Komet: A Serverless Platform for Low-Earth Orbit Edge Services

Tobias Pfandzelter
Technische Universität Berlin & Einstein Center Digital Future
Berlin, Germany
tp@3s.tu-berlin.de

David Bermbach
Technische Universität Berlin & Einstein Center Digital Future
Berlin, Germany
db@3s.tu-berlin.de

Abstract

Low-Earth orbit satellite networks can provide global broadband Internet access using constellations of thousands of satellites. Integrating edge computing resources in such networks can enable global low-latency access to compute services, supporting end users in rural areas, remote industrial applications, or the IoT. To achieve this, resources must be carefully allocated to various services from multiple tenants. Moreover, applications must navigate the dynamic nature of satellite networks, where orbital mechanics necessitate frequent client hand-offs. Therefore, managing applications on the low-Earth orbit edge will require the right platform abstractions.

We introduce Komet, a serverless platform for low-Earth orbit edge computing. Komet integrates Function-as-a-Service compute with data replication, enabling on-demand elastic edge resource allocation and frequent service migration against satellite orbital trajectories to keep services deployed in the same geographic region. We implement Komet as a proof-of-concept prototype and demonstrate how its abstractions can be used to build low-Earth orbit edge applications with high availability despite constant mobility. Further, we propose simple heuristics for service migration scheduling in different application scenarios and evaluate them in simulation based on our experiment traces, showing the trade-off between selecting an optimal satellite server at every instance and minimizing service migration frequency.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11
https://doi.org/10.1145/3698038.3698517

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; *Cloud computing*; • **Networks** → **Network services**.

Keywords

edge computing, satellite networks, serverless computing

ACM Reference Format:
Tobias Pfandzelter and David Bermbach. 2024. Komet: A Serverless Platform for Low-Earth Orbit Edge Services. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3698038.3698517

1 Introduction

Sixth-generation mobile networks will be defined by an increasing focus on edge computing and the integration of non-terrestrial networks such as low-Earth orbit (LEO) satellite constellations [9, 24, 37]. For this, edge computing brings compute and storage resources closer to clients, increasing both application service quality for users and enabling entirely new application domains with low-latency, high-bandwidth access to resources [8, 14, 57]. At the same time, satellite networks comprising thousands of satellites in LEO will enable global access to high-bandwidth communications and Internet access [9, 9, 11, 51]. While both edge computing and LEO networking have received considerable research and industry attention and are already commercially available, their combination, i.e., LEO edge computing, is still an emerging research field [10, 12, 13, 37, 45, 48, 61].

LEO edge computing refers to the integration of edge computing resources with LEO satellites, i.e., placing processors and storage on communication satellites. Similar to terrestrial edge computing, the LEO edge could provide low-latency, high-bandwidth access to application services such as online collaborative drawing, multiplayer games, metaverses, web services, or the remote IoT, albeit for a global basis of subscribers. A unique challenge of LEO edge computing is service orchestration: Orbital dynamics dictate that LEO satellites must move at speeds in excess of 27,000km/h

in relation to Earth, which results in frequent (on the order of 4–5 minutes) client handovers [11, 24, 35]. An edge service deployed on a LEO satellite will quickly be out of reach of a client, negating the benefits of proximity between clients and services.

The solution to this mobility is 'virtual stationarity', where services transparently remain in client proximity by frequent service migration thus offsetting the satellites' movements [10]. This requires that service migration is relatively cheap and can be completed without downtime. Unfortunately, as we shall see in §3, this is not the case for the state-of-the-art service deployment model with container orchestration, e.g., using Kubernetes [12]: Frequently migrating stateful containers can lead to considerable service downtime during checkpoint, transfer, and restore operations.

Instead, we propose leveraging serverless abstractions for LEO edge computing. By explicitly decoupling compute and storage services, we can deliver a large variety of LEO edge application services with virtual stationarity without downtime. We make the following contributions:

- We introduce Komet, a serverless platform for the LEO edge that combines an edge Function-as-a-Service (FaaS) compute platform with a distributed data management layer and automatically migrates services to provide virtual stationarity (§4).
- With Komet, we introduce heuristics for scheduling LEO edge services (§5).
- We demonstrate and evaluate Komet with a proof-of-concept prototype using typical edge computing applications on *Celestial* [45] LEO edge testbeds.
- In simulations based on experiment traces, we investigate the trade-off between migration frequency and service level in scheduling LEO edge compute services (§8).

We make our implementation artifacts available as open-source software.[1]

2 Background & Related Work

To understand the challenges associated with service availability in LEO edge computing, we first give an overview of the characteristics of LEO satellite networks, the goals of edge computing in general, and the state of the art in LEO edge research.

[1] https://github.com/3s-rg/komet

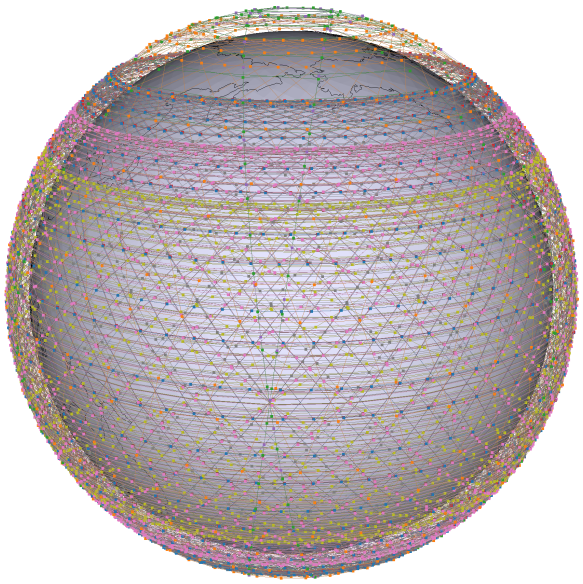


Figure 1: In its current deployment, the Starlink LEO satellite constellation comprises 4,409 satellites to achieve global coverage (screenshot from the Celestial emulation toolkit [45]). Lines between satellites indicate inter-satellite links (ISL).

2.1 LEO Satellite Networks

Large LEO satellite networks promise global low-latency, high-bandwidth Internet access [9, 24, 30, 42]. While traditional satellite Internet access using satellites in geostationary orbits has existed for decades, the high satellite altitude (more than 35,000km) incurs a high (more than 500ms round-trip time (RTT)) access delay. LEO satellite networks use orbital altitudes of 500km to 1,000km, promising higher bandwidth (given reduced radio power requirements) and reduced access latency.

Two characteristics of LEO satellites shape how LEO constellations achieve global network coverage: First, a satellite that is close to Earth has a reduced cone of coverage, i.e., a single satellite can only serve a limited region on the ground. Second, satellites in LEO move at speeds in excess of 27,000km/h, with an orbital period of one to two hours [9, 11]. Continuous global coverage by satellites thus requires constellations of hundreds or thousands of connected satellites. As shown in Figure 1, SpaceX operates 4,409 Starlink satellites evenly spaced around Earth to achieve such global coverage [35]. Other plans include Amazon Kuiper's proposed 3,236-satellite constellation and the 1,671-satellite Telesat constellation [35]. To cover large distances, e.g., between an uplink station and a remote client, satellites use high-bandwidth inter-satellite links (ISL), and satellite networks essentially perform multi-hop routing between clients [11].

2.2 Edge Computing

Edge computing extends the on-demand, elastic compute and storage resources of cloud computing throughout the access network, enabling application developers to provide their service in proximity to service consumers, e.g., IoT devices or mobile clients [14, 57]. While there are competing definitions of the term, we consider edge computing resources to be servers located at the access network, e.g., at a mobile radio tower, serving multiple clients with application services provided by multiple tenants [8, 14]. Here, edge computing provides benefits in terms of service access latency and bandwidth as well as resiliency to backbone outages and contention. Nevertheless, edge computing also introduces challenges, such as service management across geo-distributed nodes or resource allocation in environments that are more constrained than the cloud [43, 52].

These challenges also apply when transferring the concept of edge computing to LEO satellite networks, i.e., the *LEO edge* [10, 12, 13, 17, 39, 44, 46, 48]. As LEO networks are another kind of radio access network, albeit with access points on satellites in space, it can be envisioned that these access points may also provide compute resources that can be used by application providers to host services for network clients. Here, resource management and especially resource sharing among tenants become more pressing concerns given the power and weight limits of satellites. While research has shown that equipping network satellites with sufficient edge compute resources is possible [10], it becomes paramount that any available resources are allocated as efficiently as possible. Further, there is the challenge of node mobility, as LEO satellites frequently (every 4–5 minutes) move out of visibility of clients, requiring service migration to provide continuous coverage [12, 42]. As no deployed LEO edge systems are currently available, researchers use virtual LEO edge simulators, e.g., *SatEdgeSim*, or emulators, e.g., *Celestial* [45] and *StarryNet* [37], to investigate the behavior of LEO edge applications.

2.3 Serverless Computing

One promising approach to manage application services by multiple tenants efficiently while still providing high levels of service isolation is serverless computing [27, 32]. In serverless computing, operational concerns are shifted to compute platform providers, while application developers focus on business logic. Allocating underlying infrastructure, managing the core software stack, and elastically scaling services are all performed by the platform. While this has obvious benefits for application developers, in the context of edge computing this shift of operational concerns to a shared underlying platform can also help achieve more efficient resource allocation [5, 43, 52].

Function-as-a-Service (FaaS) as the most prominent serverless programming model allows developers to design and deploy applications as collections of small, loosely coupled *functions* [27, 32]. Each function is invoked with a single message or event, has the ability to invoke further functions or interact with external services such as data stores, and (optionally) responds with a return value. FaaS functions run on a FaaS platform that elastically scales function handlers (that run the function) in response to incoming events, including scaling to zero when no computation is necessary. Functions can share their underlying runtimes, e.g., multiple functions written in the Python programming language can share a hardware and software stack, making them lightweight and easy to distribute over the network.

