TU WIEN Informatics

Skylark: Stateful Serverless Functions for the Edge-Cloud-Space 3D-Continuum

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Leonard Guelmino, BSc.
Registration Number 01503940

to the Faculty of Informatics

at the TU Wien

Advisor: Asst. Prof. Dr. Stefan Nastic
Assistance: Dipl.-Ing. Cynthia Marcelino

Vienna, March 11, 2025

Leonard Guelmino	Stefan Nastic

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Skylark: Stateful Serverless Functions für das Edge-Cloud-Space 3D-Kontinuum

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Leonard Guelmino, BSc.
Matrikelnummer 01503940

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Asst. Prof. Dr. Stefan Nastic
Mitwirkung: Dipl.-Ing. Cynthia Marcelino

Wien, 11. März 2025

Leonard Guelmino	Stefan Nastic

Declaration of Authorship

Leonard Guelmino, BSc.

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, March 11, 2025

Leonard Guelmino

Acknowledgements

First, I'd like to thank Assistant Prof. Dr. Stefan Nastic for the opportunity to develop this thesis. Special thanks to Dipl.-Ing Cynthia Marcelino for your consistent and valuable guidance and feedback throughout the process.

I also want to thank my partner, friends, and family for their continuous support in my educational journey.

Thank you, Rózsika, for helping me develop the part of my character I'm most proud of. You've been a better grandmother than I could ever have wished for. Although you left us far too early, a part of you continues to live on through me. I promise to cultivate this part and spread its seeds until the end of my days.

Kurzfassung

Serverlose Funktionen ermöglichen es Entwicklern, sich auf die Anwendungslogik zu konzentrieren, während die Plattform die Infrastrukturverwaltung wie Laufzeitverwaltung und Skalierung abstrahiert. In rein Cloud-basierten Umgebungen stützt sich dieser Ansatz in der Regel auf zentralisierte Cloud-Speicher von Drittanbietern, was zu hohen Latenzen führe kann. Darüber hinaus stammen die Daten, die als Funktionsinput verwendet werden, oft von Sensoren am Netzwerkrand (Edge). Die Daten müssen daher große Entfernungen zum Rechenzentrum überwinden, was die Netzwerkbelastung und Latenzen weiter erhöht. Die Verlagerung der Funktionsausführungsumgebung in die Nähe der Datenquelle kann dem entgegenwirken. Wenn LEO-Satelliten (Low Earth Orbit) in die Datenverarbeitungsstruktur integriert werden und so ein Edge-Cloud-Space (3D)-Kontinuum bilden, wird die Netzwerktopologie aufgrund der Orbitalbewegung der Satellitenknoten dynamisch. Edge-Geräte in abgelegenen Gebieten wie dem Amazonas-Regenwald können LEO-Satelliten nutzen, um dem Netzwerk beizutreten. Ständige Satellitenbewegungen und schwankende Verbindungsqualitäten erschweren jedoch die Entscheidungen über die Datenplatzierung für serverlose Anwendungen, die aus mehreren, voneinander abhängigen Funktionen bestehen. Traditionelle, Cloud-zentrierte Serverless-Modelle sind für Serverless-Workflows, die im heterogenen und dynamischen 3D-Kontinuum eingesetzt werden, nicht geeignet.

In dieser Arbeit wird Skylark vorgestellt, ein neuartiges serverloses Framework für das 3D-Kontinuum. Skylark bietet lokalen Speicher auf jedem Cluster-Knoten und führt den Skylark Elect Service und Skylark SDK ein, um den datentransfer von voneinander abhängigen Funktionen zu optimieren. Ein Netzwerktopologie- und SLO-bewusster Datenübertragungsmechanismus wählt Nachfolgeknoten für die Speicherung des Funktionsausgangszustands auf Grundlage der Netzwerktopologie und Funktionsverfügbarkeit aus. Darüber hinaus verbessert ein Datenbündelungsmechanismus die Effizienz der Datenverwaltung, indem er die Daten verkoppelter Funktionen als eine Einheit für Lese- und Schreibeoperationen behandelt.

