheduling decisions, supporting the direct comparison between local scheduling, remote scheduling, and default (vanilla) scheduling behaviours and quantify the benefits of decentralised vs centralised scheduling decisions in geographically distributed serverless applications.

5.1 Setup

The platform is deployed over 4 nodes in Google Cloud Platform (GCP) in 2 different regions. Each node being a type e2-medium virtual machine with 2 vCPUs and 2 GB of RAM. Two nodes are located in the EU region (europe-west1) and two in the US region (us-central1). In each zone, 1 node is used as a FunLess Worker and the other as a FunLess Core. Both nodes acting as Cores also deploy a PostgreSQL database with a bidirectional replication in order to provide the Cores a consistent view of the system state. In addition, the EU Core hosts the Prometheus instance. Finally, two Locust[5] instances are also hosted in the Core nodes to generate the localized traffic.

The function invoked is a simple Rust function that returns a sample string compiled to WebAssembly. Listing 1.2 shows the APP script used for the tests.

[5] https://locust.io

```
- default:
  - workers: '*'

- eu:
  - workers:
    - 'euworker'
  followup: default

- us:
  - workers:
    - 'usworker'
  followup: default
```

Listing 1.2. APP script used for the tests.

Scenario	Avg	Median	Std
APP Local Scheduling - requests from EU	230.68	230.57	14.35
APP Local Scheduling - requests from US	15.46	15.17	1.49
APP Only US Scheduling - requests from EU	120.58	119.62	5.01
APP Only US Scheduling - requests from US	15.62	15.07	2.22
Vanilla - requests from EU	201.15	231.22	64.00
Vanilla - requests from US	80.64	127.77	61.82

Table 1. Statistics of function invocation requests for 1 Core US Only (in milliseconds).

5.2 Results

Single Core Test For the single Core scenario, where the Core was deployed in the US region, three different test runs were performed:

- **APP for both Workers (Local Scheduling)**: the requests from the EU node were tagged with `eu` and assigned to the EU Worker, while the requests from the US node were tagged with `us` and therefore assigned to the US Worker.
- **APP for US Worker (Only US Scheduling)**: all requests were tagged with `us` and assigned to the US Worker, regardless of their origin.
- **Vanilla**: no function was tagged, corresponding of the situation in which no APP script is used.

Table 1 shows the measurements of the three test runs, including the average, median, and standard deviation of the response times for each requests' origin. There is a notable difference in the response times between the usage for a near-Core Worker and a far-Core Worker. With APP using both Workers, the latencies for requests from the EU region is significantly higher than the requests

from the US region, while it is halved when using just the US Worker. It shows the significant impact of the double round-trip effect when the requests are assigned to the EU Worker. First the request is sent from the EU Locust instance to the US Core, then the Core sends the request back to the EU region for the Worker to execute the function. The result is then sent back to the US region for the Core to process it and send the final response back to the EU Locust instance.

With vanilla invocations, both Cores can use any Worker, and the scheduling is done based on CPU load and memory usage via the scraped metrics from Prometheus updated on a 5-second interval. In our run we observed that both Cores sent the invocation requests mainly to the EU Worker, performing similarly to the APP script with both Workers with requests from the EU region. The requests from the US region also generally performed worse due to the round-trip to the EU Worker. Fig. 3 shows a distribution of the response times for the vanilla invocations. It clearly shows the bimodal distribution when the Cores switch from one Worker to the other.

The most stable conditions are observed when the requests reach the nearest Core and the functions are allocated to the nearest Worker.

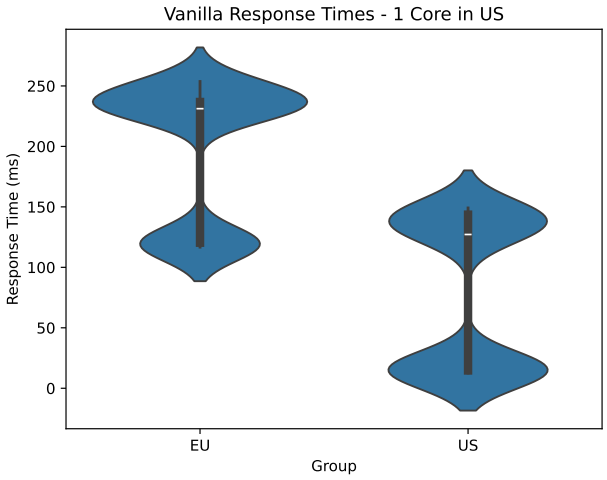

Fig. 3. Distribution of function response time without the APP script. The bimodal distribution is due to the Cores switching from one Worker to the other.

Multi-Core Test For the multi-core scenario, where each region has its own Core, two test runs were performed by sending 1000 requests from the EU and US regions to their respective Cores, at the same time. First using the APP script from Listing 1.2 to assign the requests to the nearest Worker, and once without

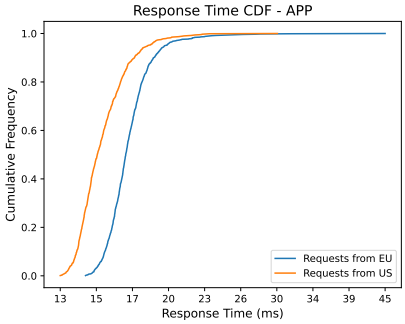

Fig. 4. Distribution of response time for invocations with the APP script.