These attributes make the FaaS programming and execution model a good fit for edge computing [43, 52]: Sharing a hardware and software stack means that limited edge resources can be shared to a high degree and elastically scaling function resources (by creating and destroying handlers) in response to demand enables a fine-grained allocation of these resources over time.

To build stateful applications on top of stateless FaaS, (serverless) data management and synchronization services can be integrated [16, 23, 28, 47, 54, 58]. While trivial in the cloud, this poses challenges for distributed edge deployments where function instances in different geographic locations need to access the same data, introducing a trade-off between the goal of local function execution and the need to synchronize data access. While still an area of active research, current proposals use, e.g., request routing to execute functions near their data dependencies [23, 58] or commutative replicated data types (CRDTs) to enable distributed concurrent execution without synchronization [31, 47, 56].

2.4 Related Work

Bhattacherjee et al. [10] propose the concept of serving edge applications from LEO satellites using a concept they call 'virtual stationarity'. Virtual stationarity provides an edge service from the same geographic location despite the highly dynamic LEO network infrastructure by handing off application state along with the hand-offs of clients, i.e., as satellites move along their orbits, services are moved in the opposite direction. Further, the authors show two methods for server selection: A *MinMax* approach selects the closest server to a client at every instant, resulting in frequent changes. An alternative *Sticky* approach aims to minimize state transfers by predicting which satellite with a reasonable distance to the client (within 10% of the optimum) will be in view of the client for the longest duration and has the smallest hand-off latency, thus trading off a small increase in access latency for less frequent hand-offs and hand-offs with low latency.

While these approaches are interesting theoretical mechanisms for managing LEO edge services, the authors provide no platform abstraction into which services can be deployed.

Existing work on LEO edge application platforms focuses largely on container orchestration, similarly to research on terrestrial edge computing. Bhosale et al. [12] present *Krios*, an extension of Kubernetes for LEO satellite edge computing. Kubernetes orchestration is reactive, i.e., the scheduler waits for an issue or error (possibly along with timeouts or retries) before deciding to reschedule a service. In the context of LEO edge computing, the authors find that this leads to unnecessary downtime when a satellite that provides an application service for a client leaves the visibility of that client, as Kubernetes does not natively understand that the edge server can move away from a client. Krios extends Kubernetes with proactive scheduling that takes satellite orbit models into account: Krios will automatically schedule an application hand-off shortly before a satellite loses connection to a client, achieving continuous virtual stationarity. As we discuss and show in §3, however, using container scheduling for LEO edge computing requires that container services are stateless, as migrating stateful containers comes with the cost of service downtime. This could only be mitigated by explicitly designing the containerized service to externalize state or to gracefully handle concurrent instantiation at different satellite nodes without inconsistent data. While this is indeed possible, it would expose the challenges of LEO edge computing to the application developers and essentially negate any developer benefits of platform abstractions.

C. Wang et al. [60] and S. Wang et al. [62] propose a *cloud-native* satellite architecture. This architecture uses cloud-native technologies such as Docker, Kubernetes, and edge-cloud networking on top of virtualized computing and storage to build flexible satellites. The authors have successfully implemented this architecture on the *BUPT-1* satellite that is part of the planned *Tiansuan* constellation [61, 64], showing that containerized computing in LEO is feasible. This architecture targets isolated satellites or small satellite clusters rather than constellations of thousands of networked satellites, however, making their use-case and requirements different from the ones we discuss in this paper.

In previous work [48], we proposed using FaaS as a LEO edge execution paradigm. The combination of stateless functions and an external data management system supports migration of services against the orbital movement of LEO satellites that is transparent to clients, who experience continuous service availability, and application developers, who have explicit constraints on how to manage state in their applications by the programming model. We have hitherto not provided an implementation or integrated architecture for this approach and did not evaluate its feasibility experimentally before this work.

These works all build on a strong foundation of research on orchestration and scheduling in terrestrial edge computing including work on edge container orchestration [6, 15, 31, 53], serverless edge computing [5, 52, 55, 63], and stateful edge-to-cloud computing [18, 23, 47, 58]. The mobility of LEO edge satellite servers, however, means that we cannot directly use these existing approaches for LEO edge computing [10, 48].

3 Downtime in LEO Edge Container Migration

Using container orchestration for LEO edge services, such as in *Krios* [12], requires service migration to achieve virtual stationarity. To migrate a (stateful) container, it first needs to be checkpointed, i.e., the container is stopped and its memory content is written to disk. Second, that checkpoint needs to be transferred to the new execution location, e.g., the next satellite a client will be handed off to. Third, the container can be restored from the checkpoint on the new location. We argue that this process, which has to be repeated continuously throughout the service's lifetime, leads to considerable service downtime.

Consider a simple example: A client in Redmond, WA, USA, accesses an in-memory store on a LEO satellite in a single orbital plane of the phase I Starlink constellation (550km altitude, 22 satellites per orbital plane, i.e., 16.4° spacing between nodes [35]). The mean contact length between the ground station and a satellite is 260 seconds, i.e., for an optimal latency between client and server, the container that hosts this service must be migrated every 260 seconds.

We implement this scenario on top of the Celestial LEO edge emulation toolkit [45] using a *Redis* Docker container based on the lightweight `redis:alpine` image. We use the *Checkpoint/Restore In Userspace* (CRIU) software [20] in combination with the *Podman* container manager [50] for checkpoint-restore operations. Our client loads data into the container's memory through the Redis API and continuously reads single values. When the client is handed off to a new satellite, we checkpoint the container on the source satellite, migrate this checkpoint, and restore the container. We measure checkpoint, transfer, and restore time for our service for different data sizes between 0 and 1,000MB. The ground station and each satellite server have 4 vCPUs and 8GB of memory and run Alpine Linux 3.18, we assume 10Gbps bandwidth for ISLs and ground-to-satellite links. Our host server has two 12-core Intel™ Xeon™ Silver 4310 2.10GHz CPUs, 64GB memory, SSD storage, and runs Ubuntu 22.04 LTS. We run each experiment for 15 minutes and repeat it ten times.

Unsurprisingly, the results in Figure 2 show a linear correlation between the size of data in container memory and the migration time. In our experiments, there is a minimum

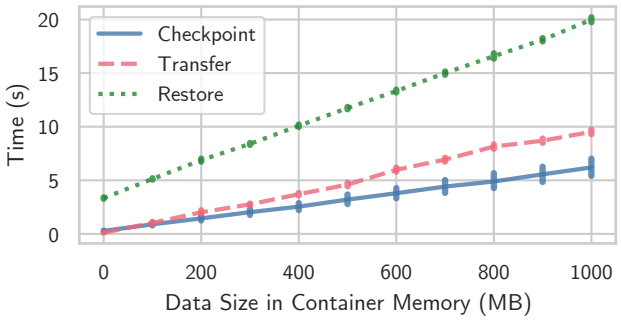


Figure 2: Container checkpoint, transfer, and restore times in our container migration experiment. It is obvious that the total migration time for a stateful container grows linearly with the amount of memory the container uses.

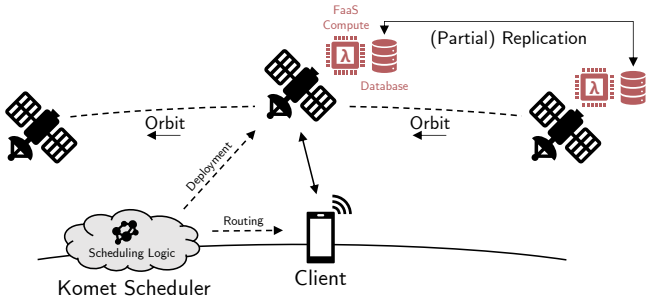


Figure 3: Komet comprises a per-satellite FaaS compute platform and a replicated data store, hosting replicas of the application service on multiple satellites.

mean 3.82s migration delay (dominated by a mean 3.39s restore time) even without any data in the container's memory, simply to restore the started and configured Redis service. For 1,000MB data, this migration delay grows to a mean 35.7s. With a service migration every 260s to mitigate the effect of satellite orbital movement, this would lead to a considerable ~14% downtime. Note that this includes only application state, while the Redis base image already exists at each host. We expect similar replication costs if full virtual machines, microVMs, or unikernel VMs were used instead of containers, as process transfer costs remain.

While these results may not be representative of all hardware configurations and there are indeed advances in the efficiency of container migration [33, 34, 40], it is clear that such an approach that is transparent to application developers, i.e., the software is not adapted for frequent migration, is *not* transparent for service clients.

4 Komet Architecture

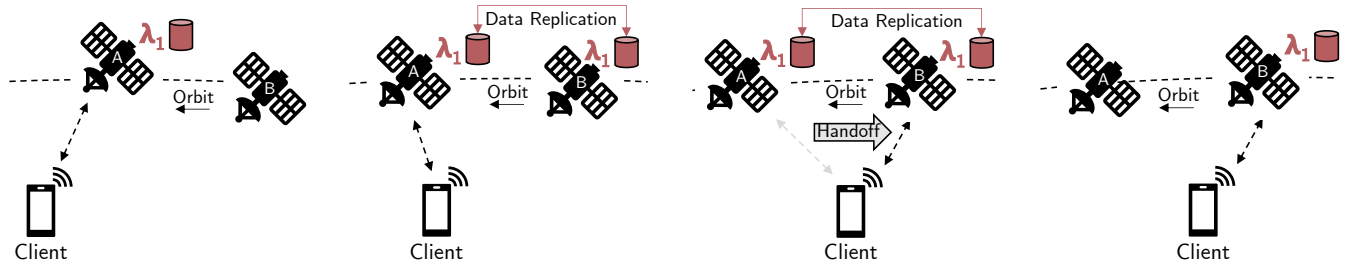
To address the complexity of building LEO edge applications that can be transparently migrated, we propose Komet, a serverless platform for the LEO edge. Komet is built on the intuition that decoupling state from compute enables transparent live migration of services and that the semantics of FaaS are a well established approach for such decoupling. The core technique that enables this is the concurrent deployment of two function instances backed by replicated data during the migration of a service.