Wir evaluieren Skylark anhand eines serverlosen Workflows, der einen Anwendungsfall zur Erkennung von Waldbränden modelliert, welcher die Verarbeitung von Bilddaten erfordert. Unsere experimentellen Ergebnisse zeigen, dass Skylark im Vergleich zu herkömmlichen Architekturen und eine auf Zufall basierende Datenplatzierungsstrategien die Latenz des Workflows um bis zu 33% und die Lesezeit um bis zu 66% reduziert, während der Durchsatz um bis zu 91% erhöht wird. Der Datenbündelungsmechanismus verringert die Latenz bei

der Funktionsausführung um bis zu 24%, indem er die Anzahl von Speicheroperationen minimiert. Diese Ergebnisse zeigen die Fähigkeit von Skylark, die Ausfürhungszeiten von serverlosen Anwendungen im Edge-Cloud-Space-3D-Kontinuum zu verbessern, und ebnen den Weg für effizientere und skalierbare serverlose Anwendungen in dynamischen, heterogenen Umgebungen.

Abstract

Serverless functions enable developers to focus on application logic while the platform abstracts infrastructure management, such as runtime and scaling. In purely cloud-based environments, this approach typically relies on centralized, third-party cloud storage for function state, which introduces latency and bandwidth overhead. Additionally, data used as function input often originates at the network edge but must traverse large distances to a remote data center, exacerbating these overheads. Moving the function execution environment close to the data source (i.e., Edge computing) can ease latency and network strain. When Low Earth Orbit (LEO) satellites are added to the computing fabric, forming an Edge-Cloud-Space (3D) Continuum, the network topology becomes dynamic due to the orbital movement of satellite nodes. Edge devices in remote areas such as the Amazon rainforest can leverage LEO satellites to join the network. However, constant satellite movement and fluctuating link qualities further complicate state placement decisions for serverless applications comprised of multiple, interdependent functions. Traditional, cloud-centric serverless models are ill-suited for serverless workflows deployed in the heterogeneous and dynamic 3D Continuum.

This thesis introduces Skylark, a novel serverless framework designed for stateful serverless computing in the 3D Continuum. The proposed architecture provides local storage on each node and introduces the Skylark Elect Service and Skylark SDK to optimize state management of function input and output state. Two novel mechanisms leverage the architecture: i) a network topology- and SLO-aware state propagation mechanism, responsible for Target Storage election for function output state, and ii) a state bundling mechanism, responsible for enhancing state management efficiency by treating co-located function state as a single unit for data retrieval and migration operations.

We evaluate Skylark through a serverless workflow modeling a wildfire detection use case that requires image data processing. Our experimental results demonstrate that Skylark reduces workflow latency by up to 33% and state retrieval time by up to 66% compared to stateless architectures and random state placement strategies while increasing throughput by up to 91%. The state bundling mechanism decreases function execution latency by up to 24% by minimizing redundant storage operations. These findings highlight Skylark's ability to improve stateful serverless execution across the 3D Continuum, paving the way for more efficient and scalable serverless applications in dynamic, heterogeneous environments.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The rapid evolution of distributed computing has given rise to more flexible and decentralized approaches beyond traditional cloud data centers. Edge computing pushes processing and storage closer to where data originates, significantly reducing latency and bandwidth usage [11, 32]. In parallel, there is growing interest in leveraging Low Earth Orbit (LEO) satellites to extend computation capabilities beyond terrestrial systems, creating an Edge-Cloud-Space (3D) Continuum [39]. The vision is to unite terrestrial Edge nodes, cloud data centers, and satellites into a seamless computing fabric that can host resource-intensive, latency-sensitive, and globally distributed applications [53, 35, 36]. Figure 1.1 illustrates the 3D Continuum, allowing computing nodes such as terrestrial drones and Earth Observation (EO) satellites–previously excluded due to their remote location–to join the network.

Satellites in Low Earth Orbit (LEO) move at speeds exceeding 27,000 km/h and orbit roughly 500–600 km above the Earth's surface. These satellites form a highly dynamic network topology with frequently changing link qualities and intermittent connectivity to ground stations due to their constant movement. Despite these challenges, modern LEO constellations often include high-bandwidth Inter-Satellite Links (ISLs) and robust data routing, making them increasingly attractive for real-time or near-real-time data processing tasks [9], enabling use cases such as wildfire detection and floating algae monitoring through meteorological EO imaging [55], task offloading from remote terrestrial areas such as deserts, forests, and oceans [53], but also competitive gaming and augmented reality over large distances [10].

