

A Decentralized Microservice Scheduling Approach Using Service Mesh in Cloud-Edge Systems

Yangyang Wen¹, Paul Townend¹, Per-Olov Östberg¹, Abel Souza², and Clément Courageux-Sudan¹

¹Umeå University, Umeå, Sweden

{*yangyang.wen, paul.townend, per-olov.ostberg, clement.courageux-sudan*} @ umu.se

²UC Santa Cruz, CA, USA

absouza@ucsc.edu

Abstract—As microservice-based systems scale across the cloud-edge continuum, traditional centralized scheduling mechanisms increasingly struggle with latency, coordination overhead, and fault tolerance. This paper presents a new architectural direction: leveraging service mesh sidecar proxies as decentralized, in-situ schedulers to enable scalable, low-latency coordination in large-scale, cloud-native environments. We propose embedding lightweight, autonomous scheduling logic into each sidecar, allowing scheduling decisions to be made locally without centralized control. This approach leverages the growing maturity of service mesh infrastructures, which support programmable distributed traffic management. We describe the design of such an architecture and present initial results demonstrating its scalability potential in terms of response time and latency under varying request rates. Rather than delivering a finalized scheduling algorithm, this paper presents a system-level architectural direction and preliminary evidence to support its scalability potential.

Index Terms—Microservice-based systems, Service mesh, Decentralized scheduling, Sidecar proxy, Scalability, Latency, Distributed systems

I. INTRODUCTION

Modern distributed systems are undergoing a paradigm shift, extending beyond centralized data centers into a heterogeneous cloud-to-edge continuum [1]. With the rapid growth of federated cloud infrastructures and the proliferation of IoT devices, computation, and data are growing in scale and becoming increasingly decentralized. This evolution introduces new challenges for coordination, as traditional centralized models struggle to maintain efficiency in diverse high-load environments [2]. The scale and geographical distribution of these systems require new approaches to system management, particularly in areas such as scheduling and resource allocation. These trends are also reflected in the rise of stateless, on-demand execution models such as serverless computing, which further challenge traditional centralized scheduling designs.

The microservice architecture has become the dominant model for modern distributed systems [3], breaking down applications into loosely coupled fine-grained services. In particular, microservices enable the replication of services for better scalability. However, this decoupling necessitates

numerous network communications between services located on different machines. In this context, finding the best route between the service replicas to execute a request and meet performance objectives is a significant challenge.

A traditional approach to optimizing the performance of tasks executed within cloud-edge infrastructures relies on schedulers with global knowledge of the system. However, on scale and within dynamic environments, centralized models face inherent limitations. The increasing amount of requests to be scheduled makes schedulers a bottleneck, reducing the performance of applications [4].

The service mesh, typically implemented with Istio [5] or Linkerd [6], offers a programmable networking layer by deploying sidecar proxies alongside each service instance [7]. These sidecars intercept and manage service traffic for better observability, security, and traffic control in a decentralized manner without modifying the application code. In particular, current service mesh solutions offer load balancing between microservice replicas for scalable scheduling of requests [7].

Inspired by this architecture, we investigate whether the network management capability of the service mesh can be extended beyond the classic load balancing algorithms and serve as autonomous in situ schedulers for scalable, low-latency request scheduling at the edge. Specifically, we explore a service scheduling approach centered around service mesh sidecars to make autonomous and local scheduling decisions. This design reduces coordination overhead and eliminates bottlenecks for significantly increased scalability and fault tolerance. To evaluate the proposed service-mesh-based scheduling framework, we instantiate it with a carbon-aware scheduling policy as a representative use case. However, the framework is designed to be agnostic to specific optimization goals and can support a wide range of scheduling strategies.

Prior research has explored decentralized scheduling and application-specific goals in isolation, but their combined impact on scalability in large-scale microservice systems remains underexplored [8], [9]. Moreover, the impact of decentralized scheduling on scalability, particularly under high-load conditions, has not been systematically addressed.

This paper addresses: 1) To what extent can service mesh-

based decentralized scheduling reduce coordination overhead and improve latency, particularly under increasing request concurrency in cloud-edge environments?; 2) How does the proposed sidecar-level decentralized scheduling architecture compare to a centralized scheduler in terms of makespan and stability under varying request rates?; and 3) What are the trade-offs introduced by decentralization in terms of performance consistency and system complexity?

The key contributions of this paper are as follows:

- 1) We propose a novel decentralized scheduling architecture that leverages service mesh sidecar proxies to embed autonomous scheduling logic alongside microservices. This deployment model eliminates centralized coordination overhead while remaining compatible with modern cloud-native platforms.
- 2) We design and implement a simulation-based experimental environment using SimGrid, which models real-world network latency, communication delays, execution times, and energy consumption across a network of multiple hosts. This platform enables a systematic comparison of the proposed decentralized scheduler with a MILP-based (Mixed Integer Linear Programming) centralized approach inspired by Casper [10], a representative centralized scheduler.
- 3) We evaluate the system's scalability by comparing centralized and decentralized scheduling under varying workloads. The results demonstrate that the decentralized approach, enabled by the service mesh architecture, achieves a significantly lower makespan under high load, with minimal performance overhead.

