Self-Distributing Systems in the Wild: An Experimental Study on a Real-world Edge-Cloud Continuum Platform

Matheus Ávila Naspolini[1], Alison R. Panisson[1], Jim Lau[1], Martín Vigil[1], Luiz F. Bittencourt[2], Roberto Rodrigues-Filho[3]

[1]Departamento de Computação – Universidade Federal de Santa Catarina (UFSC)
Araranguá – SC – Brasil

[2]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brasil.

[3]Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Brasília – DF – Brasil.

matheus.avila@grad.ufsc.br,
{alison.panisson, jim.lau, martin.vigil}@ufsc.br,
bit@ic.unicamp.br, roberto.filho@unb.br

Abstract. *Contemporary distributed applications must often simultaneously satisfy low-latency requirements and high computational demand in order to deliver high Quality of Experience to end users. The edge–cloud continuum has emerged as a promising infrastructure paradigm to address these requirements by combining proximity to users at the edge with the abundant computing resources of the cloud. However, despite this infrastructural potential, service-level mechanisms capable of jointly exploiting network and computing resources remain limited, as these dimensions are often managed independently. To address this gap, the concept of self-distributing systems (SDS) has been proposed, enabling applications originally designed for local execution to autonomously spread themselves across distributed infrastructures at runtime. By dynamically relocating and replicating components, SDS facilitate the simultaneous exploitation of low-latency edge resources and high-performance cloud resources. Nevertheless, the practical integration of SDS with modern deployment and orchestration environments is still unexplored. This paper presents SDS_SWARM, an architecture that integrates the SDS paradigm with Docker and Docker Swarm for real-world edge and cloud infrastructures. We evaluate the architecture using a CPU-intensive service to assess its adaptive capabilities. Experimental results demonstrate improved adaptability across heterogeneous environments, validating the feasibility and benefits of combining self-distributing systems with modern edge–cloud technologies.*

1. Introduction

Modern distributed systems must often be mobile and flexible in their deployment to fully exploit edge–cloud infrastructures [Bittencourt et al. 2025, Bittencourt et al. 2018] and deliver applications that maintain a high Quality of Experience (QoE) for end users [Gama et al. 2024, Gao and Wen 2021]. Applications can be placed close to users at the edge to reduce network latency or scaled in the cloud through the creation of multiple replicas, thereby improving responsiveness and overall throughput.

This need for deployment flexibility has driven industry standards toward stateless and highly componentized software architectures, such as Function-as-a-Service (FaaS) [Calavaro et al. 2025] and microservice-based architectures [Fu et al. 2022]. These architectural styles facilitate service placement across the computing continuum and enable the exploitation of platform-level features, including automatic scaling, rolling updates and rapid service redeployment.

However, despite these advantages, most non-trivial applications inevitably require state. A common strategy is to externalize and isolate state in databases, allowing services to remain stateless while depending on an external data source. While effective, this approach restricts service mobility and introduces new challenges in latency and scalability, particularly in edge environments. These limitations have motivated research areas such as stateful FaaS [Pfandzelter and Bermbach 2023] and Self-distributing Systems (SDS) [Rodrigues-Filho et al. 2023, Rappa et al. 2024, Rodrigues-Filho and Porter 2022].

SDS is based on constructing systems from small, highly reusable software components that can be composed to form complex applications. These components may be relocated or replicated across a distributed infrastructure to improve system performance and adapt to changing execution conditions. Since some of these components may be stateful, the SDS paradigm provides autonomous mechanisms for state management while distributing its components across the infrastructure. Although SDS has been explored in the literature as a potential programming model for the computing continuum [Rodrigues-Filho et al. 2023], it has not yet been thoroughly evaluated in real-world, large-scale edge–cloud infrastructures.

This paper focuses on extending and evaluating the SDS concept in practical computing continuum platforms, spanning from edge devices to cloud computing resources. To this end, we employ Docker[1], Docker Swarm[2], and Tailscale[3] to provide a homogeneous execution environment in which SDS services can be deployed and managed across heterogeneous infrastructures. A CPU-intensive benchmarking application spanning edge devices and cloud resources hosted on Google Cloud was developed to demonstrate the feasibility of SDS in a real-world computing continuum. The application serves as a case study to evaluate the behavior and performance of SDS applications in realistic deployment scenarios. We make our implementation publicly available[4] to support the reproducibility of our results.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the SDS_SWARM architecture. Section 4 presents the experimental scenarios and discusses the results. Finally, Section 5 concludes the paper.

2. Related Work

This section surveys relevant related work by first revisiting the definition of SDS and prior research in this domain. It then reviews alternative programming paradigms for the

[1] `https://www.docker.com/`
[2] `https://docs.docker.com/engine/swarm/`
[3] `https://tailscale.com/`
[4] `https://github.com/avilamatheus/self_distributing_systems/`

computing continuum, including Function-as-a-Service (FaaS), microservices, and stateful FaaS. Finally, it discusses the limitations of existing SDS implementations that hinder their deployment and full exploration in real-world edge–cloud continuum infrastructures.