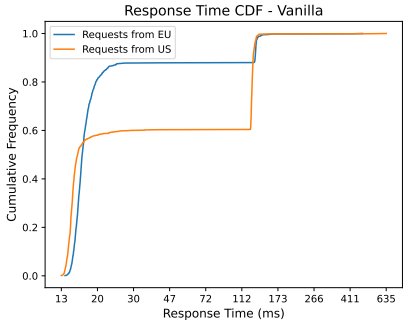

Fig. 5. Distribution of response time for invocations without the APP script.

Scenario	**Avg**	**Median**	**Std**
APP Local Scheduling - requests from EU	17.31	17.03	1.80
APP Local Scheduling - requests from US	15.67	15.30	1.57
Vanilla - requests from EU	32.45	16.38	60.73
Vanilla - requests from US	62.52	15.51	87.36

Table 2. Statistics of function invocation requests with 2 Cores (in milliseconds).

tagging any function and therefore simulating the vanilla scenario in which an APP script is not available.

The results of the test runs are shown in Table 2. The optimal performance is achieved when the invocations are requested from Workers co-located with the Core, which is the case for all requests when using APP. With vanilla invocations, as seen in the experiment with only one Core, both Cores primarily used the EU Worker due to the low impact of the function execution and the delay of the scraping interval, leading to the response times from the US region being significantly higher.

It is interesting to note that the APP script used for these multi-core experiments is the same as the one used in the experiments with the single Core experiments named "APP Local Scheduling" in Table 1. Hence, a comparison between these two scenarios is a fair way to evaluate the advantages of a decentralised architecture for APP-based function scheduling. We have that in the decentralised architecture both the groups of function invocations from EU have a fast average response time (US: 17.31 ms, EU: 15.67 ms) while in the centralised case one of the two groups — the functions requests from the region without the Core — shows the worst performance (US: 15.46 ms, EU: 230.68 ms). This experimental evidence confirms that the decentralised approach can

positively impact the performance of serverless applications, especially when they are geographically distributed and use locality-based scheduling policies.

Figs. 4 and 5 present the cumulative distribution functions (CDFs) of response times in log scale for the two test scenarios—with and without the APP script, respectively. When using APP, the response times are stable and distributed between 13 and 24 milliseconds. When using vanilla invocations, more than 80% of the requests from the EU region used the EU Worker, therefore, keeping response times similar to those with the APP script. With the remaining 20% of the requests, the EU Core switched to the US Worker, significantly increasing latency up to more than 140 milliseconds. This behaviour is also visible from the US region, where, in this case, half of the requests were assigned to the US Worker, and the other half to the EU Worker.

6 Related Work and Conclusion

In this work, we introduce and evaluate a decentralised variant of the APP scheduling architecture, implemented in the FunLess serverless platform. The extension enables multiple controllers to cooperate in scheduling functions, improving scalability, responsiveness, and fault tolerance, especially in edge and multi-cloud deployments. Our implementation follows an optimistic decentralisation strategy where workers locally validate scheduling constraints. This approach removes the need for global coordination and enables controllers to be almost stateless, improving elasticity and system adaptability controller-wise, meaning that controllers can dynamically (dis)appear depending on, e.g., load conditions.

The problem of supporting multiple controllers in APP has also been tackled in previous work [9], through the introduction of a variant of the language, called TAPP (Topology-Aware APP) that enables targeted function scheduling within specific topological zones, implemented on Apache OpenWhisk [23]. Compared to our work, in TAPP, each controller has privileged access to local workers and limited access to remote ones. In our approach, all controllers have full access to all workers. Furthermore, TAPP maintains a single platform entry point, while our implementation enables direct access to individual controllers, significantly reducing latency in geographically distributed scenarios.

We note that OpenWhisk itself supports multiple controllers. However, it does not inherently offer coordination or consistency guarantees for scheduling decisions across them. This aspect limits its usage in scenarios requiring policy-aware scheduling across distributed control points, as controllers may race or generate conflicting decisions due to lack of coordination. Our approach addresses these aspects by embedding constraint checks within the workers themselves.

Similarly, other serverless platforms such as Knative and OpenFaaS support multiple ingress points or decentralized deployments (e.g., via Kubernetes autoscaling, multiple gateways, etc.). However, these systems primarily rely on platform-level load balancing and do not offer declarative, fine-grained scheduling guarantees like those expressible in APP. In both cases, scheduling is either opaque or relies on infrastructure-specific heuristics (e.g., pod autoscaling, metric-

based routing), with no mechanism to express or enforce function-level policies, such as prioritisation, locality, or resource constraints.