II. RELATED WORK

A. Centralized and Decentralized Scheduling

A broad body of work has explored centralized and decentralized scheduling for improved scalability and responsiveness in distributed systems. Centralized schedulers, such as those used in early cloud and cluster environments, often suffer from scalability bottlenecks and single points of failure, particularly under high load or heterogeneous conditions. Omega [11] proposes a shared-state, multi-scheduler architecture to mitigate contention among multiple scheduling components. Sparrow [12] introduces a distributed, sampling-based scheduler aimed at low-latency task placement. Ray [13] presents a distributed runtime for AI workloads with an actor-based model, enabling dynamic, locality-aware task execution.

In centralized scheduling, global coordination relies on aggregating full system state into a centralized controller to perform global optimization, leading to scalability bottlenecks. In contrast, decentralized scheduling leverages lightweight coordination protocols (e.g., gossip, local state exchange) to achieve eventual consistency, enabling scalable and fault-tolerant scheduling decisions in large-scale cloud-edge environments.

Recent research has increasingly focused on decentralized approaches, distributing scheduling decisions closer to the

execution layer. Such strategies reduce coordination overhead and improve resilience under dynamic and large-scale conditions. Notably, Polaris [14] explores service mesh-compatible scheduling for cloud-native applications, while Hydra [15] avoids the bottleneck of centralized coordination by building a peer-to-peer overlay network between nodes, allowing each node to act as both a computing resource and undertake orchestration tasks.

B. Scheduling for Serverless and Microservices

The rise of Function-as-a-Service (FaaS) platforms has further emphasized the need for scalable, stateless, and dynamic scheduling mechanisms [16], [17]. Statelessness, fine-grained execution, and elasticity in these environments present unique scheduling demands that challenge traditional centralized designs. Systems like Hydra [15] demonstrate the potential of decentralized control in such settings; however, these approaches overlook the challenges of scheduling at the microservice granularity, where long-running services, interdependencies, and locality awareness play a more critical role. Microservice-based environments introduce similar challenges at a finer granularity. Dyme [18] and RESCAPE [19] illustrate how service granularity can improve responsiveness. However, these works rely on centralized schedulers and do not address scalability concerns.

Scheduling algorithms in cloud computing have evolved to address various system constraints, such as sustainability. CASA [20] and CASPER [10] are centralized frameworks that optimize resource allocation to meet system-level goals, but they remain constrained by the limitations of global controllers in terms of scalability and fault tolerance.

Early efforts like GreenScale [8] introduced edge computing with some degree of decentralization, but these systems still face challenges in achieving full scalability and microservice-level granularity. Recent works such as CarbonClipper [21] and Caspian [22] continue to focus on centralized orchestration, which may not fully address the growing needs of modern cloud-edge systems.

C. Classification Criteria

To compare existing scheduling and autoscaling approaches, we classify them based on three key dimensions: decentralization, scalability, and system overhead.

1) *Decentralization*: We define centralized systems as those where a global controller makes all scheduling and scaling decisions.

- Partial decentralized: It refers to architectures where limited local autonomy exists under global policy control.
- Semi-decentralization: It describes hierarchical systems where multiple local controllers handle fine-grained decisions while a global scheduler coordinates high-level resource allocation.
- Fully decentralized: Fully decentralized systems distribute decision-making entirely among local agents with no global coordinator.

2) *Scalability*: We evaluate scalability based on the system's ability to handle increasing cluster sizes while maintaining performance and stability. Systems supporting only small-scale clusters (e.g., up to 8 nodes) are classified as *Low*. Those successfully evaluated on tens of nodes (e.g., up to 32-50 nodes) are categorized as *Moderate*. Systems capable of operating on hundreds or more nodes fall into the *High* category.

3) *System Overhead*: System overhead reflects the computational and communication cost incurred during scheduling and auto scaling decisions at runtime. *Low* overhead systems rely on static rules or threshold-based policies with minimal runtime computation. *Moderate* overhead systems employ lightweight runtime heuristics, predictive models, or partial coordination among components. *High* overhead systems involve complex runtime optimization (e.g., local search, reinforcement learning, or multi-objective trade-offs) and require extensive global state collection and coordination.

Table I summarizes a number of existing scheduling approaches based on decentralization, scalability, and system overhead. *Partial decentralized* refers to approaches where local nodes maintain limited autonomy under global control, while *semi-decentralization* indicates hierarchical models combining global coordination with local schedulers. Scalability is classified based on the system's capability to support larger clusters while maintaining performance (e.g., from tens to hundreds of nodes). System overhead captures the computational and communication costs introduced by scheduling decisions, including runtime optimization complexity and control plane coordination.

TABLE I: Comparison of existing Scheduling approaches

Approach	Decentralization	Scalability	System Overhead
CASA [20]	Centralized	Moderate	High
CASPER [10]	Centralized	Moderate	Moderate
GreenScale [8]	Partial	Moderate	Moderate
Hydra [15]	Fully Decentralized	High	Moderate
Polaris [14]	Semi-Decentralized	High	High
This work	Fully Decentralized	High	Moderate

The classification is based on the architecture of the system, the evaluation scale, and the complexity of the scheduling algorithm described in the original papers and follows a consistent interpretation across all compared methods.