In parallel with these developments, serverless computing has revolutionized how applications are deployed and managed. Function as a Service (FaaS) abstracts away server administration by running event-driven functions in ephemeral containers. This approach

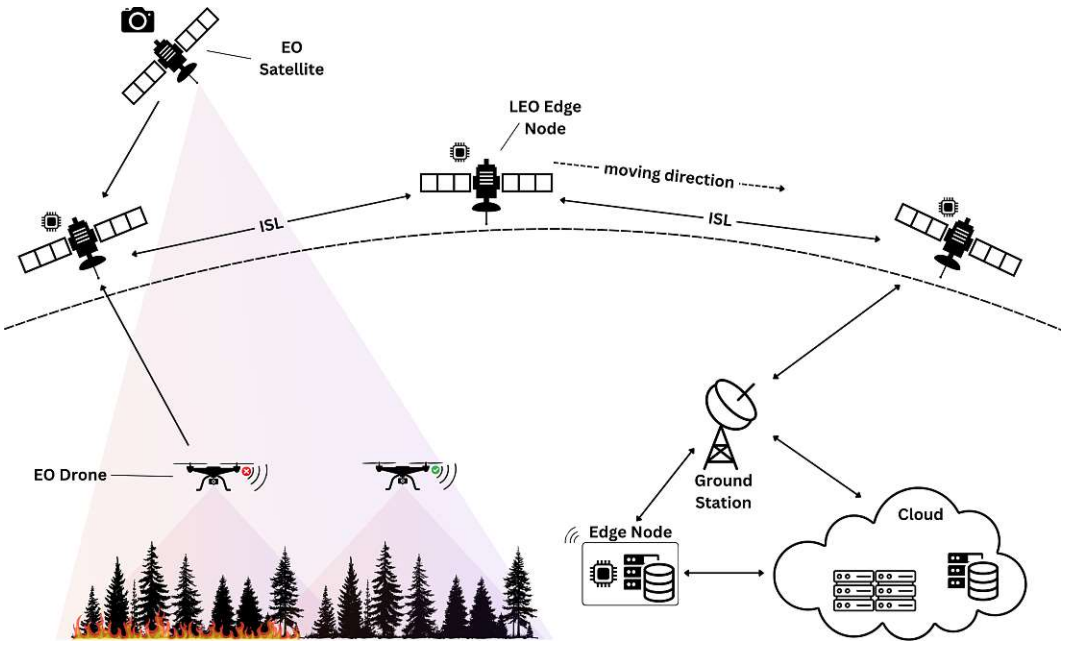


Figure 1.1: Edge, Cloud, and Satellite computing layers in the 3D Continuum

features automatic scaling, including the ability to "scale to zero" and a pay-as-you-go pricing model that charges only for execution time rather than idle uptime [29, 25, 51]. Popular implementations, such as AWS Lambda [46] or Azure Functions [31], relieve developers of infrastructure management overhead, allowing them to focus on application logic. However, integrating serverless computing across terrestrial and satellite nodes introduces new challenges concerning how and where to manage and store function input and output state. Functions rely on third-party services to fetch and store state [30, 7] due to their stateless nature. Co-locating trusted functions (i.e., placing them into a single, sandboxed runtime) relying on each other's state reduces latency [28]. Stateless functions can be composed into serverless workflows, forming larger applications that are still loosely coupled and event-driven [15]. Ensuring that the function state is readily accessible to downstream functions becomes increasingly complex in environments with fluctuating connectivity, latency, resource constraints, and dynamic function readiness.

1.2 Illustrative Scenario

This thesis focuses on an example application highlighting the advantages and challenges of a serverless architecture operating across the 3D Continuum. Consider a global network of edge sensors collecting environmental image and video data that must be processed in (near) real-time [4]. These sensors are deployed in remote regions, such as drones in the Amazon rainforest and EO satellites in Medium Earth Orbit (MEO) or