4.1 Application Runtime

Komet thus comprises two components on each satellite node: A single-node FaaS platform that supports elastically running application services as FaaS functions and a data replication component that provides the stateful backend for those services. We show the architecture of Komet in Figure 3. In Komet, each satellite server hosts a FaaS compute platform and a database system. Komet deploys FaaS functions on a satellite server near the client that wishes to invoke them. Alongside the function handlers, each function has its own data pool in the database system to keep state across function invocations. This data can be replicated during migrations to ensure that a client can seamlessly switch from one satellite service to the next. As the function code itself is stateless, replicating only the data is sufficient for the service to transparently appear as a monolithic entity rather than a distributed application. This has the additional benefit that only application state has to be replicated rather than the entire software stack, as with container migration. The high-bandwidth, low-latency ISLs between satellites make data replication easily possible.

4.2 Migration

Data replication in Komet is also used to transparently migrate the service against the satellites' orbital movement, as shown in Figure 4. By default, a client expects the edge service it uses to be located on a nearby satellite server (Figure 4a). As the satellite network evolves, i.e., the satellites follow their orbital paths, Komet proactively replicates the edge service to the selected next satellite (Figure 4b). Here, only the state in the database is replicated, while the actual application processes are started anew, requiring no expensive process checkpointing. Note that the highly predictable behavior of satellite orbits makes such calculations trivial and accurate. When the client connection is handed off to the next satellite, the client can continue to access its edge service at the new location, without being aware of service migration occurring in the background (Figure 4c). As soon as the original satellite serving the client is out of view, the



(a) Client accesses service λ_1 at satellite A

(b) Service λ_1 is proactively replicated to satellite B

(c) As client uplink is handed off to satellite B, client transparently continues to access λ_1 at client's new satellite B

(d) The copy of λ_1 can be removed from satellite A when it leaves client visibility

Figure 4: Service Migration in Komet

service copy at that satellite can be deleted to reduce the replication costs (Figure 4d). Finally, the process can be repeated for the next migration.

4.3 Scheduler

The migration of LEO edge services requires some coordinating entity that deploys Komet function replicas and synchronizes hand-offs between satellite service and client. We propose a per-application, centralized (e.g., cloud-based) Komet scheduler. This scheduler considers the global network state, i.e., clients, deployed services, network connections, and predicted network evolution based on satellite trajectories, and schedules service deployments accordingly. Specifically, the scheduler is responsible for (i) (proactively) deploying functions and data replicas to satellite servers, (ii) informing clients when they should connect to a new satellite server, and (iii) removing functions and data replicas from servers when they are no longer used. While a central scheduler may be less scalable than a distributed, on-satellite scheduling component, it is easier to reason about, and we believe that it does not impact service quality, as scheduling is asynchronous and not on the critical path of client requests to edge services. Furthermore, satellite movement is so predictable that it would in fact be possible to schedule (and distribute) a migration plan that covers several days with at-runtime scheduling only sending minor live adaptations and synchronizing all clients and servers to perform migration at roughly the same time. This would effectively remove the single point of failure, simply by having extensive time for recovery in case of failures. Note also that centralized scheduling is not uncommon in satellite networks: SpaceX is believed to use a central scheduler for its entire Starlink network, updating ground stations and routes every 15s [30, 42]. Still, we plan to investigate a distributed scheduler that is collocated with the edge server Komet platform in future work, as we discuss in §9.

5 Scheduling Heuristics

The serverless abstractions in Komet provide flexibility in service migration and can be used with a variety of scheduling approaches. Finding an optimal service migration schedule depends on many factors beyond service access latency and is thus not the aim of this paper (we discuss some avenues for further research on this in §9). Rather than finding an optimal schedule, we propose simple heuristics for service scheduling that serve as a starting point for LEO edge service scheduling with Komet. Similarly to the *Sticky* approach proposed by Bhattacherjee et al. [10], we are interested in trading some service latency (being only within, say, 10% of the optimum) for fewer service migrations. While service migrations in Komet are seamless, replicating data across the network can still be costly in terms of bandwidth and should be avoided where possible.

We propose heuristics for three scenarios: a single client accessing a single service instance (*one-to-one*), multiple clients sharing a single service instance (*many-to-one*), and multiple clients sharing multiple service instances (*many-to-many*).

5.1 One-to-One

For a single client, we can proactively migrate the LEO edge service instance by calculating the trajectory of the satellite network, which is not compute-intensive [24, 36]. This allows us to determine the network distance between the client and all available satellite servers in the future. Our heuristic first simply selects the closest satellite but afterwards only switches servers if a new server is at least 10% better in terms of network distance compared to the currently selected satellite. The 10% threshold is adapted from

the existing *Sticky* approach as an arbitrary point along the trade-off between service latency and migration frequency, and we measure the impact of this threshold (compared to others) in §8. Further, we also measure service migration delay (time to deploy the necessary functions and data replica to the new satellite server) and initiate this deployment with sufficient lead-time, ensuring that the service is ready when the hand-off needs to be performed.

5.2 Many-to-One

For many clients sharing a single server, we can extend our heuristic to take multiple network distances into account by aggregating them. We use the root-mean-square for this aggregation as it is more sensitive to outliers than the mean, ensuring that all clients have comparable service access latency. This can similarly be calculated ahead-of-time, and we again only initiate hand-offs if the score for a new satellite is at least 10% better than the currently selected one.

5.3 Many-to-Many

FaaS functions backed by replicated data in Komet also opens up the possibility of a many-to-many deployment: Multiple clients share multiple instances of the service that are all replicated. Each client can access a service copy at a satellite near itself, while the underlying state is replicated, essentially providing identical services at different locations. The difference to the many-to-one deployment is the trade-off between service latency and data staleness.

Selecting multiple servers requires a more advanced heuristic as there is now the additional cost of each additional replica (in terms of, e.g., bandwidth for replication). We follow a similar approach using pre-calculated satellite trajectories. For each client, we first calculate the set of satellite servers that are within 10% network distance of the closest satellite to that client. We then calculate a *hitting set*, i.e., the smallest set of satellites that contains at least one of each client's closest satellites. While this is theoretically NP-hard, note that there (i) exist sufficient approximations [21], (ii) the number of satellites is limited, and (iii) this calculation is not on the critical path in live scheduling, as trajectories can be pre-calculated. The number of satellites required to host the service may thus also change over time. For each newly selected satellite, we select the closest (in terms of network distance) satellite currently running the service as a replication source. This means that multiple satellites could receive data from the same source satellite and that some satellites may not migrate their local state further. Further, note that this heuristic only optimizes for service latency and does not take into account the resource requirements for serving multiple clients. We consider this out of scope for this work but discuss avenues for future research on scheduling in §9.

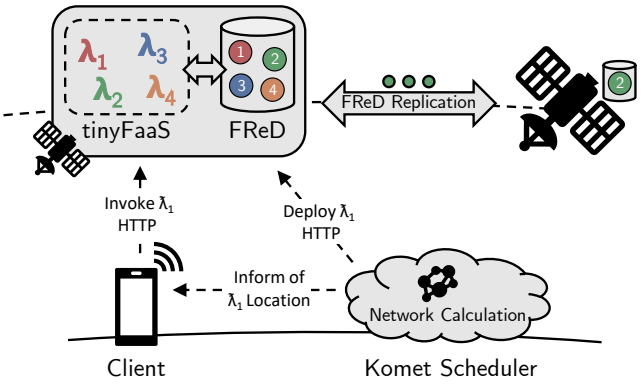


Figure 5: Our Komet prototype combines the tinyFaaS lightweight edge FaaS platform and FReD edge data management middleware with a central scheduling component. Functions are backed by replicated data and can be invoked by clients through an HTTP endpoint.

6 Prototype Implementation

To evaluate Komet, we implement a proof-of-concept prototype. We combine existing research prototypes in the field of serverless edge computing with a novel LEO-edge-focused scheduling component that handles proactive replication. Note that the goal of our prototype is not to provide a full-fledged software system with the best possible performance but rather to illustrate and evaluate the Komet concept. We show an overview of our implementation in Figure 5.

We use the *tinyFaaS* [43] lightweight FaaS platform that can run Python functions on a single Linux host. tinyFaaS exposes an HTTP endpoint to invoke functions and a separate HTTP endpoint to upload and remove functions. Function handlers are isolated using Docker containers. While this does not necessarily meet the security requirements of all services, we consider the issue of efficient isolation for multi-tenant FaaS systems orthogonal to our work. We select tinyFaaS specifically for its low overhead and resource footprint, but the Komet architecture could also be implemented with other FaaS platforms such as *OpenFaaS* [19], *Knative* [59], or *nuclio* [29].

Further, we rely on the *FReD* [49] data management platform for geo-distributed edge-to-cloud environments. FReD manages multiple independent data pools, called *keygroups*, across a distributed set of servers, called *nodes*. Specifically, clients can dynamically specify to which nodes a keygroup should be replicated, and FReD will ensure data replication for that keygroup with client-centric consistency guarantees. While FReD only supports key-value data with a simple CRUD interface, this is sufficient for our proof-of-concept prototype. Again, Komet could also be implemented with

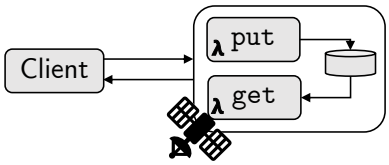


Figure 6: Single-client cache service: The application has a `get` and a `put` function that read and write data into the cache, respectively. The client alternates between calling these two functions every second.

alternative data management tools that offer application-controlled replica placement [25, 26].