Our approach leverages a sidecar-based fully decentralized scheduling framework, where each pod performs local decision-making using lightweight greedy algorithms based on real-time local observations. This design eliminates centralized coordination overhead, enabling high scalability, while the runtime cost of sidecars leads to moderate system overhead compared to purely reactive or centralized models.

III. SYSTEM MODEL AND ASSUMPTIONS

A. System Architecture

The system consists of multiple geo-distributed regions, each with a cluster of compute nodes hosting stateless microservices. Services are pre-deployed in one or more regions, enabling redundancy and low-latency access.

Each service can execute a fixed number of requests in parallel using a thread pool. Service instances are paired with sidecars, responsible for the decentralized routing of requests between service replicas.

The system maintains an eventually consistent distributed metadata store to support decentralized scheduling. This shared control layer provides dynamic state information, such as resource utilization (e.g., CPU and memory, updated every few seconds), request queue statistics (which may fluctuate at millisecond granularity), and other relevant performance metrics (e.g., network load, latency). Propagation protocols, for instance, based on gossip can ensure the propagation of updates at scale without strong consistency but are out of the scope of this work.

Request handling is modeled as a queue-based mechanism: each service instance buffers incoming requests and executes them in a first-in, first-out (FIFO) manner to preserve arrival order. Service execution follows an actor-based concurrency model. This model enables parallel and distributed processing.

B. Decentralized Scheduling Mechanism

Scheduling follows a hop-by-hop, sidecar-driven approach, as shown in figure 1. Each application is represented as a service chain:

$$SC = [S_1, S_2, \dots, S_n]$$

As soon as service S_{i-1} finishes execution, its co-located sidecar selects an eligible host based on the scheduling objectives to execute S_i . The decision is taken based on the local cached view of the global metadata.

This strategy allows adaptive, decentralized scheduling; each sidecar makes decisions independently, improving fault tolerance and scalability. The architecture aligns with the Function-as-a-Service (FaaS) paradigm, where stateless, ephemeral (i.e., short-lived, and on-demand) services can scale elastically. Frequently used services remain warm, while others are gradually scaled down to preserve resources.

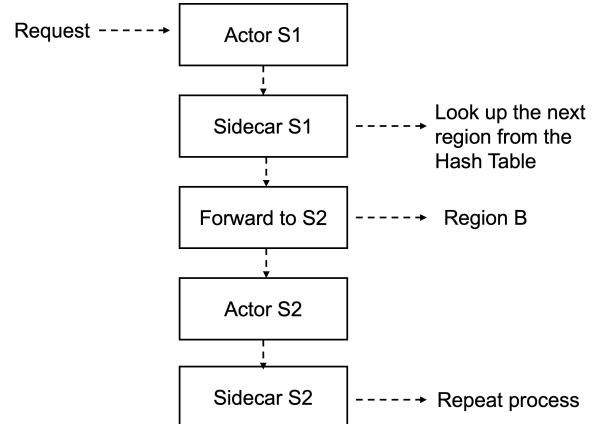


Fig. 1: Chain-based Execution Model: Hop-by-hop decentralized scheduling driven by sidecar agents using locally cached global metadata.

C. Execution and Coordination Model

Figure 1 illustrates the system’s architecture and the coordination mechanism. Key characteristics of our approach include: (1) Autonomous Scheduling: Each sidecar performs routing decisions independently using the local state. (2) Scalability: Stateless services scale dynamically based on observed demand. (3) Cold-start trade-offs: Cold starts are mitigated by retaining warm replicas in frequently active regions.

a) *Assumptions and Discussion:* We assume: (1) Services are stateless and replicated. (2) Execution is modeled using an actor paradigm. (3) Scheduling is handled by co-located sidecar agents. (4) Metadata is shared via an eventually consistent global store.

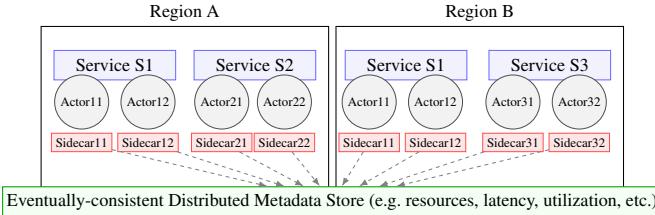


Fig. 2: System Architecture: Geo-distributed regions host replicated services. Each actor instance is paired with a sidecar that performs local scheduling using metadata from a shared, eventually-consistent distributed metadata store.

IV. METHODOLOGY

In this section, we present our scheduling methodology for microservice requests, covering both centralized and decentralized approaches. We begin by defining a common scheduling objective, and then describe each approach in detail, including algorithms, architectural differences, and latency implications. Finally, we compare the computational complexity of the two scheduling strategies based on experimental runtime measurements.

A. Scheduling Objective

To enable a fair comparison between centralized and decentralized scheduling, we define a common objective. The goal of this work is to minimize the overall resource utilization (or cost) incurred by executing a microservice request across a service chain while satisfying end-to-end latency constraints (QoS). Our architecture supports various scheduling goals beyond the one presented here.

$$\min_{r_1, \dots, r_n} \sum_{i=1}^n \text{ResourceCost}(S_i, r_i) \quad \text{s.t. } \text{Latency}(SC) \leq L_{\max},$$

Where $\text{ResourceCost}(S_i, r_i)$ denotes the cost associated with executing service S_i in region r_i , considering factors such as energy, computational, and resource usage, while $\text{Latency}(SC)$ is the end-to-end delay through the service chain. This formulation serves as a baseline for this work, but the framework can support a variety of alternative scheduling objectives.