Self-Distributing Systems (SDS), as defined in [Rodrigues-Filho et al. 2023, Rodrigues-Filho and Porter 2022, Costa et al. 2025], have been proposed as a programming and execution paradigm for applications deployed across the edge–cloud computing continuum, aiming to autonomously adapt their distribution at runtime in response to changing conditions. Foundational work formalizes SDS through component-based models in which application components can be dynamically relocated, replicated, or sharded based on observed metrics and learned decisions. In this context, intelligent placement mechanisms driven by machine learning or reinforcement learning play a central role in determining suitable distributed configurations. Frameworks such as Hatch [Rodrigues-Filho and Porter 2022] further abstract distribution complexity by providing transparent communication layers that enable runtime component mobility while preserving application semantics, demonstrating the feasibility of SDS as a general approach for the dynamic computing continuum.

Parallel to SDS research, Function-as-a-Service (FaaS) [Calavaro et al. 2025] and microservice-based architectures [Fu et al. 2022] have emerged as dominant programming paradigms for the computing continuum in the industry, largely due to their stateless nature which facilitates deployment. Recent efforts have extended these paradigms with support for stateful execution [Pfandzelter and Bermbach 2023, Cicconetti et al. 2021, Barcelona-Pons et al. 2019, Sreekanti et al. 2020], typically by coupling serverless functions or microservices with external storage systems or embedded caches to reduce access latency. While effective in many scenarios, these approaches often rely on generic state management strategies that treat state as a uniform abstraction, regardless of application-specific access patterns or consistency requirements. As a result, such solutions follow a "single-solution-fits-all" model that may lead to suboptimal performance or unnecessary overhead in scenarios where state behavior is tightly coupled with application logic or distribution dynamics, an aspect more explicitly addressed in SDS through fine-grained, adaptive state placement, and management.

Despite their conceptual advances, existing SDS implementations have primarily focused on learning strategies, system models, and runtime decision mechanisms, with limited attention to the deployment technologies required to operate across real computing continuum infrastructures. As a result, these solutions have not fully leveraged continuum-oriented technologies that enable homogeneous platforms spanning edge devices and cloud resources. In particular, aspects such as seamless connectivity across nodes without public addressing and lightweight orchestration suitable for both edge and cloud have remained underexplored. The absence of practical deployment abstractions hinders the realization of SDS in realistic environments, where simplified networking and orchestration mechanisms can significantly ease edge deployment and enable flexible component distribution across the continuum.

3. SDS_SWARM Architecture

SDS_SWARM integrates the Self-Distributing Systems (SDS) paradigm with container orchestrators, allowing an SDS application to programmatically place and redistribute

its components across the **edge–cloud continuum**. The approach integrates four layers: (i) secure connectivity across heterogeneous nodes, (ii) cluster-wide service management (placement, replication, and updates), (iii) a control-plane API that exposes orchestration primitives to the application, and (iv) the SDS application itself, executed as containers and capable of requesting its own redistribution at runtime. The architecture overview is shown in Fig. 1.

3.1. Design goals and rationale

SDS_SWARM is driven by three design goals. First, **practical deployability**: edge nodes are frequently behind NAT and do not provide stable public addressing, so the infrastructure must form a cluster without requiring inbound port exposure or complex network setup. Second, **lightweight orchestration**: the continuum nodes can be resource-constrained (especially the edge nodes), therefore the orchestration layer should be simple to operate while still supporting multi-node scheduling, service discovery, and elastic scaling. Third, **application-facing control**: the application must be able to request runtime reconfiguration, so we expose placement and scaling through a stable API rather than manual CLI actions.

3.2. Architecture overview

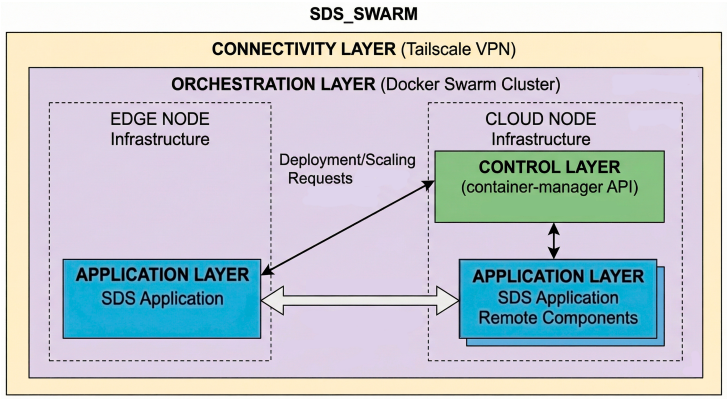

Figure 1. SDS_SWARM layered architecture overview. The system is structured as nested layers: Application and Control reside within the Orchestration layer, which is encapsulated by the Connectivity layer.

Connectivity layer (Tailscale VPN). Edge nodes commonly lack public, fixed IP addresses, which prevents straightforward cluster formation. As illustrated in Figure 2, a direct edge-cloud connection may fail when the edge node has no public IP. SDS_SWARM uses a device-oriented VPN, namely, Tailscale, to provide a private, location-transparent network that unifies geographically distributed nodes (edge and cloud) as if they were on the same LAN, reducing the need for manual port exposure and enabling secure inter-node communication.