The prototype of our Komet scheduler is implemented in Python. Clients connect to this scheduler using a WebSocket and receive new server locations as soon as a service is migrated.

7 Demonstration & Experimental Evaluation

We evaluate Komet using our prototype implementation on the Celestial LEO edge emulator [45]. Celestial emulates LEO satellite and ground station servers using Firecracker microVMs [2] and adjusts network parameters such as latency and bandwidth according to a simulated LEO constellation. We deploy three different applications on our Komet prototype: a simple read-write cache for a single client, evaluating the feasibility of Komet (§7.1); an IoT application that implements a shared service for multiple clients (§7.2); and a content delivery network demonstrating a multi-user application with a shared, distributed cache (§7.3). As our intention with the Komet prototype is that it serves as a proof-of-concept of our approach, we focus on latency and migration performance in this evaluation, showing that Komet can maintain a consistent service level for different kinds of applications over time, despite the dynamic satellite network topology.

7.1 Single-Client Cache

Our simple read-write cache follows the structure of our motivating example in §3: We deploy our cache as two functions, as shown in Figure 6. The `put` function writes data into the backend data store, while the `get` function returns data for a key. A single client in Redmond, WA, USA, accesses this cache, reading and writing a single data item every second. We again use a single plane from the phase I Starlink constellation with 22 satellites evenly spaced at 550km altitude. This implementation uses our one-to-one scheduling heuristic, and preliminary measurements have revealed a 20-second migration time (data replication and function instantiation) that our scheduler takes into account

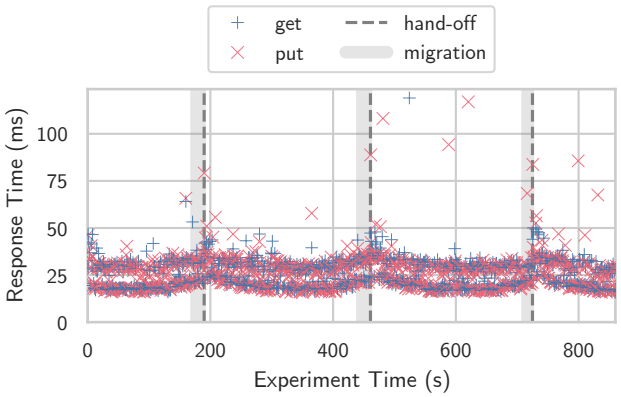


Figure 7: Request-response latency for each request during the single-client experiment along with migration and hand-off times. Results show a consistent client service level that follows orbit patterns, as the client's closest satellite passes over the client location.

for proactive migration. Our functions and scheduler are implemented in Python, while the client is a static Go binary. Our ground station and each satellite server again have 4 vCPUs and 8GB of memory, with 10Gbps bandwidth for ISLs and ground-to-satellite links, and we use the same 24-core host server. Our experiment runs for a total of 15 minutes.

We show the request-response latency for each client request along with migration periods and hand-off events in Figure 7. As the service is deployed on the client's nearest satellite, request-response latency follows the familiar pattern of shrinking and growing as the satellite passes over the client (except for outliers, where the service is slow to respond). The Komet orchestrator proactively initiates service migration before the client is handed off, as shown in the figure: During these migration periods, the service is replicated across the client's nearest satellite and the anticipated *soon-to-be nearest* satellite. In our experiments, this replication occurs 20.8s before the hand-off, on average. In other words, the service is replicated for ~2.4% of our experiment time. As soon as the hand-off is completed, the original replica is destroyed and only a single service replica on the client's then nearest satellite server remains. As our results show, this hand-off technique leads to consistent request-response times without downtime.

Naturally, the replication time in Komet also depends on the size of data in the data store. To directly compare the performance of Komet with our container replication experiment in §3, we measure data replication times in our setup using between zero and 1,000 (in increments of 100) 1MB items. The results in Figure 8 show a similar, expected linear correlation between data size to replicate and replication

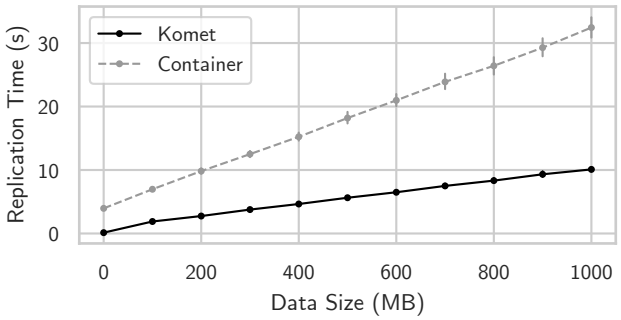


Figure 8: Time to replicate data to a new satellite server in our Komet implementation compared to total time for container migration (single client)

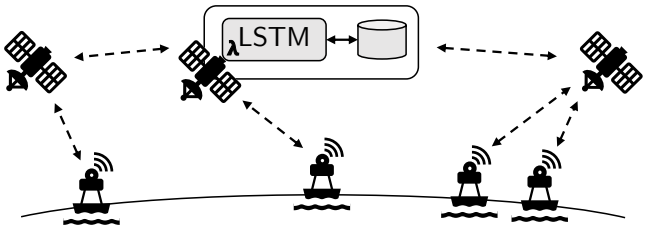


Figure 9: Our IoT application connects 30 DART seismic monitoring buoys with a shared LSTM service running on a satellite server. Current values for each sensor are stored in the backing database and ML inference is performed over all values.

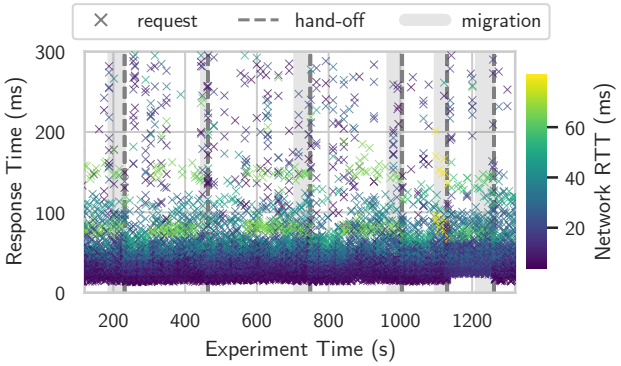


Figure 10: Response latency for each function invocation along with migration and hand-off times in the IoT experiment. Colors indicate calculated network round-trip times. Our heuristic leads to seven hand-offs during our 20-minute experiment. While there is some variance in the response latency, this shows how service response times remain consistent regardless of satellite movement.

times. In Komet, however, this replication time does not lead to service downtime. Further, the small minimum replication time of only 131ms to replicate the dataset with 0MB (compare 3.74s with containers) demonstrates the benefits of replicating only decoupled application state rather than the entire execution stack.

7.2 Internet of Things

A more complex application of LEO edge computing is supporting remote IoT devices. The *National Oceanic and Atmospheric* (NOAA) *Deep-ocean Assessment and Reporting of Tsunami* (DART) [22, 41] ocean buoys are located throughout the Pacific Ocean and collect seismic measurements to detect tsunamis early. As these buoys are far from any terrestrial network connection, they already use satellite networks to send data to a central monitoring station. We have previously proposed equipping the satellite network that supports the DART buoys with compute resources to aggregate sensor data and perform ML inference to detect tsunami risks with low latency [45]. Specifically, they have implemented a sample *Long Short-Term Memory* (LSTM) ML model that runs on satellite servers.

We use this existing implementation as a starting point for a serverless remote IoT application based on Komet. As shown in Figure 9, our application runs the existing LSTM model (implemented in TensorFlow) as a serverless function. Remote buoys send their measurements to this function, which stores the latest value for each buoy in the backend data store. Using these values, the LSTM model generates a risk factor and sends it as a response. Multiple clients share the same service instance.

We use the locations of all 30 DART buoys in the North Pacific Ocean for our clients and the phase I Starlink constellation as our LEO network. Buoys send a sensor measurement every second. Our Komet scheduler is located on Fort Island, HI, USA (the location of the Pacific Tsunami Warning Center), although we do not expect this to impact our results. This implementation uses the many-to-one scheduling heuristic. Our functions and scheduler are again implemented in Python, while the client is a static Go binary. All ground stations and satellite servers have 8 vCPUs and 8GB of memory, with 10Gbps bandwidth for ISLs and ground-to-satellite links. Given the resource requirements, we perform this experiment on eight `n2-standard-16` Google Compute Engine VM instances with 16 vCPUs and 64GB of memory in the `europe-west3` (Frankfurt, Germany) region. Our experiment runs for a total of 20 minutes plus two minutes of ramp-up time.

We show the response times for client requests in Figure 10. Additionally, we also show the expected network round-trip

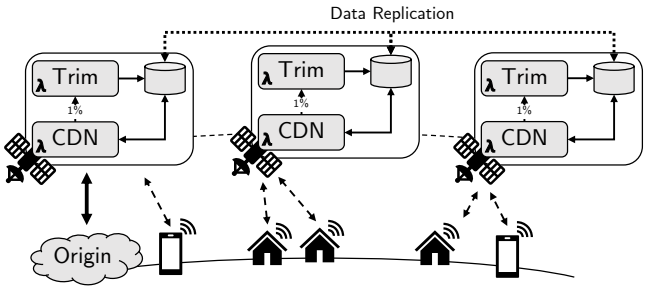


Figure 11: In the LEO edge CDN based on Komet, clients request data items from the CDN function deployed on their closest satellite. Data items are served from the replicated cache if available or first pulled from the origin location. A trimming function compacts the replicated cache periodically (randomly with a 1% chance).