B. Architectural Assumptions

We model each microservice request as a chain of services, executed in sequence across a distributed infrastructure. Services are deployed as independent actor instances, each paired with a sidecar agent responsible for scheduling decisions. Asynchronous communication in the system is implemented using SimGrid, and the system architecture mimics some aspects of a service mesh, enabling communication between services.

Each sidecar maintains a metadata store for real-time metrics including:

- Resource usage across different regions,
- Latency estimates to upstream services,
- Resource availability per region.

These metrics are the basis for making scheduling decisions.

C. Centralized Scheduler

In centralized scheduling architectures (e.g., Casper [10]), a global controller computes the full execution path for each service chain request. Optimization is framed as a MILP problem, subject to resource and latency constraints:

- A global scheduler selects hosts for all services in a request’s chain.
- The MILP objective minimizes total emissions while satisfying CPU and network latency constraints.
- The availability of hosts is verified prior to optimization; the system fails if no valid mapping exists.

Algorithm 1: Centralized Scheduling with MILP Optimization

Input: Request r with service chain $SC = [S_1, \dots, S_n]$
Output: Request completion or failure due to constraints

```

1 for each service  $S_i$  in  $SC$  do
2   Find available hosts for  $S_i$ ;
3   if no hosts available then
4     Return failure;
5 Define MILP solver
6 Define host selection variables  $x_{i,h}$  for each service  $S_i$  and host  $h$ 
7 Set objective: Minimize carbon emission
8 Add constraints for CPU and latency
9 Solve MILP;
10 if no optimal solution then
11   Return failure;
12 Select optimal hosts for each service;
13 For each selected host, retrieve corresponding mailbox;
14 Calculate total latency between services;
15 Return execution path;
16 Send request to first mailbox;
```

We implement a customized MILP-based central scheduler (as shown in Algorithm 1) based on the formulation of Casper [10]; once the MILP problem is solved, the optimal

execution path is obtained from the set of constraints, and the request can be forwarded to the first service of the chain. All service instances are pre-provisioned and managed centrally, leading to potential bottlenecks and reduced elasticity under high load.

D. Decentralized Scheduler

Algorithm 2: Decentralized Scheduling via Sidecars

Input: Request r with service chain $SC = [S_1, \dots, S_n]$
Output: Request completed or dropped

- 1 Initialize $i \leftarrow 1$, currentRequest $\leftarrow r$;
- 2 **while** $i \leq n$ **do**
- 3 $S_i \leftarrow$ next service in chain;
- 4 Actor-sidecar at S_i queries local metadata store:
 Get system metric data (e.g., resource availability, latency, bandwidth, etc.);
- 5 **if no feasible region for S_i then**
 Drop request;
 Return failure;
- 6 Select execution region r_i via local scoring rule
(e.g., resource cost + latency penalty);
- 7 Send request to actor replica of S_i in region r_i ;
- 8 Wait for S_i execution to complete;
- 9 $i \leftarrow i + 1$;
- 13 Return success;

Our decentralized scheduler (as seen in Algorithm 2) distributes control across services, enabling local decisions using sidecar agents. Each actor-sidecar pair selects an execution region by filtering candidates that meet both latency and resource availability constraints, before selecting the region with the best performance metrics (e.g., resource utilization and latency).

This model leverages the Actor Model [23] for scalability and fault isolation. Communication is asynchronous, with localized scheduling introducing sidecar overhead at each hop.

E. Modeling Scheduling Latency

In addition to computational cost, the end-to-end request latency exhibits structural differences between centralized and decentralized scheduling. Under centralized scheduling, latency includes a one-time global scheduling delay followed by alternating transmission and processing stages along the service chain. In contrast, decentralized scheduling introduces a sidecar latency for each hop due to local executions in the sidecar, trading centralized coordination for per-hop overhead.

Given a service chain $S = \{S_1, S_2, \dots, S_n\}$, the total latency under centralized scheduling is:

$$L_{\text{central}} = L_{\text{sched}} + \sum_{i=1}^{n-1} \left(\text{TWL}_i + L_{\text{proc}}^{(i)} \right) + L_{\text{proc}}^{(n)} \quad (1)$$

where L_{sched} is the centralized scheduling latency, TWL_i is the transmission latency before service S_i , and $L_{\text{proc}}^{(i)}$ is the processing latency of service S_i .

Under decentralized scheduling, the total latency is:

$$L_{\text{decentral}} = \sum_{i=1}^{n-1} \left(\text{TWL}_i + L_{\text{proc}}^{(i)} + L_{\text{sidecar}}^{(i)} \right) + \text{TWL}_n + L_{\text{proc}}^{(n)} \quad (2)$$

Here, $L_{\text{sidecar}}^{(i)}$ represents the sidecar scheduling delay introduced after each intermediate service hop. The final hop omits sidecar latency, assuming no further decision is needed post-processing.