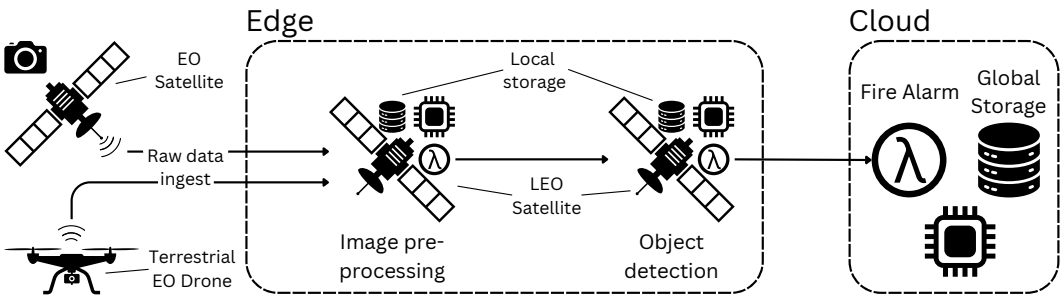


Figure 1.2: Simplified EO image processing workflow

Geostationary Orbit (GEO), connected to LEO Edge computing satellites part of the 3D Continuum. Figure 1.2 illustrates a wildfire detection process modeled as a serverless workflow leveraging the combined computing fabric.

EO satellites and terrestrial drones gather forest video and image data and send it to LEO edge nodes, where the serverless workflow is initiated. Tasks such as combining raw image data, pre-processing, filtering, and object detection can be performed on LEO edge satellites. Resource-intensive tasks such as machine learning inference can only be performed in cloud environments.
Each function in the workflow chain relies on the input state from its successor function to accomplish the task. We refer to the mechanism facilitating the state transition between chained functions as State Propagation [15]. Providing the input state close to the execution environment of the function, therefore, significantly impacts function latency.

1.3 Problem Statement

Despite advances in serverless computing, existing models do not account for the challenges posed by integrating LEO satellites into the current Edge-Cloud Continuum:
EO Satellites. As of now, EO satellites such as the ESA Sentinel 2 satellites each produce around 1.5 TB data per day [3, 5] and rely on dedicated ground stations with typical downlink speeds of around 300Mbps [1], rendering real-time data processing unfeasible. Experiments using ISLs between GEO and LEO satellites have shown bandwidths of up to 100 Gbps [2], drastically improving the outlook on downstream data processing capabilities.
Dynamic node topology. To maintain orbit, LEO satellites must travel at speeds exceeding 27,000km/h, which results in an orbital period of less than 100 minutes [9]. Relative to stationary terrestrial computing nodes, the node topology constantly changes. Topology awareness, therefore, is critical for choosing the location of the function output state.
Storage availability. Following the traditional serverless model [29], the function state

is stored in a central storage in the cloud. Accessing this state from LEO satellites introduces latency and network strain.

Function availability. A key feature of serverless functions is their availability types. A function can be in the warm queue or the cold queue. A cold function is scaled down to zero, while a warm function has resources allocated and is ready to accept requests. Since invoking cold functions introduces cold-start overhead, schedulers avoid doing so in favor of invoking it on a node where the same function is warm.

SLO compliance. Adding thousands of satellites to the computing fabric enlarges the number of storage candidates. However, storing the image of a drone in the Amazon rainforest on a satellite above Japan would be costly compared to a satellite directly above the drone. A Service Level Objective (SLO) can be utilized to reduce the solution space of storage candidates and to align state placement decisions with function execution decisions of SLO-aware schedulers [39, 41, 40].

To our knowledge, a serverless model doesn't exist to address these concerns. This thesis presents a novel model addressing these challenges to enable serverless workflows in the 3D Continuum.

1.4 Contributions

In this thesis, we make the following contributions:

1. **Skylark Serverless Workflow Model.** A serverless Framework for executing interdependent serverless functions on cloud, edge, and satellite nodes. This model ensures that serverless workflows adapt to dynamic network conditions, reducing latency by storing data closer to where it is processed.

2. **Skylark Elect Service Architecture.** A decision-making system that dynamically selects where function output data should be stored based on network conditions and system constraints. This service improves data availability for downstream functions while considering data locality, latency requirements, and function readiness.

3. **Skylark SDK Architecture and a novel State Bundling Mechanism.** Provides developers with tools to manage function state efficiently. By integrating with the Skylark Elect Service, the SDK enables atomic and co-located functions to store, retrieve, and migrate function state seamlessly, reducing the number of storage operations by leveraging a novel State Bundling mechanism. As a result, Skylark decreases function execution latency by up to 24%.

4. **Novel State Propagation Mechanism.** To proactively move the function output state to a location near the successor function that will use it as input. Instead of relying on centralized cloud storage, this approach leverages distributed storage across cloud, edge, and satellite nodes, significantly reducing retrieval delays and

network overhead. Skylark decreases workflow latency by up to 33% and improves throughput by 91%.