Orchestration layer (Docker Swarm + overlay network). On top of the VPN, the infrastructure is organized as a Docker Swarm cluster. Swarm provides a low operational footprint while supporting multi-node scheduling, service management, and an **overlay network** that enables transparent container-to-container communication across hosts (Figure 3).

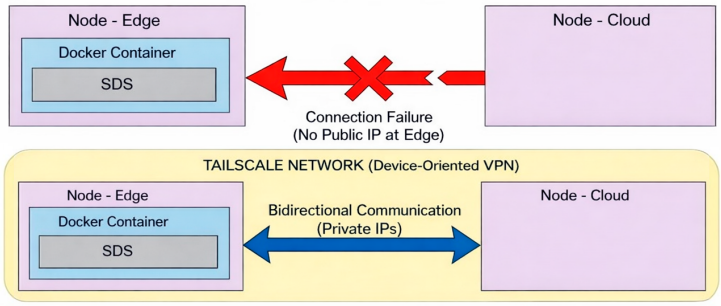

Figure 2. Tailscale private-IP overlay connectivity between edge and cloud nodes.

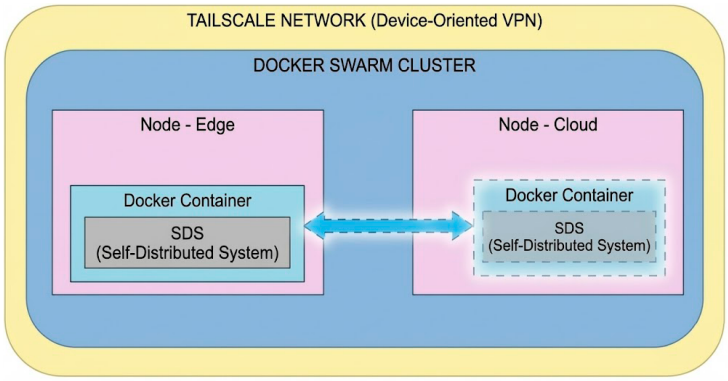

Figure 3. SDS deployed across edge and cloud nodes within a Docker Swarm cluster over a Tailscale overlay.

Control layer (container-manager API). The orchestration interface is exposed via `container-manager`, a Java/Spring Boot API that mediates between SDS application logic and Docker Swarm. It provides endpoints to declaratively request component instantiation and scaling in specific continuum locations (e.g., "edge", "cloud"), abstracting service creation and placement constraints behind HTTP calls.

Application layer (SDS). The SDS application is packaged as Docker containers and can programmatically request the deployment and scaling of its components across any location in the Swarm cluster via the control-plane API. After the cluster converges to the desired placement, a proxy component relies on Swarm's internal DNS (overlay network) to discover the remote components and dispatches RPC-over-HTTP calls to them.

3.3. Deployment model and placement

To enforce continuum locality, each Swarm node is labeled with its physical/logical location (e.g., `edge1`, `fog1`, `cloud1`). The control API translates an application request into Swarm **placement constraints** based on these labels, ensuring containers are scheduled in the intended region.

All services communicate via an attachable overlay network (e.g., `sds_network`), allowing containers to address each other by logical names resolved by Swarm's internal DNS, independently of the host they run on. This is critical for SDS scenarios, where the set of active workers changes over time and the application must avoid hard-coded endpoints.

3.4. Control API and runtime reconfiguration

The `container-manager` API runs on the Swarm **manager node**, since the manager enforces global cluster state and scheduling rules. Internally, it uses the `docker-java` library to manipulate Swarm services programmatically, replacing direct CLI usage with a stable code-level abstraction.

Two endpoints are central:

- **POST `/docker/start-containers`**: receives a JSON plan describing (i) a logical container name, (ii) the command to run, and (iii) a list of {`location`, `numberOfContainers`} deployments. The API creates/updates Swarm services using placement constraints derived from node labels.
- **GET `/docker/list-containers/{name}`**: returns logical identifiers of active instances whose names match the provided value, enabling the application to discover remote workers through the cluster namespace (without explicit IPs or ports).

Desired-state semantics. To support dynamic adaptation, the API follows a **desired-state** model. Given a new plan, it reconciles current services with the requested specification: scaling up/down replicas and (when needed) creating/removing services so that the resulting deployment matches the latest plan. This avoids incremental, error-prone imperative sequences and keeps reconfiguration logic explicit and auditable at the API boundary.

3.5. Service discovery and invocation

SDS_SWARM uses Swarm's internal DNS for service discovery. After requesting a distribution plan, the application queries `/docker/list-containers/{name}` to obtain the active instance's identifiers (or logical targets), and the proxy forwards RPC-over-HTTP requests to those targets. Because the overlay network provides uniform reachability, the proxy does not depend on host IPs, and workers can move across nodes as long as they remain attached to the overlay network.