F. Complexity comparison of scheduling algorithms

To better understand the scalability of each scheduling approach, we analyze their computational complexity in terms of service chain length and replica count. We implement both in C++ and profile their execution time; initial empirical results are presented in Section VII-A.

V. IMPLEMENTATION USING SIMGRID

To realistically evaluate our scheduling strategies, we implement the scheduling approaches using the SimGrid simulator. This section details how we implement the two schedulers, the microservice applications and service mesh sidecars.

A. Actor-Based Execution Model

To simulate resource usage in a cloud-edge environment, we adopt an execution model implemented within the SimGrid simulation framework [24]. This model captures the interplay between task concurrency and CPU core allocation, representing realistic constraints of distributed resource scheduling. The simulation of a computational machine (host) consists of two primary components:

- A physical CPU, configured with several cores, frequency, and energy profiles.
- A set of actors simulating the resource usage of processes (service handlers, sidecar) running on the CPU and using simulated network interfaces.

Our service mesh scheduling simulations are governed by the following.

The key parameters governing our service mesh scheduling simulations are summarized in Table II.

TABLE II: Simulation Parameters

Symbol	Description
A	Number of parallel actors per host per service replica
C	Number of physical CPU cores on each host
λ	Request arrival rate (requests per time unit)
T_{comp}	Fixed computation time per request (ideal, no contention)

B. Execution Scenarios and Resource Contention

We consider two configurations based on the relationship between A and C :

a) *Case 1: $C \geq A$ and no CPU bottleneck (Underloaded or Balanced System)*: In this case, sufficient CPU cores are there to serve all requests instantly. When a request arrives on a machine:

- It is distributed to one of the available actors.
- That actor makes exclusive use of a CPU core to run the request with a computation time T_{comp} , and a waiting time is negligible unless the arrival rate λ exceeds the processing capability of the CPU.

The maximum concurrency is thus limited by A :

$$N_{\text{concurrent}} = \min(\lambda, A)$$

The system's CPU utilization is bounded by:

$$U_{\text{CPU}} = \frac{\min(\lambda, A) \cdot T_{\text{comp}}}{C \cdot T_{\text{comp}}} = \frac{\min(\lambda, A)}{C}$$

b) *Case 2: $C < A$ (Overloaded System)*: In this setting, the number of actors exceeds the number of CPU cores, leading to potential oversubscription. SimGrid models CPU contention through time-sharing. Each actor receives a fraction of the available compute capacity, resulting in extended execution times and increased latency:

- When all cores are busy, actors share CPU time.
- Each active actor experiences extended computation time due to time-slicing and CPU contention.
- Let n_{active} denote the number of simultaneously executing actors; if $n_{\text{active}} > C$, then each actor receives $\frac{1}{n_{\text{active}}}$ of a core.

The effective computation time per request becomes:

$$T'_{\text{comp}} = T_{\text{comp}} \cdot \frac{n_{\text{active}}}{C}$$

This slowdown affects both request latency and overall makespan. If the request arrival rate λ exceeds A , then queueing occurs, and waiting time contributes to total delay:

$$T_{\text{total}} = T_{\text{wait}} + T'_{\text{comp}}$$

C. Concurrency and Scalability Insights

Concurrency is a foundational concept in cloud computing, enabling efficient task execution and resource utilization through parallel processing [25], [26]. In our model, we capture hard parallelism by assigning multiple CPU cores per host and implementing soft concurrency through the number of available actors. This separation allows us to explore how system throughput and latency respond to both physical and logical concurrency constraints:

- Hard parallelism: determined by the number of CPU cores C .
- Soft concurrency: determined by the number of actors A that can handle simultaneous requests.

By tuning A and C , we emulate both saturation scenarios and fully parallelizable workloads. For instance, if $\lambda \leq C$, requests are processed without contention; if $C < \lambda \leq A$, concurrent processing occurs with extended T'_{comp} ; if $\lambda > A$, queuing becomes inevitable.

This abstraction supports a realistic evaluation of cloud scheduling strategies under dynamic workloads and hardware constraints.

D. Evaluation Architecture for Concurrency and Scheduling Comparison

To evaluate scheduling overhead and scalability in isolation, we adopt the default cluster topology recommended by SimGrid [27], which in our case consists of 1100 hosts connected via homogeneous network links and grouped into 10 logical regions. While the framework can support more realistic energy or latency-aware models, we focus specifically on the scheduling layer and abstract away detailed bandwidth or geographic latency differences. Historical region-level heterogeneity (e.g., carbon intensity) is introduced to support a comparative evaluation of decentralized behavior, with our scope centered on scheduling logic rather than the complete end-to-end service execution process (from request submission to response delivery).

To ensure a fair comparison between centralized and decentralized scheduling approaches, both of them are evaluated on the same network topology. The infrastructure includes 1100 hosts, each with 24 CPU cores, an Ethernet network interface, and an energy profile as summarized in Table III.

TABLE III: Key SimGrid Host Configuration Parameters

Parameter	Value
Number of Hosts	1100
CPU Cores per Host	24
CPU Speed	10 GFLOPS
Network Bandwidth	125 MBps
Network Latency	50 μ s
Backbone Bandwidth	2.25 GBps
Backbone Latency	500 μ s
Power (Off State)	10 W
Power (Idle: 0% usage)	20 W
Power (Max: 100% usage)	200 W

VI. EVALUATION

In evaluating scalability, we focus primarily on the scheduler's decision latency growth under increasing request rates. While resource utilization reflects scheduling optimality, it is orthogonal to the system's scalability capabilities.