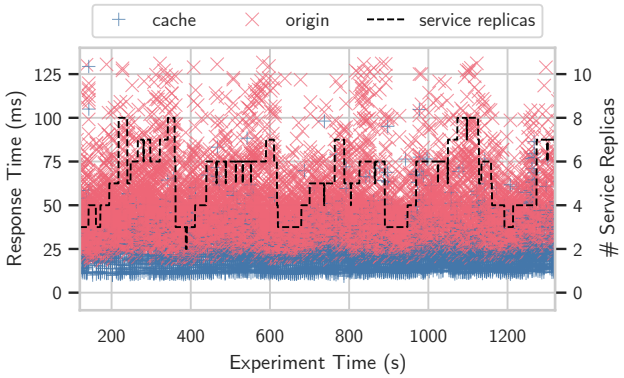


Figure 12: Response times for client requests in our CDN experiment. As expected, responses served from the CDN cache are faster than those pulled from the origin location. We also show the number of service replicas deployed by our scheduler over time.

time from each client to the satellite server currently running the LSTM service. While there are some outliers, we can see the correlation between request-response latency and network latency, i.e., response times are higher if the client is further away from the selected server. Over the course of the 20-minute experiment, Komet migrates the service seven times. The measured response times show that this keeps the service at a consistent service level for clients.

7.3 Content Delivery Network

A benefit of attaching replicated data storage to our LEO edge functions is that beyond sharing a single satellite server, clients can also share a LEO edge service that is distributed across multiple servers. Consider content delivery networks (CDN), a further possible LEO edge application [10, 12]: Clients use a nearby CDN caching server to request web pages or data, such as images or videos. If the cache has a copy of the requested item, it is served to the client from this nearby cache. If the cache does not have a copy, the item is first pulled from an origin location, then cached locally, and finally sent to the client. While this has benefits for a single client, the benefit is even greater for multiple clients sharing a regional CDN caching layer, as clients in geographic proximity can exhibit similar request behavior.

We can implement such a CDN cache on top of Komet as shown in Figure 11. Multiple copies of the CDN function are distributed across satellites over a wider geographic area. The CDN functions can use their local data store replicas to read and write data. If a client accesses a file that is not in the store, any service replica can pull it from the origin location and store it. Periodically (randomly with a 1% chance in our implementation), a separate trimming function is called asynchronously to trim the cache to the most-recently used 1,000 items.

We use 50 client locations in the Northwest of the United States and an origin location in Umatilla County, OR, USA (site of an AWS data center in that region [4]). Our data set is based on image request traces from the *Wikimedia Mediacounts* data set [3], from which clients request a data item every five seconds. The scheduler in this experiment uses the many-to-many scheduling heuristic. Our functions and scheduler are again implemented in Python, while the client is a static Go binary. All ground stations and satellite servers have 8 vCPUs and 8GB of memory, with 10Gbps bandwidth for ISLs and ground-to-satellite links. We perform this experiment on 16 `n2-standard-16` Google Compute Engine VM instances with 16 vCPUs and 64GB of memory in the `europe-west3` (Frankfurt, Germany) region. Our experiment runs for a total of 20 minutes plus two minutes of ramp-up time.

We show the response times for client requests in Figure 12. Unsurprisingly, responses served from the CDN cache are served with lower latency than those that first have to be pulled from the origin location. On average, cached responses take 21.58ms while non-cached responses take 46.44ms. In this workload, 34.8% of requests can be served from cache, although only 1,000 (of a possible 542,112 in the dataset) are kept in the cache.

Interesting to observe is also the number of service replicas deployed over the course of the experiment: Our scheduler deploys a maximum of eight replicas and a minimum of two replicas to serve all 50 clients, with a mean 5.3 replicas deployed. This demonstrates an interesting dynamic in satellite networks, where a highly varying number of satellites is required to provide coverage for the same, static clients over time.

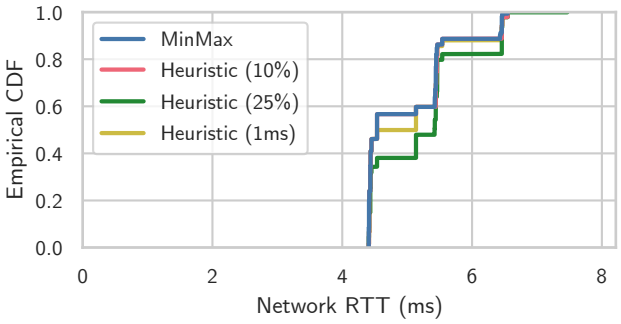


Figure 13: Network RTT between the single client and the satellite server the scheduling strategies select for that client. All strategies perform similarly, with a minimum mean 4.99ms RTT for MinMax and a maximum mean 5.22ms RTT for the Komet heuristic with a 25% threshold.

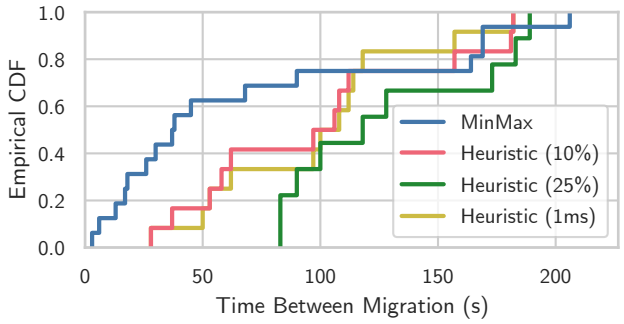


Figure 14: Time between service migrations differ in our single-client scenario, with a mean 68.7s duration for MinMax and 127.4s for the 25% heuristic.

8 Scheduling Simulations

We evaluate the effectiveness of our scheduling heuristics in simulations based on traces generated in our three experiments. Here, we focus on evaluating the proposed heuristics in an isolated manner, without the additional effects of service migration, service performance, resource variability, and especially without being influenced by our particular Komet implementation.

8.1 Single-Client (One-to-One)

Using Celestial, we generate traces for the single-client cache for a client in Redmond, WA, USA, using the full first shell of the phase I Starlink, which comprises 1,584 satellites at 550km altitude. These traces contain the network state at every second for a total duration of 20 minutes. Using these traces, we simulate the behavior of different server selection strategies in Komet, noting the distance between the client and the selected satellite server as well as the duration between service hand-offs. We compare the MinMax approach [10] and our one-to-one heuristic with different parameters: the default 10% threshold for service migration, a 25% threshold, and a threshold of 1ms, where a new satellite is only selected if it can provide service latency at least 1ms lower than the currently selected node.

The network distance measurements for this simple experiment in Figure 13 show similar results for all four heuristics, with a mean 4.98ms RTT for the MinMax strategy, 4.99ms RTT for our 10% heuristic, 5.22ms for the 25% heuristic, and 5.04ms for the 1ms heuristic. Despite these nearly identical results, the duration between hand-offs shown in Figure 14 show the different behaviors of our heuristics. During the 20-minute trace, the MinMax strategy performs 16 service migrations, whereas the 10% and 1ms heuristics perform twelve and the 25% heuristic performs only nine. On average, a service in the MinMax is active for only 68.7s, 98.4s in the 10% and 1ms heuristic, and 127.4s in the 25% heuristic. This illustrates how our heuristic trades a marginal increase in RTT for significantly fewer of the costly service migrations.

8.2 IoT (Many-to-One)

To further investigate the trade-off between service distance increase and migration count, we use traces generated from the IoT experiment with 30 clients in the North Pacific. Here, we again compare the MinMax strategy [10] (the lowest sum of client-satellite distance at every instance) to our heuristic with 10% (default) and 25% thresholds. Additionally, we also simulate the Sticky strategy [10]. Note that the Sticky strategy was intended only for a many-to-one scenario, and porting it to other evaluation scenarios would give limited comparability to existing work.

The distance measurements of our IoT experiments for the trace period of 20 minutes (Figure 15) again show how similar the MinMax and heuristic strategies perform. The mean RTT is 19.31ms for MinMax, 19.78ms for the 10% heuristic, and 19.68ms for Sticky. Only the 25% heuristic has a slightly higher mean RTT of 21.34ms. More interesting here is the time between service migrations, shown in Figure 16. During the 20-minute trace, the 10% heuristic performs five service migrations, while the 25% heuristic performs only three. Compare this to MinMax, which requires a total of 113 service migrations for marginal benefits in service distance. As a result, service instances in MinMax are active for only 10.33s on average before being migrated to the next satellite, compared to 195s (10% heuristic) and 386s (25% heuristic). The Sticky heuristic performs similarly, requiring four migrations with a mean 266.75s between migrations.

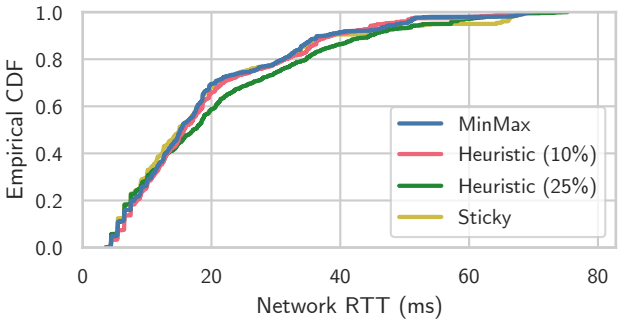


Figure 15: RTT measurements between clients and the service in our IoT scenario are similar regardless of the chosen strategy, with a mean 19.31ms, 19.78ms, 21.34ms, and 19.68ms RTT for MinMax, 10% heuristic, 25% heuristic, and Sticky, respectively.