We see different open directions to pursue in the future. Notably, while the decentralised variant of FunLess ensures that a function can be executed on a worker only if it satisfies the scheduling constraints in APP — because the constraints are checked by the worker before execution — it can introduce behaviours that are not allowed by the centralised scheduling architecture — globally, the behaviours of the centralised and decentralised variants of the platform can differ. This divorce can happen because in the centralised version each scheduling instance happens atomically, while in the decentralised variant multiple scheduling instances can interleave, reaching configurations that the centralised case would not. We plan to study the consequences of this discrepancy, and, in case they determine significant limitations, we foresee the investigation of modifications to the scheduling protocols or consistency models that can reconcile these anomalies, while retaining scalability and decentralisation. We also envision applying techniques similar to the ones presented in this paper to other serverless platforms and evaluating whether one can extend the latter to offer policy-aware scheduling guarantees in the likes of APP-based FunLess.

References

1. Knative. https://knative.dev/ (2023)
2. Apache Software Foundation : Apache Kafka. https://kafka.apache.org/ (2025)
3. Azure, M.: Microsoft azure functions. https://azure.microsoft.com/ (11 2022)
4. Bernstein, P.A., Newcomer, E.: Principles of Transaction Processing. Morgan Kaufmann, 2nd edn. (2009)
5. Broadcom: Rabbitmq. https://www.rabbitmq.com/ (2025)
6. Cloud, G.: Google cloud functions. https://cloud.google.com/functions/ (11 2022)
7. De Palma, G., Giallorenzo, S., Laneve, C., Mauro, J., Trentin, M., Zavattaro, G.: Leveraging static analysis for cost-aware serverless scheduling policies. Int. J. Softw. Tools Technol. Transf. **26**(6), 781–796 (2024). https://doi.org/10.1007/S10009-024-00776-9
8. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Custom serverless function scheduling policies: An APP tutorial. In: Dorai, G., Gabbrielli, M., Manzonetto, G., Osmani, A., Prandini, M., Zavattaro, G., Zimmermann, O. (eds.) Joint Post-proceedings of the Third and Fourth International Conference on Microservices, Microservices 2020/2022, May 10-12, 2022, Paris, France. OASIcs, vol. 111, pp. 5:1–5:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/OASICS.MICROSERVICES.2020-2022.5
9. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: A declarative approach to topology-aware serverless function-execution scheduling. In: IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022. pp. 337–342. IEEE (2022). https://doi.org/10.1109/ICWS55610.2022.00056
10. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Formally verifying function scheduling properties in serverless applications. IT Prof. **25**(6), 94–99 (2023). https://doi.org/10.1109/MITP.2023.3333071

11. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Function-as-a-service allocation policies made formal. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. REoCAS Colloquium in Honor of Rocco De Nicola - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15219, pp. 306–321. Springer (2024). https://doi.org/10.1007/978-3-031-73709-1_19
12. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Funless: Functions-as-a-service for private edge cloud systems. In: IEEE International Conference on Web Services, ICWS 2024, Shenzhen, China, July 7-13, 2024. pp. 961–967. IEEE (2024). https://doi.org/10.1109/ICWS62655.2024.00114
13. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: An open-whisk extension for topology-aware allocation priority policies. In: Castellani, I., Tiezzi, F. (eds.) Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14676, pp. 201–218. Springer (2024). https://doi.org/10.1007/978-3-031-62697-5_11
14. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Webassembly at the edge: Benchmarking a serverless platform for private edge cloud systems. IEEE Internet Computing (01), 1–8 (dec 2024). https://doi.org/10.1109/MIC.2024.3513035
15. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Affinity-aware serverless function scheduling. In: 22nd IEEE International Conference on Software Architecture, ICSA 2025, Odense, Denmark, March 31-April 4, 2025. IEEE (2025)
16. De Palma, G., Giallorenzo, S., Mauro, J., Zavattaro, G.: Allocation priority policies for serverless function-execution scheduling optimisation. In: Kafeza, E., Benatallah, B., Martinelli, F., Hacid, H., Bouguettaya, A., Motahari, H. (eds.) Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12571, pp. 416–430. Springer (2020). https://doi.org/10.1007/978-3-030-65310-1_29
17. De Palma, G., Giallorenzo, S., Mauro, J., Zavattaro, G.: Allocation priority policies for serverless function-execution scheduling optimisation. In: Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12571, pp. 416–430. Springer (2020). https://doi.org/10.1007/978-3-030-65310-1_29
18. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16) (2016)
19. Jurič, S.: Elixir in action. Manning (2024)
20. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229, https://doi.org/10.1145/279227.279229
21. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: Gibson, G., Zeldovich, N. (eds.) Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014. pp. 305–319. USENIX Association (2014), https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

22. OpenFaaS: Openfaas. https://www.openfaas.com/ (11 2022)
23. OpenWhisk, A.: Apache openwhisk. https://openwhisk.apache.org/ (11 2022)
24. Services, A.W.: Introducing aws lambda. https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/ (11 2022)
25. Stenman, E.: Beam: a virtual machine for handling millions of messages per second (invited talk). In: Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages. p. 4. VMIL 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3281287.3281289, https://doi.org/10.1145/3281287.3281289