3.6. End-to-end interaction flow

At runtime, SDS_SWARM follows a simple orchestration loop:

1. The SDS application determines (according to its execution logic) the desired distribution and builds a JSON distribution plan.
2. The plan is sent to `container-manager` via **POST `/docker/start-containers`**.
3. The API materializes the plan as Swarm services using placement constraints according to node location labels.
4. The application queries **GET `/docker/list-containers/{name}`** to obtain the identifiers of active remote instances.
5. The application proxy forwards RPC-over-HTTP calls to these identifiers; name resolution and routing are handled by the Swarm overlay network and internal DNS.
6. Work is distributed across edge and cloud resources according to the current plan.

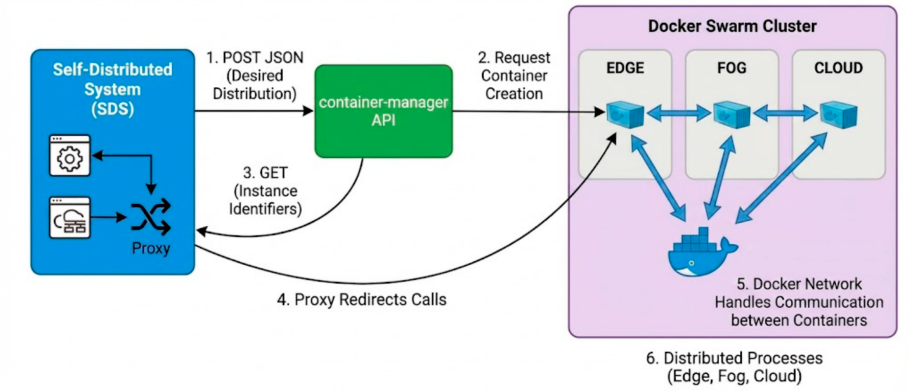

Figure 4. Runtime orchestration flow between the SDS application, the control-plane API, and the Swarm cluster.

4. Evaluation

This section evaluates the practical behavior of *SDS_SWARM* under controlled experiments. We focus on how distribution and parallelism across the edge–fog–cloud continuum affect the end-to-end response time of a CPU-bound workload, and whether the architecture can be reconfigured at runtime to migrate computation across the platform.

We use a reference Self-Distributing System (SDS) implemented in the Dana programming language [5]. The application exposes an HTTP API with two operations: (i) `POST`, which inserts integers into a list, and (ii) `GET`, which returns the current list and computes how many elements are prime numbers. The computation is intentionally CPU-intensive (deterministic trial division up to $\sqrt{n}$), making it suitable to highlight the impact of parallel execution and network latency.

The application is structured around a *List* component, which encapsulates both the state and the computation: it stores the inserted integers and, upon a `GET` request, performs the prime-number counting over the current collection.

The reference application supports two execution modes. In **local mode**, all components execute within a single process on the same host, resulting in centralized execution without parallelism. In **distributed mode**, the local *List* is replaced by a proxy component, which preserves the same behavior but delegates operations to a set of remote workers. Each worker is instantiated as an independent `remote-dist` process (container) that hosts its own *List* instance. Conceptually, `remote-dist` represents a replicated execution unit that maintains a partition of the data: `POST` requests are routed so that inserted values are spread across workers, and `GET` triggers the prime counting on each partition. The proxy then aggregates partial results to produce the final response, enabling multiple prime-counting tasks to execute concurrently.

4.1. Experimental Setup

The infrastructure comprised three nodes (viz., edge, fog, and cloud) interconnected through a private Tailscale VPN and organized as a single Docker Swarm cluster, with the edge node acting as the Swarm manager. The nodes were configured as follows: (i)

[5] `https://www.projectdana.com/`

Edge: Intel Core i5-7200U (2 Cores, 4 Threads), 6 GiB RAM; (ii) **Fog**: AMD Ryzen 5 4600G (6 Cores, 12 Threads), 32 GiB RAM; (iii) **Cloud**: Google Cloud Compute Engine (region `southamerica-east1`), N1 series, 32 vCPUs and 32 GiB RAM.

To ensure comparability across experiments, the SDS application container and the `container-manager` API remained on the **edge node** in all scenarios. The only variable was the *location* and *number* of `remote-dist` worker processes instantiated by *SDS_SWARM*. All required Docker images were preloaded on each node to avoid measurement interference from image downloads.

4.2. Workload and Measurement

Each run follows the same procedure. First, a `POST` request populates the list with a fixed number of integers. Then, a `GET` request triggers the prime-number counting. Our primary metric is the `GET` response time (ms), measured by an external Python script executed on the edge node. The script interacts with the system exclusively through HTTP and is not part of the *SDS_SWARM* architecture.