This section presents the initial evaluation of our centralized and decentralized scheduling strategies using the SimGrid simulation framework. We compare their performance across varying workloads, focusing on request makespan and system scalability. The simulation is still under development and thus results are only indicative until comparatively evaluated against a real implementation.

A. Simulation Environment and Experimental Setup

Experiments are conducted using the SimGrid simulation framework [24]. We chose SimGrid, a flow-level simulator because of its capability to accurately simulate the network and application performance of cloud-edge infrastructures,

including microservices [28].¹ We simulate a system of 1100 hosts distributed across 10 regions. Each service in the 10-stage microservice workflow has 50 replicas, evenly placed across the regions.

Requests are generated at rates ranging from 1 to 15,000 requests per second (rps). In the centralized setting, requests are routed through a global scheduler node; in the decentralized setting, decisions are made locally at each service node. Tables IV and V summarize the infrastructure allocation under both scheduling modes. Table VI summarizes the distribution of servers, replicas, and system metrics across the regions, for both centralized and decentralized scheduling.

The primary metric is the end-to-end makespan per request, defined as the elapsed time from request generation to final response. It can capture the cumulative impact of scheduling decisions, network latency, and resource contention.

TABLE IV: Hosts distribution for the centralized algorithm.

Region	Number of Hosts (Centralized)
Region 1-9	109 hosts each (Total 981 hosts)
Region 10	117 hosts
Client	1 host (randomly assigned region)
Central Scheduler	1 host (randomly assigned region)

TABLE V: Hosts distribution for the decentralized algorithm.

Region	Number of Hosts (Decentralized)
Region 1-9	109 hosts each (Total 981 hosts)
Region 10	118 hosts
Client	1 host (randomly assigned region)
Server (Replica Execution)	1099 hosts (randomly assigned region)

TABLE VI: System and Request Configuration

Item	Description
Number of request types	1
Request length	10 (involves services S1 to S10)
Number of service types	10
Replicas per service	50
Total number of services	500
Number of regions in system	10
Hosts per region	Around 100
Total number of hosts	Over 1000
Replica placement	Distributed across hosts in all regions

VII. RESULTS AND ANALYSIS

A. Empirical results

To quantify the runtime impact of centralized vs. decentralized scheduling, we measure the computational time taken for scheduling decisions. We implement both centralized and decentralized algorithms in C++, and execute them under varying configurations. We use varying service chain lengths {3, 5, 10, 20, 50, 100} and service replicas counts {5, 10, 20, 50, 100, 200, 500, 1000}. For each configuration, we record the average wall-clock time over five executions

¹A flow-level simulator models communication at the granularity of logical flows (e.g., data transfers between services), rather than simulating individual packets. This abstraction significantly improves simulation speed while maintaining sufficient accuracy for evaluating large-scale distributed systems.

to reduce the impact of external interferences. Figure 3a depicts the duration of processing requests with the centralized approach and Figure 3b for the decentralized approach.

Empirical observations match theoretical expectations: increasing the search space with longer chains and a higher number of replicas leads to longer execution times for centralized scheduling than for a decentralized approach dividing the search space.

B. Performance Comparison

We perform an analysis of makespan for varying request rates using our simulator. The makespan is defined as the total time taken to execute a microservice request including the scheduling time, in seconds. The makespan was extracted from the simulations' output for both algorithms, and the average makespan was computed.

Figure 4 shows indicative simulation results of the end-to-end latency of request executions against the rate of requests sent to applications per second.

The graph indicates that the centralized algorithm exhibits lower makespan at lower request rates but suffers significant latency increases as load grows, due to the overhead of centralized decision-making. In contrast, the decentralized algorithm results indicate a more consistent performance as the system scales. These performance indications show a decentralized service mesh-based approach has promise, but our findings still need to be validated in a real hardware environment.

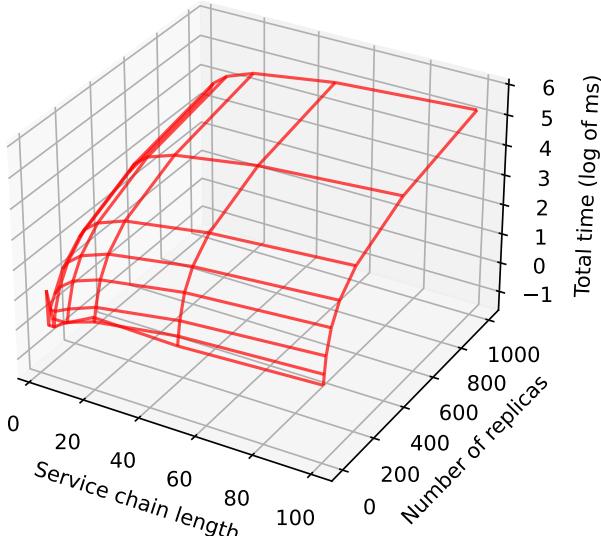
VIII. CHALLENGES IN DECENTRALIZED SCHEDULING

While our proposed architecture demonstrates the feasibility and scalability of decentralized scheduling via local sidecar agents, it also introduces several open challenges that warrant further exploration:

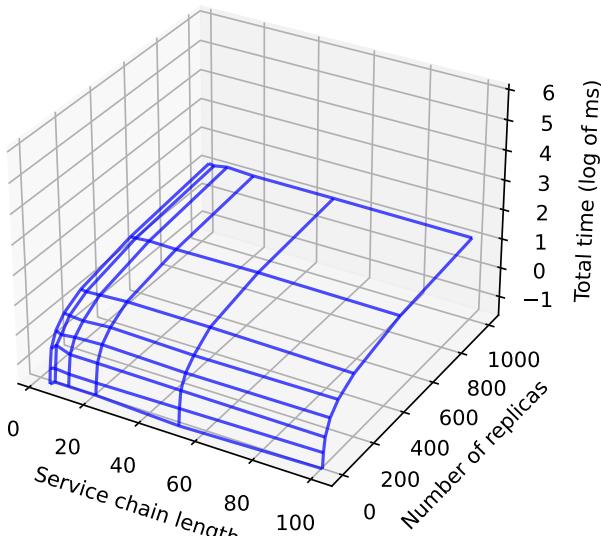
a) Load Balancing and Performance Consistency: Local decision-making based on incomplete system metadata can lead to coordination blind spots—for example, multiple agents independently selecting the same underloaded node during bursty traffic, resulting in resource contention and performance degradation. Designing lightweight coordination or feedback mechanisms remains an open research direction.

b) Operational Complexity and Observability: The shift from centralized to distributed control increases the complexity of monitoring, debugging, and managing system-wide behavior. Effective abstractions and observability tools must be developed to maintain transparency and debuggability without reintroducing centralized bottlenecks.

c) Policy Conflicts and Heuristic Divergence: Autonomous sidecars may implement heterogeneous heuristics, especially in multi-tenant or cross-team environments. This lack of harmonization can lead to inefficient resource usage or conflicting scheduling decisions. Establishing interoperable coordination protocols or shared policy layers may help align local actions with global objectives.



(a) Centralized scheduling execution time



(b) Decentralized scheduling execution time

Fig. 3: 3D surface plots of scheduler wallclock time as a function of service chain length and replica count. Centralized scheduling incurs steep growth in computational demand, while decentralized scheduling remains more scalable.

d) Fault Tolerance and Failure Containment: Without centralized oversight, local failures (e.g., misbehaving sidecars or overloaded replicas) may propagate more easily. Future designs must address how to isolate faults, introduce redundancy, or apply selective consensus without compromising the

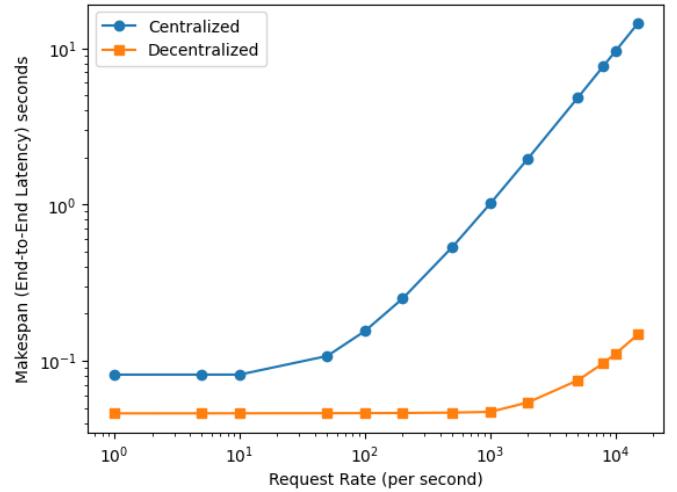


Fig. 4: Indicative Simulation Results Showing Makespan Comparison between Centralized and Decentralized Scheduling Algorithms

system's scalability.

These challenges define the research frontier for decentralized scheduling and highlight the necessary trade-offs between autonomy, coordination, and operational simplicity in distributed service infrastructures.

IX. CONCLUSION

This work proposes a decentralized, actor-based scheduling architecture for stateless microservices deployed across geodistributed regions; each service replica is paired with a lightweight sidecar agent that performs hop-by-hop scheduling based on locally synchronized metadata from a distributed hash table. This aims to eliminate centralized bottlenecks - enabling elastic scalability, improved responsiveness under dynamic workloads, and efficient decision-making based on local system metrics.

The approach aligns with the growing trend of Function-as-a-Service (FaaS) and microservices architectures, where statelessness and locality facilitate adaptive control.

Initial simulation-based results provide an indication that our decentralized approach (despite being based on a greedy path selection) outperforms centralized scheduling in terms of scalability and makespan, especially under high request rates. The results also indicate challenges related to balancing autonomy with global coordination.

Future work will begin with a comprehensive validation of this initial simulation-based work through implementation on a real Kubernetes-based cluster environment. We will use this implementation to also investigate additional optimization metrics, including cost, resource utilization, and latency.

Subsequent to this, we aim to focus on improving scheduling resilience in the presence of stale or partial metadata. Additionally, we will explore hybrid scheduling approaches that combine local autonomy with limited global coordination

to address the challenges of failure isolation and coordination under bursty loads.

ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. This work is also supported by the European Commission through the Horizon Europe project SovereignEdge.Cognit under Grant Agreement 101092711.

REFERENCES

- [1] Y. Patel, P. Townend, A. Singh *et al.*, “Modeling the green cloud continuum: integrating energy considerations into cloud–edge models,” *Cluster Computing*, vol. 27, pp. 4095–4125, 2024.
- [2] W. Khalouli and J. Huang, “Cluster resource scheduling in cloud computing: literature review and research challenges,” *Journal of Supercomputing*, vol. 78, pp. 6898–6943, 2022.
- [3] P. Jamshidi, C. Pahl, N. C. Mendonga, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.
- [4] K. Senjab, S. Abbas, N. Ahmed, and A. u. R. Khan, “A survey of kubernetes scheduling algorithms,” *Journal of Cloud Computing*, vol. 12, no. 1, p. 87, 2023.
- [5] Istio, “Istio,” <https://istio.io/>, last accessed: 2025-05-10.
- [6] Linkerd, “Linkerd: The only service mesh designed for human beings,” <https://linkerd.io/>, last accessed: 2025-05-10.
- [7] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service mesh: Challenges, state of the art, and future research opportunities,” in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225.
- [8] Y. G. Kim, U. Gupta, A. McCrabb, Y. Son, V. Bertacco, D. Brooks, and C.-J. Wu, “Greenscale: Carbon-aware systems for edge computing,” *arXiv preprint arXiv:2304.00404*, 2023.
- [9] S. Aggarwal, M. Bastopcu, T. Basar, S. Ulukus, N. Akar *et al.*, “Fully decentralized computation offloading in priority-driven edge computing systems,” *arXiv preprint arXiv:2501.05660*, 2025.
- [10] A. Souza, S. Jasoria, B. Chakrabarty, A. Bridgwater, A. Lundberg, F. Skogh, A. Ali-Eldin, D. Irwin, and P. Shenoy, “Casper: Carbon-aware scheduling and provisioning for distributed web services,” in *Proceedings of the 14th International Green and Sustainable Computing Conference*, 2023, pp. 67–73.
- [11] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 351–364.
- [12] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, “Sparrow: distributed, low latency scheduling,” in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 69–84.
- [13] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, “Ray: A distributed framework for emerging {AI} applications,” in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.
- [14] T. Puszta, S. Nastic, A. Morichetta, V. C. Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, “Polaris scheduler: Slo-and topology-aware microservices scheduling at the edge,” in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2022, pp. 61–70.
- [15] L. L. Jimenez and O. Schelen, “Hydra: Decentralized location-aware orchestration of containerized applications,” *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2664–2678, Oct.-Dec. 2022.
- [16] P. Zuk and K. Rzadca, “Scheduling methods to reduce response latency of function as a service,” in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 132–140.
- [17] Y. Zhao, W. Weng, R. van Nieuwpoort, and A. Uta, “In serverless, os scheduler choice costs money: A hybrid scheduling approach for cheaper faas,” in *Proceedings of the 25th International Middleware Conference*, 2024, pp. 172–184.
- [18] A. Samanta and J. Tang, “Dyme: Dynamic microservice scheduling in edge computing enabled iot,” *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6164–6174, 2020.
- [19] J. Wang, G. Wang, T. Wo, X. Wang, and R. Yang, “Rescape: A resource estimation system for microservices with graph neural network and profile engine,” in *2024 IEEE International Conference on Joint Cloud Computing (JCC)*. IEEE, 2024, pp. 37–44.
- [20] S. Qi, H. Moore, N. Hogade, D. Milojicic, C. Bash, and S. Pasricha, “Casa: A framework for slo-and carbon-aware autoscaling and scheduling in serverless cloud computing,” in *2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2024, pp. 1–6.
- [21] A. Lechowicz, N. Christianson, B. Sun, N. Bashir, M. Hajiesmaili, A. Wierman, and P. Shenoy, “Carbonclipper: Optimal algorithms for carbon-aware spatiotemporal workload management,” *arXiv preprint arXiv:2408.07831*, 2024.
- [22] T. Bahreini, A. N. Tantawi, and O. Tardieu, “Caspian: A carbon-aware workload scheduler in multi-cluster kubernetes environments,” in *2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2024, pp. 1–8.
- [23] C. Camilleri, J. G. Vella, and V. Nezval, “Actor model frameworks: an empirical performance analysis,” in *International Conference on Information Systems and Management Science*. Springer, 2022, pp. 461–472.
- [24] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, “Lowering entry barriers to developing custom simulators of distributed applications and platforms with SimGrid,” *Parallel Computing*, vol. 123, pp. 103–125, 2025.
- [25] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, “Above the clouds: A berkeley view of cloud computing,” Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Tech. Rep., 2009.
- [26] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [27] SimGrid, “Platform examples,” 2025, accessed: 2025-05-10. [Online]. Available: https://simgrid.org/doc/latest/Platform_examples.html
- [28] C. Courageux-Sudan, A.-C. Orgerie, and M. Quinson, “Automated performance prediction of microservice applications using simulation,” in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2021, pp. 1–8.