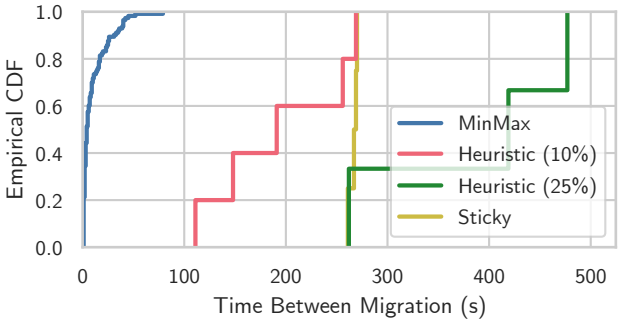


Figure 16: The server selection strategies differ in the amount of service migrations during the 20-minute trace, with MinMax performing 113 migrations in the IoT experiment (mean 10.33s between migrations) and our heuristics with 10% and 25% thresholds performing 5 and 3 migrations (mean 195s and 386s between migrations), respectively. Sticky performs 4 migrations with a mean 266.75s between migrations.

To further investigate this trade-off, we extend our trace generation to one hour for the IoT scenario. We now simulate different thresholds for our heuristic, between 0% and 50% in five percentage point increments. We show the achieved 99th percentile RTT and the number of migrations during our one-hour trace in Figure 17. As expected, the MinMax and 0% heuristic strategies perform the most migrations, at 348 over the course of one hour. Note that the MinMax and 0% heuristic only differ in terms of aggregation, with MinMax using the average distance from clients to servers and our heuristic using the root-mean-square to score potential satellites. The result is a lower 99th percentile latency as the root-mean-square is more fair to outliers (mean RTT is

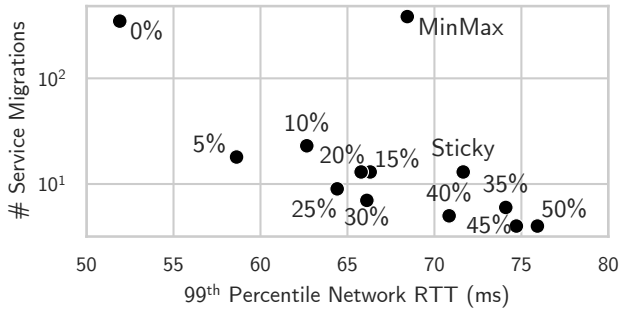


Figure 17: Increasing the threshold in our scheduling heuristic decreases the number of migrations while increasing the 99th percentile latency, clearly showing a Pareto front in the trade-off between the number of migrations and the possible RTT.

slightly higher at 19.75ms compared to 19.33ms). Increasing the threshold in our scheduling heuristic decreases the number of migrations while increasing the 99th percentile latency, clearly showing a Pareto front in the trade-off between the number of migrations and the possible RTT. The results also show that our heuristic is not optimal in all cases, with the 5% threshold leading to less hand-offs (18 compared to 23) and 99th percentile RTT (58.62ms compared to 62.66ms) than the 10% threshold. A possible reason is that the 5% heuristic makes a satellite selection that turns out to be more stable than the other heuristic as a result of the dynamics of the satellite network.

8.3 CDN (Many-to-Many)

We simulate our many-to-many heuristic using traces from our CDN example with 50 clients in Northwest USA. We again compare the default 10% heuristic from Komet with a 25% threshold, 1ms threshold (select minimum set of servers to cover all clients within 1ms of their optimum distance).

As shown in Figure 18, the distance measurements from our clients to their nearest service instance follow a trend that is similar to our single-client simulation: MinMax and our 10% heuristic achieve a comparable mean 4.36ms and 4.39ms RTT, respectively, with a slightly higher 4.73ms and 4.79ms for the 25% and 1ms heuristics. For this scenario, calculating the time between migrations is not possible, as there are multiple concurrent service instances. Instead, we are interested in the number of such service instances, which we show in Figure 19. Despite the highly similar RTT the different strategies achieve, they require different numbers of service replicas. The MinMax chooses up to 15 service replicas (mean 9.84) throughout the 20-minute trace, while the 10% heuristic requires only a maximum 10 (mean 4.36). The more relaxed 25% and 1ms thresholds again lead to

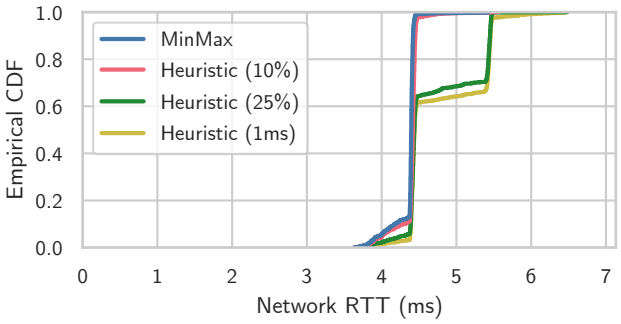


Figure 18: The network distances between clients and their selected satellite servers in the CDN use-case are similar between server selection strategies, between mean 4.36ms for MinMax and 4.79ms for the 1ms heuristic.

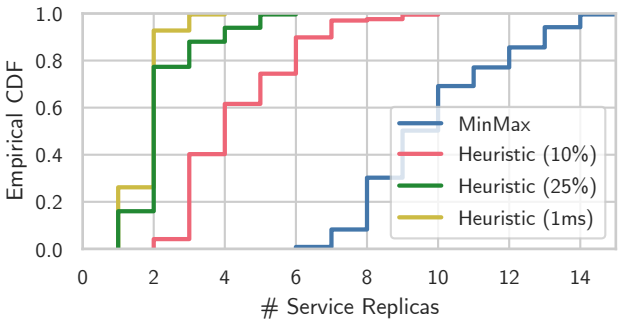


Figure 19: The CDF shows the number of concurrent service replicas each strategy needs in every second of the simulation, a lower number is better (less replication necessary). The strategies require different numbers of service replicas, with MinMax requiring a mean 9.84 and the 1ms heuristic requiring only 1.82.

less service replicas, with a maximum six (mean 2.25) and four (mean 1.82), respectively. This demonstrates a similar trade-off to the one between service distance and migration frequency also in scenarios with multiple service replicas, i.e., this also affects utilization of compute resources in the satellite constellation.

9 Discussion & Future Work

We have introduced Komet, a serverless framework for LEO edge services that handles service migration seamlessly. This section investigates limitations of our current architecture and derives opportunities for future work.

Limitations of the FaaS Programming Model. In our evaluation, we have shown how the FaaS programming model we use in Komet can be employed to build a variety of LEO edge applications. Beyond that, FaaS has been demonstrated to be a particularly good fit for edge applications in general [5, 52, 55, 63], especially with the addition of a data management backend [18, 23, 47, 58]. Nevertheless, the FaaS programming model has limitations: For example, it cannot easily support applications with continuous inputs or outputs, such as live audio transcoding or video conferencing. Regardless of the chosen programming model, such applications will be particularly difficult to deploy on the LEO edge, given that service migration would have to occur seamlessly. As these applications would benefit from low-latency edge deployment, we plan on investigating how they may be migrated efficiently in future work, possibly by integrating a continuous stream abstraction in a serverless platform.

Limitations of Replicated Data. As in any replicated system, Komet is subject to data consistency tradeoffs [1] and end users inherit the consistency properties of the datastore used.

For applications with only single-writer access (either a single FaaS function instance or only one end user): With the snapshot migration approach from §3, end users will see consistent data for a non-replicated datastore but will encounter a period of downtime during the migration. If such a datastore is replicated, end users will experience the consistency guarantees offered by that datastore, possibly enhanced with client-centric consistency guarantees [7]. In our approach, FReD [49] already provides such client-centric guarantees by exposing vector clocks-based versioning and having the FReD client library request the appropriate versions from the datastore backend via the function instance used. For the period of data migration, this means that our approach does not encounter downtime but will instead have small latency spikes when requests to data items that are not yet up-to-date on the new satellite are transparently served from the old replica instead. We believe that this is preferable to downtime, especially considering that these not-too-high (it is a nearby satellite after all) latency spikes only occur when clients access data that has been written right before switching satellites. All other data will already be up-to-date locally.

For all multi-writer scenarios: The snapshot approach will have the downtime problem outlined above, in case of synchronous replication multiplied by the number of datastore replicas. Furthermore, end users will again inherit the consistency and latency properties of the replication strategy used by the respective datastore – from synchronous primary copy to asynchronous update everywhere. In our approach, end users may not see updates from other end users instantly due to the asynchronous replication strategy used in FReD, i.e., they will encounter staleness, but client-centric consistency guarantees are provided. Furthermore, there is no downtime

and the mentioned latency spikes will happen at different points in time to different clients worldwide (whenever a migration happens), possibly with lower latency spikes as another replica might be closer than the one that is currently being migrated.

Satellite Server Selection. We have shown in simulations that the scheduling heuristics we propose efficiently allocate satellite servers for services along the trade-off between service latency and migration frequency. Nevertheless, the challenge of finding an optimal schedule requires far more research attention in the future. Specifically, we plan to further investigate the costs of service latency and service migration. Furthermore, there are additional factors that must be taken into account for service scheduling, such as satellite server resources, energy demand, or server temperature [13, 38]. The fluctuating demand for services, e.g., by bursty traffic, complicates scheduling further, requiring dynamic heuristics or an appropriate ever-changing solution to a formal assignment problem as services claim or release resources. While fully solving distributed LEO edge scheduling is beyond the scope of this paper, we believe Komet makes meaningful progress toward this goal: Its serverless architecture gives the platform fine-grained control over resource allocation at a per-request level, allowing it to adhere to any scheduling strategy. The zero-downtime migration feature is especially valuable, enabling the platform to move services to nearby satellites with available resources.