4.3. Scenarios

We evaluated five scenarios:

1. **Scenario 1 – Local mode (baseline).** All components execute in a single process on the edge node.
2. **Scenario 2 – Distribution on the edge.** The workload is distributed across **4** `remote-dist` containers on the edge node, isolating the effect of parallelism with minimal network latency.
3. **Scenario 3 – Distribution on the fog.** The workload is distributed across **8** `remote-dist` containers on the fog node, combining increased compute capacity with low-latency communication.
4. **Scenario 4 – Distribution on the cloud.** The workload is distributed across **16** `remote-dist` containers on the cloud node, exploring high compute capacity with high latency.
5. **Scenario 5 – Dynamic execution.** The run starts in local mode and is reconfigured at runtime to migrate the workload across different distribution topologies (edge, fog, and cloud), illustrating the adaptive behavior enabled by *SDS_SWARM*.

4.4. Results

4.4.1. Scenario 1 – Local mode (baseline)

The first scenario evaluates **local execution**, which serves as the baseline for all subsequent comparisons. In this mode, the entire application runs fully centralized on the **edge node** as a **single process**, with no distribution and no parallelism (Figure 5).

Figure 5 shows that the `GET` response time increases rapidly as the list grows. With 40 items, the request takes approximately 2000 ms. When the workload doubles to 80 items, the response time rises to about 8000 ms, roughly a fourfold increase for only twice the input size. Near 100 items, the response time exceeds 12000 ms (12 s), indicating a steep degradation in responsiveness.

This behavior is expected for a CPU-bound prime-number verification workload. As the list size increases, more primality checks must be executed sequentially, and the lack of parallelism makes the edge node the dominant bottleneck.

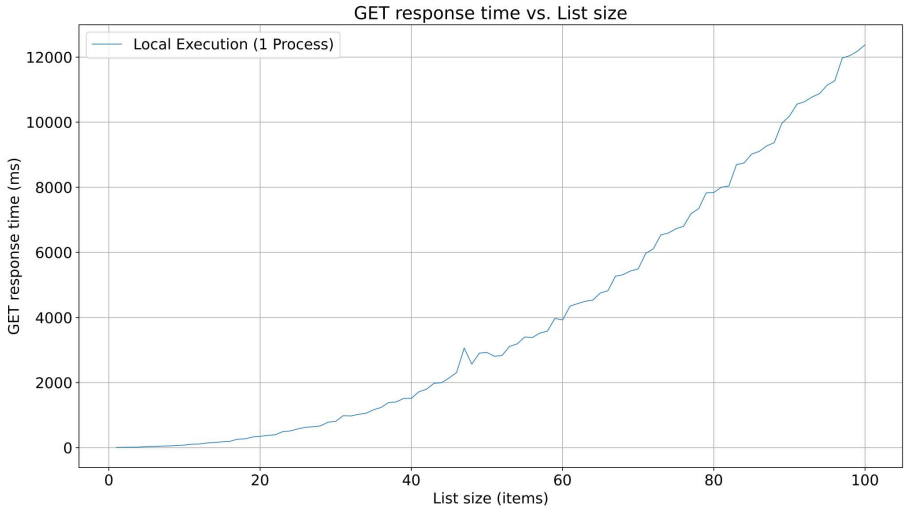

Figure 5. Local execution baseline: `GET` response time as a function of list size (single process on the edge node).

4.4.2. Scenario 2 – Distribution on the edge

Figure 6 evaluates the first optimization level: distributing the workload *within* the edge node itself. In this configuration, the application runs in distributed mode and delegates the prime-number computation to four `remote-dist` worker processes executing in parallel on the same device.

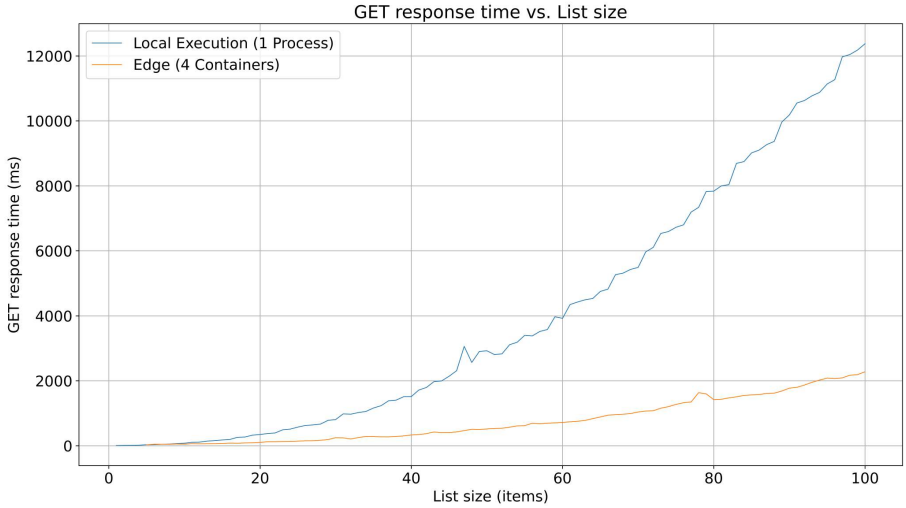

Figure 6. Scenario 2 (edge distribution): `GET` response time vs. list size comparing local execution (single process) against distributed execution on the edge with 4 `remote-dist` containers.