Scheduler Location. Komet includes a centralized, per-application scheduler, which we believe is sufficient for our use-case, as scheduling decisions are not latency-critical given that they are not on the critical path of client requests. More advanced distributed and fault-tolerant approaches are feasible yet outside the scope of this paper. An interesting avenue for future research is the investigation of other scheduler architectures, such as a distributed, on-satellite scheduling component where each satellite server can make autonomous decisions, or a centralized scheduler that combines scheduling for all services. For example, despite the possibility of per-application sharding (where each application is managed by a separate scheduler), scalability could become a challenge when handling many clients and potential satellite servers, particularly in many-to-one or many-to-many scenarios. The main issue is distributing the heuristic calculations in a way that ensures they are completed within the required timeframes. However, given that systems such as Starlink successfully use a similar centralized scheduling model, we are confident that such calculations are feasible. Similarly, when considering fault tolerance, because scheduling occurs asynchronously and is not on a critical path, a scheduler failure would have minimal impact on LEO edge services. A simple failover mechanism, such as using a shadow scheduler, would quickly restore normal operation.

Failure Tolerance. LEO edge computing can be subject to on-board server failure due to radiation [10, 46]. Although such failure is unlikely, the right failover mechanisms must be in place to provide continuous service coverage. While this requires further research, we believe that the abstractions in Komet are suited for transparent failover. Using data replication, a LEO edge service could keep a back-up secondary data replica on a nearby satellite at all times. In case of satellite failure, the client could then be handed off to this secondary replica seamlessly, similarly to a normal hand-off. While this comes with additional scheduling challenges and communication costs for additional replicas, it is only possible when services can be replicated easily by the underlying platform, as is the case in Komet.

10 Conclusion

This paper introduces Komet, a serverless platform for LEO edge computing. Komet integrates FaaS compute abstractions and data replication to enable transparent service migration against satellite orbital movement to keep services deployed close to the clients that access them. We implement a proof-of-concept prototype of Komet that we evaluate in an emulated testbed, demonstrating how the abstractions of Komet can be used to build three different example LEO edge applications. Our evaluation shows that Komet can provide continuous service availability with high service levels despite satellite movement. We also propose simple heuristics for service migration on the LEO edge that we evaluate based on our experiment traces. Our simulation results show the trade-off between optimum service network distance and migration frequency and provide a starting point for future research on LEO edge service scheduling.

Acknowledgments

We thank our anonymous reviewers and our shepherd Jon Crowcroft for their insightful feedback that helped shape this paper. We also thank Stanislav Kosorin for his support in evaluating container migration. This work is supported by the Bundesministerium für Bildung und Forschung (BMBF, German Federal Ministry of Education and Research) – 16KISK183 and 01IS23068.

References

[1] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE Computer* 45, 2 (Jan. 2012), 37–42. https://doi.org/10.1109/MC.2012.33

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In

Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (Santa Clara, CA, USA) *(NSDI '20)*. USENIX Association, Berkeley, CA, USA, 419–434.

[3] Christian Aistleitner, Federico Leva, Paladox, James Hare, Dan Andreescu, Tilman Bayer, Nuria Ruiz, Francisco Dans, Morten Warncke-Wang, Tricia Burmeister, and Alex Paskulin. 2024. *Data Platform/Data Lake/Traffic/Mediacounts*. Wikimedia Foundation. Retrieved July 8, 2024 from https://wikitech.wikimedia.org/wiki/Data_Platform/Data_Lake/Traffic/Mediacounts

[4] Amazon Staff. 2023. *In eastern Oregon, Amazon is working with a local utility to power AWS data centers with clean energy*. Amazon Web Services. Retrieved July 8, 2024 from https://www.aboutamazon.com/news/aws/data-center-oregon-renewable-energy

[5] Mohammad S. Aslanpour, Adel N. Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. 2021. Serverless Edge Computing: Vision and Challenges. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference* (Dunedin, New Zealand) *(ACSW '21)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3437378.3444367

[6] Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott. 2023. Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing. In *Proceedings of the 2023 USENIX Annual Technical Conference* (Boston, MA, USA) *(ATC '23)*. USENIX Association, Berkeley, CA, USA, 215–231.

[7] David Bermbach, Jörn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. 2013. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *Proceedings of the 1st IEEE International Conference on Cloud Engineering* (San Francisco, CA, USA) *(IC2E 2013)*. IEEE, New York, NY, USA, 114–123. https://doi.org/10.1109/IC2E.2013.32

[8] David Bermbach, Frank Pallas, David García Pérez, Pierluigi Plebani, Maya Anderson, Ronen Kat, and Stefan Tai. 2017. A Research Perspective on Fog Computing. In *Proceedings of the 2nd Workshop on IoT Systems Provisioning & Management for Context-Aware Smart Cities* (Malaga, Spain) *(ISYCC 2017)*. Springer, Cham, Switzerland, 198–210. https://doi.org/10.1007/978-3-319-91764-1_16

[9] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, Brighten P. Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. 2018. Gearing up for the 21st Century Space Race. In *Proceedings of the 17th ACM Workshop Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3286062.3286079

[10] Debopam Bhattacherjee, Simon Kassing, Melissa Licciardello, and Ankit Singla. 2020. In-orbit Computing: An Outlandish thought Experiment?. In *Proceedings of the 19th ACM Workshop Hot Topics in Networks* (Virtual Event, USA) *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 197–204. https://doi.org/10.1145/3422604.3425937

[11] Debopam Bhattacherjee and Ankit Singla. 2019. Network Topology Design at 27,000 km/hour. In *Proceedings of the 15th International Conference on Emerging Network Experiments And Technologies* (Orlando, FL, USA) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/3359989.3365407

[12] Vaibhav Bhosale, Ketan Bhardwaj, and Ada Gavrilovska. 2020. Toward Loosely Coupled Orchestration for the LEO Satellite Edge. In *Proceedings of the 3rd USENIX Workshop Hot Topics in Edge Computing (HotEdge '20)*. USENIX Association, Berkeley, CA, USA.

[13] Vaibhav Bhosale, Ketan Bhardwaj, and Ada Gavrilovska. 2023. Don't Let Your LEO Edge Fade at Night. In *Proceedings of the Workshop on Hot Topics in System Infrastructure* (Orlando, FL, USA) *(HotInfra '23)*. 1–6.

[14] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (Helsinki, Finland) *(MCC '12)*. Association for Computing Machinery, New York, NY, USA, 13–16. https://doi.org/10.1145/2342509.2342513

[15] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. 2019. How to place your apps in the fog: State of the art and open challenges. *Software: Practice and Experience* 50, 5 (Nov. 2019), 719–740. https://doi.org/10.1002/spe.2766

[16] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proceedings of the ACM on Programming Languages* 5, Article 133 (Oct. 2021), 27 pages. https://doi.org/10.1145/3485510

[17] Pietro Cassará, Alberto Gotta, Mario Marchese, and Fabio Patrone. 2022. Orbital Edge Offloading on Mega-LEO Satellite Constellations for Equal Access to Computing. *IEEE Communications Magazine* 60, 4 (April 2022), 32–36. https://doi.org/10.1109/MCOM.001.2100818

[18] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. 2019. Fog Function: Serverless Fog Computing for Data Intensive IoT Services. In *Proceedings of the 2019 IEEE International Conference on Services Computing* (Milan, Italy) *(SCC '19)*. IEEE, New York, NY, USA, 28–35. https://doi.org/10.1109/SCC.2019.00018

[19] Alex Ellis. 2024. *OpenFaaS – Serverless Functions, Made Simple*. OpenFaaS Ltd. Retrieved July 2, 2024 from https://www.openfaas.com/

[20] Pavel Emelyanov, Sergey Bronnikov, Kir Kolyshkin, Dmitry Safonov, Radostin Stoyanov, MettaCrawler, Andrei Vagin, Cyrill Gorcunov, and Wenhui Zhang. 2023. *CRIU*. OpenVZ. Retrieved July 9, 2024 from https://criu.org/Main_Page

[21] Uriel Feige. 1998. A threshold of ln n for approximating set cover. *J. ACM* 45, 4 (July 1998), 634–652. https://doi.org/10.1145/285055.285059

[22] Frank I. Gonzalez, Hank M. Milburn, Eddie N. Bernard, and Jean C. Newman. 1998. Deep-Ocean Assessment and Reporting of Tsunamis (DART®): Brief Overview and Status Report. In *Proceedings of the International Workshop on Tsunami Disaster Mitigation* (Tokyo, Japan). 19–22.

[23] Daniel Habenicht, Kevin Kreutz, Soeren Becker, Jonathan Bader, Lauritz Thamsen, and Odej Kao. 2022. SyncMesh: improving data locality for function-as-a-service in meshed edge networks. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking* (Rennes, France) *(EdgeSys '22)*. Association for Computing Machinery, New York, NY, USA, 55–60. https://doi.org/10.1145/3517206.3526275

[24] Mark Handley. 2018. Delay is Not an Option: Low Latency Routing in Space. In *Proceedings of the 17th ACM Workshop Hot Topics in Networks (HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 85–91. https://doi.org/10.1145/3286062.3286075

[25] Jonathan Hasenburg, Martin Grambow, and David Bermbach. 2019. *FBase: A Replication Service for Data-Intensive Fog Applications*. Technical Report MCC.2019.1. Technische Universität Berlin & ECDF, Mobile Cloud Computing Research Group, Berlin, Germany.

[26] Jonathan Hasenburg, Martin Grambow, and David Bermbach. 2020. Towards A Replication Service for Data-Intensive Fog Applications. In *Proceedings of the 35th ACM Symposium on Applied Computing, Posters Track* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 267–270. https://doi.org/10.1145/3341105.3374060

[27] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*. USENIX Association, Berkeley, CA, USA, 33–39.