The results show a substantial performance improvement. The edge-distributed curve grows much more slowly than the local baseline. For a workload of 100 items, local execution takes more than 12 s, whereas the distributed execution on the edge completes in approximately 2.3 s, reducing execution time by a factor of more than five. This confirms that parallelization is effective even on a resource-constrained edge node.

An important detail is that the edge-distributed series only appears from 5 items onward in our measurements. This is consistent with the reference application's distribution logic, which requires the list to contain at least one item per instantiated worker; with 4 containers, very small list sizes do not provide enough work to distribute meaningfully. For low workloads (below roughly 7 items), both modes achieve similar response times, which is expected due to the fixed overhead of coordinating multiple workers. Beyond this point, the distributed mode becomes consistently faster and the gap widens as the list size increases.

4.4.3. Scenario 3 – Distribution on the fog

Figure 7 evaluates the offloading of the workload from the edge to the **fog** node. Compared to the edge device, the fog node provides substantially higher compute capacity, which allowed us to double the number of `remote-dist` worker instances (from 4 to 8).

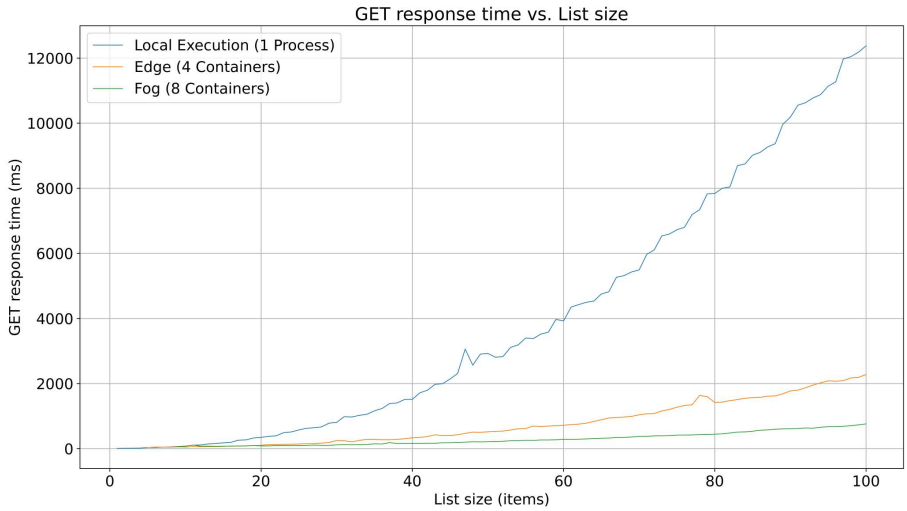

Figure 7. Scenario 3 (fog distribution): `GET` response time vs. list size comparing local execution, edge distribution (4 `remote-dist` containers), and fog distribution (8 `remote-dist` containers).

The results show another marked performance improvement. The fog curve (green) is significantly flatter than the edge-distributed curve (orange), indicating better scalability as the list grows. With 100 items, the fog completes the `GET` request in under 800 ms, roughly three times faster than the edge distribution, which requires about 2300 ms for the same workload.

A relevant detail is the behavior under very small workloads. For list sizes below approximately 10 items, the fog can be slightly slower than the edge. This is explained by network overhead: even with low latency, remote calls between the edge and fog nodes introduce additional delay. When the workload is small, this communication cost can outweigh the benefits of a faster CPU and increased parallelism. As soon as the input size becomes moderately larger, the fog node compensates for this initial overhead and remains consistently faster thereafter.

4.4.4. Scenario 4 – Distribution on the cloud

Figure 8 evaluates offloading the workload to the **cloud** node. Due to the substantially higher compute capacity available in the cloud environment, we increased the level of parallelism once more, instantiating **16** `remote-dist` worker containers, the highest number among all scenarios.

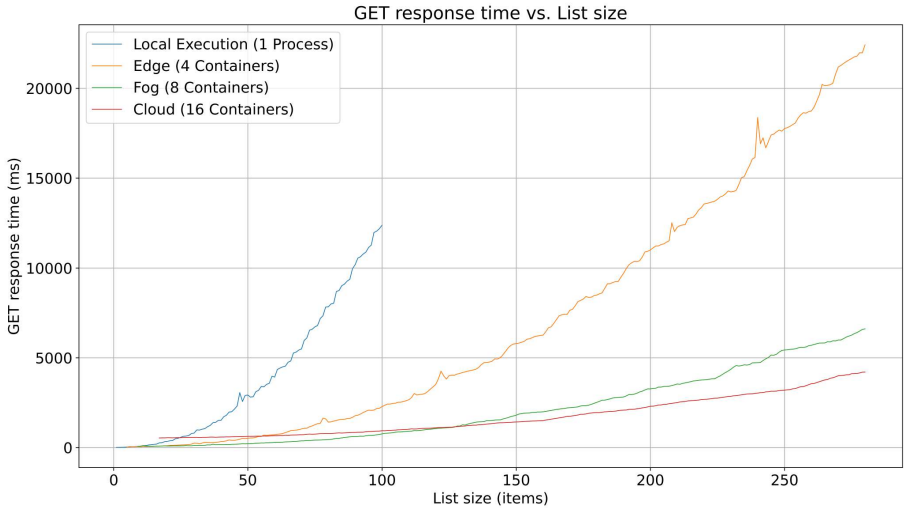

Figure 8. Scenario 4 (cloud distribution): `GET` response time vs. list size comparing local execution, edge distribution (4 containers), fog distribution (8 containers), and cloud distribution (16 containers).

Looking at the overall behavior for larger workloads, the cloud configuration is the most scalable. The cloud curve (red) grows more slowly and steadily than the fog (green) and edge (orange) curves, indicating better performance as the list size increases. In contrast, the edge distribution degrades sharply as the workload grows, and the fog curve starts to exhibit a steeper slope at higher list sizes, suggesting that it becomes increasingly compute-bound.

Note that the local-execution baseline (blue) is not shown beyond 100 items. Data collection was intentionally stopped at that point because response time had already exceeded 12 s, making local execution impractical for larger workloads. Extending the baseline further would also compress the plot scale and hinder the visual comparison between distributed scenarios.

However, the initial portion of the plot highlights the direct impact of cloud latency. For relatively small list sizes (below approximately 50 items), the cloud exhibits the worst performance among the distributed modes. In this region, the communication overhead between the edge and cloud dominates the total response time, outweighing the benefits of additional compute resources. Only when the computational workload becomes sufficiently large does the cloud's processing capacity compensate for this initial cost, after which it becomes the most advantageous option.

Overall, this scenario illustrates a key trade-off in the edge–fog–cloud continuum: cloud execution is highly effective for heavy workloads, but can be inefficient for short, latency-sensitive tasks.

4.4.5. Scenario 5 – Dynamic execution

After evaluating the static scenarios, we assess the main contribution of *SDS_SWARM*: the system's **dynamic** behavior. In the previous experiments, each configuration was fixed (local, edge, fog, or cloud). Here, we evaluate whether the architecture can adapt at runtime by progressively relocating the computation across the continuum as the workload changes.

In our experiments, the adaptation flow followed a progressive sequence: (i) **Local execution** on the edge (from 0 to approximately 12 items); (ii) **First adaptation** at approximately 12 items, from local to **edge distribution**; (iii) **Second adaptation** at approximately 50 items, from edge to **fog distribution**; (iv) **Final adaptation** at approximately 150 items, from fog to **cloud distribution**.

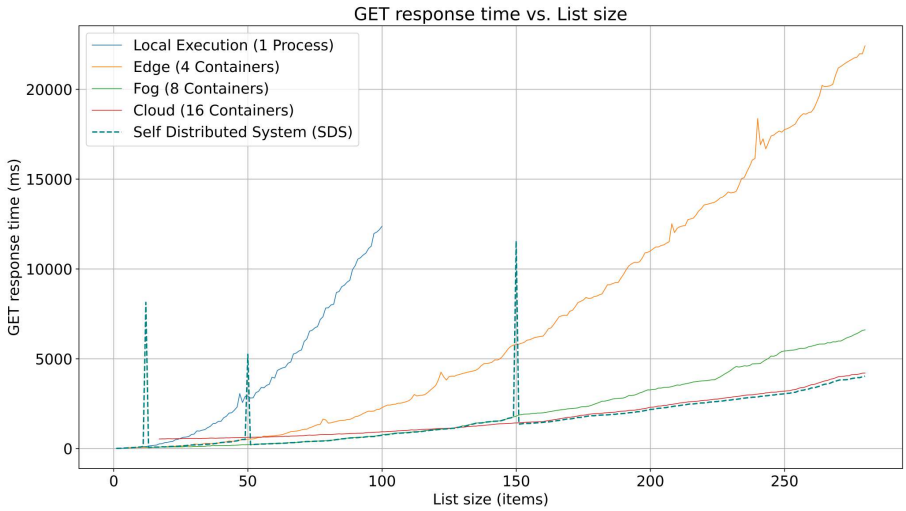

Figure 9. Scenario 5 (dynamic execution): `GET` response time vs. list size comparing static placements (local, edge, fog, cloud) against the dynamic SDS behavior.

In this scenario, the reference application operates as a self-distributing system: as the list grows, it requests a new deployment topology spanning edge, fog, and cloud nodes in order to use the most suitable resources at each phase. In self-distributing systems, reinforcement learning is commonly used to decide *when* and *where* to migrate components. In this work, however, we keep the decision-making **manual** to validate the underlying infrastructure and orchestration mechanisms required to support such adaptations.

Figure 9 shows the response-time curve of the dynamic SDS execution (dashed line) alongside the static baselines. The SDS curve can be interpreted as a composition of the best-performing strategies across the run: within each workload range, it tracks the placement that provides the lowest response time at that moment.