[28] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. 2021. μActor: Stateful Serverless at the Edge. In *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications* (Virtual, WI, USA) *(MobileServerless'21)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3469263.3470828

[29] Iguazio Ltd. 2024. *nuclio: Serverless Platform for Automated Data Science*. Retrieved July 2, 2024 from https://nuclio.io/

[30] Liz Izhikevich, Manda Tran, Katherine Izhikevich, Gautam Akiwate, and Zakir Durumeric. 2024. Democratizing LEO Satellite Network Measurement. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 8, 1, Article 13 (Feb. 2024), 26 pages. https://doi.org/10.1145/3639039

[31] Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3434770.3459730

[32] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Berkeley, CA, USA. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[33] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. 2022. Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes. In *Proceedings of the 2022 IEEE 6th International Conference on Fog and Edge Computing* (Taormina, Messina, Italy) *(ICFEC '22)*. IEEE, New York, NY, USA, 26–33. https://doi.org/10.1109/ICFEC54809.2022.00011

[34] Pekka Karhula, Jan Janak, and Henning Schulzrinne. 2019. Checkpointing and Migration of IoT Edge Functions. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking* (Dresden, Germany) *(EdgeSys '19)*. Association for Computing Machinery, New York, NY, USA, 60–65. https://doi.org/10.1145/3301418.3313947

[35] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. 2020. Exploring the "Internet from Space" with Hypatia. In *Proceedings of the ACM Internet Measurement Conference (IMC '20)*. Association for Computing Machinery, New York, NY, USA, 214–229. https://doi.org/10.1145/3419394.3423635

[36] Benjamin Kempton and Anton Riedl. 2021. Network Simulator for Large Low Earth Orbit Satellite Networks. In *Proceedings of the 2021 IEEE International Conference on Communications (ICC '21)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/ICC42927.2021.9500439

[37] Zeqi Lai, Hewu Li, Yangtao Deng, Qian Wu, Jun Liu, Yuanjie Li, Jihao Li, Lixin Liu, Weisen Liu, and Jianping Wu. 2023. StarryNet: Empowering Researchers to Evaluate Futuristic Integrated Space and Terrestrial Networks. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '23)*. USENIX Association, Berkeley, CA, USA, 1309–1324.

[38] Weisen Liu, Zeqi Lai, Qian Wu, Hewu Li, Qi Zhang, Zonglun Li, Yuanjie Li, and Jun Liu. 2024. In-Orbit Processing or Not? Sunlight-Aware Task Scheduling for Energy-Efficient Space Edge Computing Networks. (July 2024). arXiv:2407.07337

[39] Michele Luglio, Mario Marchese, Fabio Patrone, Cesare Roseti, and Francesco Zampognaro. 2022. Performance Evaluation of a Satellite Communication-Based MEC Architecture for IoT Applications. *IEEE Trans. Aerospace Electron. Systems* 58, 5 (Aug. 2022), 3775–3785. https://doi.org/10.1109/TAES.2022.3199330

[40] Lele Ma, Shanhe Yi, and Qun Li. 2017. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (San Jose, California) *(SEC '17)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3132211.3134460

[41] Christian Meinig, Scott E. Stalin, Alex I. Nakamura, and Hugh B. Milburn. 2005. *Real-Time Deep-Ocean Tsunami Measuring, Monitoring, and Reporting System: The NOAA DART II Description and Disclosure*. Technical Report. NOAA, Pacific Marine Environmental Laboratory (PMEL), Honolulu, HI, USA. https://nctr.pmel.noaa.gov/Dart/Pdf/DART_II_Description_6_4_05.pdf

[42] Nitinder Mohan, Andrew E. Ferguson, Hendrik Cech, Rohan Bose, Prakita Rayyan Renatin, Mahesh K. Marina, and Jörg Ott. 2024. A Multifaceted Look at Starlink Performance. In *Proceedings of the ACM on Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 2723–2734. https://doi.org/10.1145/3589334.3645328

[43] Tobias Pfandzelter and David Bermbach. 2020. tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *Proceedings of the Second IEEE International Conference on Fog Computing* (Sydney, NSW, Australia) *(ICFC '20)*. IEEE, New York, NY, USA, 17–24. https://doi.org/10.1109/ICFC49376.2020.00011

[44] Tobias Pfandzelter and David Bermbach. 2021. Edge (of the Earth) Replication: Optimizing Content Delivery in Large LEO Satellite Communication Networks. In *Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing* (Melbourne, Australia) *(CCGrid '21)*. IEEE, New York, NY, USA, 565–575. https://doi.org/10.1109/CCGrid51090.2021.00066

[45] Tobias Pfandzelter and David Bermbach. 2022. Celestial: Virtual Software System Testbeds for the LEO Edge. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (Quebec, QC, Canada) *(Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 69–81. https://doi.org/10.1145/3528535.3531517

[46] Tobias Pfandzelter and David Bermbach. 2023. Edge Computing in Low-Earth Orbit – What Could Possibly Go Wrong?. In *Proceedings of the the 1st ACM Workshop on LEO Networking and Communication 2023* (Madrid, Spain) *(LEO-NET '23)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3614204.3616106

[47] Tobias Pfandzelter and David Bermbach. 2023. Enoki: Stateful Distributed FaaS from Edge to Cloud. In *Proceedings of the 2nd International Workshop on Middleware for the Edge* (Bologna, Italy) *(MiddleWEdge '23)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3630180.3631203

[48] Tobias Pfandzelter, Jonathan Hasenburg, and David Bermbach. 2021. Towards a Computing Platform for the LEO Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 43–48. https://doi.org/10.1145/3434770.3459736

[49] Tobias Pfandzelter, Nils Japke, Trever Schirmer, Jonathan Hasenburg, and David Bermbach. 2023. Managing Data Replication and Distribution in the Fog with FReD. *Software: Practice and Experience* 53, 10 (June 2023), 1958–1981. https://doi.org/10.1002/spe.3237

[50] Podman. 2024. *Podman*. Retrieved July 9, 2024 from https://podman.io/

[51] Imadur Rahman, Sara Modarres Razavi, Olof Liberg, Christian Hoymann, Henning Wiemann, Claes Tidestav, Paul Schliwa-Bertling, Patrik Persson, and Dirk Gerstenberger. 2021. 5G Evolution Toward 5G Advanced: An overview of 3GPP releases 17 and 18. *Ericsson Technology Review* 2021, 14 (Oct. 2021), 2–12. https://doi.org/10.23919/ETR.2021.9904665

[52] Philipp Raith, Stefan Nastic, and Schahram Dustdar. 2023. Serverless Edge Computing – Where We Are and What Lies Ahead. *IEEE Internet Computing* 27, 3 (May 2023), 50–64. https://doi.org/10.1109/MIC.2023.3260939

[53] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. 2021. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems* 114 (Jan. 2021), 259–271. https://doi.org/10.1016/j.future.2020.07.017

[54] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[55] Gabriele Russo, Tiziana Mannucci, Valeria Cardellini, and Francesco Lo Presti. 2023. Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum. In *Proceedings of the 2023 IEEE International Conference on Pervasive Computing and Communications* (Atlanta, GA, USA) *(PerCom '23)*. IEEE, New York, NY, USA, 131–140. https://doi.org/10.1109/PERCOM56429.2023.10099372

[56] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. Institut National de Recherche en Informatique et en Automatique (INRIA), Paris, France. https://hal.inria.fr/inria-00555588

[57] Weisong Shi and Schahram Dustdar. 2016. The Promise of Edge Computing. *Computer* 49, 5 (May 2016), 78–81. https://doi.org/10.1109/MC.2016.145

[58] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. 2022. FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters. In *Proceedings of the 2022 IEEE 6th International Conference on Fog and Edge Computing* (Taormina, Messina, Italy) *(ICFEC '22)*. IEEE, New York, NY, USA, 17–25. https://doi.org/10.1109/ICFEC54809.2022.00010

[59] The Knative Authors. 2024. *Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications*. Retrieved July 2, 2024 from https://knative.dev/

[60] Chao Wang, Yiran Zhang, Qing Li, Ao Zhou, and Shangguang Wang. 2023. Satellite Computing: A Case Study of Cloud-Native Satellites. In *Proceedings of the 2023 IEEE International Conference on Edge Computing and Communications* (Chicago, IL, USA) *(EDGE '23)*. IEEE, New York, NY, USA, 262–270. https://doi.org/10.1109/EDGE60047.2023.00048

[61] Shangguang Wang, Qing Li, Mengwei Xu, Xiao Ma, Ao Zhou, and Qibo Sun. 2021. Tiansuan Constellation: An Open Research Platform. In *Proceedings of the 2021 IEEE International Conference on Edge Computing* (Chicago, IL, USA) *(EDGE '21)*. IEEE, New York, NY, USA, 94–101. https://doi.org/10.1109/EDGE53862.2021.00022

[62] Shangguang Wang, Qiyang Zhang, Ruolin Xing, Fei Qi, and Mengwei Xu. 2024. The first verification test of space-ground collaborative intelligence via cloud-native satellites. *China Communications* 21, 4 (April 2024), 208–217. https://doi.org/10.23919/JCC.fa.2022-0422.202404

[63] Renchao Xie, Qinqin Tang, Shi Qiao, Han Zhu, F. Richard Yu, and Tao Huang. 2021. When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issuess. *IEEE Wireless Communications* 28, 5 (July 2021), 126–133. https://doi.org/10.1109/MWC.001.2000466

[64] Ruolin Xing, Mengwei Xu, Ao Zhou, Qing Li, Yiran Zhang, Feng Qian, and Shangguang Wang. 2024. Deciphering the Enigma of Satellite Computing with COTS Devices: Measurement and Analysis. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking* (Washington D.C., DC, USA) *(MobiCom '24)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3636534.3649371