The most notable feature is the presence of **vertical spikes** at specific points. These spikes are not measurement errors; they represent the **cost of adaptation**, i.e., the time overhead required for the system to reconfigure. At each spike, the SDS issues a redistribution request to the `container-manager` API, which triggers Docker Swarm to instantiate and redeploy the required `remote-dist` workers. Although this orches-

tration step introduces a temporary delay, it is compensated by the lower response times observed after the new placement becomes active.

4.5. Discussion

The results indicate that *SDS_SWARM* can support the execution of Self-Distributing Systems across an edge–fog–cloud continuum by combining container-based portability with runtime reconfiguration. Across the static scenarios, increasing parallelism and moving the computation from the edge to fog and cloud reduced the `GET` response time substantially for larger workloads, while also exposing a clear latency–compute trade-off: cloud execution becomes advantageous only when the workload is sufficiently heavy, whereas for small inputs its higher network latency dominates the end-to-end response time.

Most importantly, the dynamic experiment demonstrates that the architecture can progressively adapt the deployment as the workload grows, transitioning across multiple compositions and sustaining response times close to the best-performing static placement in each range. The vertical spikes observed during the dynamic run capture the overhead of orchestration and redeployment, showing that adaptation has a measurable cost that must be amortized by subsequent performance gains.

Although the adaptation decisions were manually triggered in this study, the observed behavior confirms that the proposed infrastructure and orchestration mechanisms are operational and can support runtime relocation of components. This provides a concrete basis for future work on autonomous decision-making (e.g., reinforcement learning policies) to determine when and where to reconfigure the system.

5. Final Remarks

This paper presented SDS_SWARM, an extension of the Self-distributing Systems (SDS) paradigm for building stateful applications across the edge–cloud computing continuum. We discussed how Docker and Docker Swarm can be used to package applications into containers and to manage their execution across a diverse set of computing resources, spanning both edge devices and cloud infrastructures.

By leveraging the SDS paradigm, SDS_SWARM enables the autonomous distribution and replication of application components, improving scalability and resource efficiency. Our evaluation shows that SDS-based applications scale effectively across edge-only, hybrid edge–cloud, and cloud-only environments, making efficient use of available computational resources in realistic deployments. As future work, we plan to extend Dana-based components to resource-constrained edge devices and explore autonomous mechanisms for optimal component placement.

Acknowledgments

The authors used ChatGPT only for language revision. No AI was used in any other part of the research. This research was partially sponsored by CNPq grant # 405940/2022-0 and CAPES grant # 88887.954253/2024-00.

References

Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P., and García-López, P. (2019). On the faas track: Building stateful distributed applications with serverless

architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA. Association for Computing Machinery.

Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., Villas, L., DaSilva, L., Lee, C., and Rana, O. (2018). The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, 3–4:134–155.

Bittencourt, L. F., Rodrigues-Filho, R., Spillner, J., De Turck, F., Santos, J., da Fonseca, N. L. S., Rana, O., Parashar, M., and Foster, I. (2025). The computing continuum: Past, present, and future. *Computer Science Review*, 58:100782.

Calavaro, C., Cardellini, V., Presti, F. L., and Russo, G. R. (2025). Beyond cloud: Serverless functions in the compute continuum. *SN Computer Science*, 6(3):194.

Cicconetti, C., Conti, M., and Passarella, A. (2021). On realizing stateful faas in serverless edge networks: State propagation. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 89–96.

Costa, B. P., da Silva, H. H., Morales, A. S., Bittencourt, L. F., Panisson, A. R., and Rodrigues-Filho, R. (2025). A multi-agent approach to self-distributing systems. In Barolli, L., editor, *Advanced Information Networking and Applications*, pages 128–139, Cham. Springer Nature Switzerland.

Fu, K., Zhang, W., Chen, Q., Zeng, D., and Guo, M. (2022). Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1825–1840.

Gama, E. S., Rodrigues-Filho, R., Madeira, E. R. M., Immich, R., and Bittencourt, L. F. (2024). Enabling adaptive video streaming via content steering on the edge-cloud continuum. In *2024 IEEE 8th International Conference on Fog and Edge Computing (ICFEC)*, pages 35–42.

Gao, G. and Wen, Y. (2021). Video transcoding for adaptive bitrate streaming over edge-cloud continuum. *Digital Communications and Networks*, 7(4):598–604.

Pfandzelter, T. and Bermbach, D. (2023). Enoki: Stateful distributed faas from edge to cloud. In *Proceedings of the 2nd International Workshop on Middleware for the Edge*, pages 19–24.

Rappa, F. M., Rodrigues-Filho, R., Panisson, A. R., Marcolino, L. S., and Bittencourt, L. F. (2024). Multi-armed bandits for self-distributing stateful services across networking infrastructures. In *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pages 1–6.

Rodrigues-Filho, R., Dias, R. S., Seródio, J., Porter, B., Costa, F. M., Borin, E., and Bittencourt, L. F. (2023). A self-distributing system framework for the computing continuum. In *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10.

Rodrigues-Filho, R. and Porter, B. (2022). Hatch: Self-distributing systems for data centers. *Future Generation Computer Systems*, 132:80–92.

Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J. E., Hellerstein, J. M., and Tumanov, A. (2020). Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452.