1.5 Research Questions

In this thesis, we aim to answer the following research questions:

- **RQ1: How can a serverless computing model be designed to support stateful function execution inside serverless workflows across the Edge-Cloud-Space (3D) Continuum?**
  Distributed computing environments that span terrestrial edge nodes, cloud data centers, and LEO satellites introduce new challenges for serverless computing. Traditional FaaS platforms rely on centralized, third-party storage services [29], which leads to high data transfer latency when nodes are highly distributed [52], as in the 3D Continuum. State-of-the-art approaches introduce techniques to mitigate function overhead, such as function co-location in shared runtimes [48, 8] and shared state via a container runtime shim [28]. However, these solutions generally don't consider function workflow metadata, such as function availability and the current network topology. Consequently, answering RQ1 involves understanding how to incorporate dynamic topology information and SLOs into a serverless framework that can handle function state efficiently across a distributed, heterogeneous computing environment. It also requires mechanisms dealing with function output state placement and representation.

- **RQ2: To what extent can reducing the overhead of state operations improve the efficiency of co-located serverless functions?**
  Due to their stateless nature, serverless functions commonly treat all state as external, leading to network delays and storage costs even when multiple functions share the same runtime environment. Prior work demonstrates overhead reduction through in-memory data sharing of co-located functions [28]. However, this doesn't address state management of function output in serverless workflows. This research question investigates how reducing redundant state input/output operations of functions running on the same host can enhance throughput and decrease overall execution time. It further examines how state-of-the-art techniques—such as shared memory abstractions and lightweight local storage—could optimize state operation overheads.

- **RQ3: To what extent can increasing local state availability and reducing state retrieval latency improve the performance of serverless workflows?**
  Chained serverless functions in a workflow suffer from high state read/write times when function state is stored in a central global storage. Prior research indicates that localized data placement can significantly lower latency [52]. However, it remains unclear how best to manage dynamic State Propagation in heterogeneous environments with LEO edge satellite nodes as part of the computing fabric.

Through this research question, we aim to assess the impact of locality-aware function output State Propagation on end-to-end workflow performance, scalability, and resource consumption. By comparing different state-of-the-art state-placement strategies in serverless computing, the analysis aims to clarify the role of data locality and decision parameters such as function availability and SLOs used by adjacent tools [39, 40, 41].

1.6 Methodology

This work follows a Design Science Research (DSR) process [12], a structured approach for creating and evaluating novel solutions in information systems research. DSR guided each phase of the Skylark project, from defining the problem to demonstrating practical benefits.

1.6.1 Process

We adopted the six-step model proposed by Peffers et al. [34] and applied it as follows:

1. *Problem identification and motivation:* We began by examining how traditional serverless computing struggles with state management in the 3D Continuum, especially when LEO satellites are involved. Through this analysis, we formulated our research questions (Section 1.5).

2. *Define the objectives for a solution:* Drawing on existing literature (Chapter 3) and preliminary experimentation with serverless functions, we established goals to guide Skylark's design. These objectives included i) enabling dynamic, topology-aware, and SLO-aware State Propagation and ii) supporting co-located function state management without incurring high storage operation overheads.

3. *Design and development:* Based on these objectives, we designed the Skylark framework, including its two core components: The *Skylark Elect Service*, which identifies suitable storage locations close to Target Functions while respecting network topology, SLOs and function availability. The *Skylark SDK*, which provides developers with State Bundling and State Propagation tools, integrating seamlessly with serverless environments. We refined Skylark's architecture, data structures, and state management mechanisms during this stage. We also created a prototype implementation (see Chapter 6) to validate Skylark's feasibility.

4. *Demonstration:* We showcase Skylark using a wildfire detection use case, simulating an environment where image data travels through LEO satellites, edge nodes, and cloud nodes. This demonstration highlights how Skylark's State Propagation and bundling reduce the latency associated with traditional remote storage approaches.

5. *Evaluation:* We systematically measured performance and scalability by comparing Skylark's workflows against baseline models in a series of experiments described in

Chapter 7. Key metrics included workflow latency and state retrieval time under varying network topologies and workloads.

6. *Communication:* Finally, we compiled all findings into this thesis, including a detailed account of design decisions, experimental outcomes, an open source prototype, and potential avenues for future work. The results and supporting documentation are intended to provide researchers and practitioners with a clear view of Skylark's benefits and limitations.

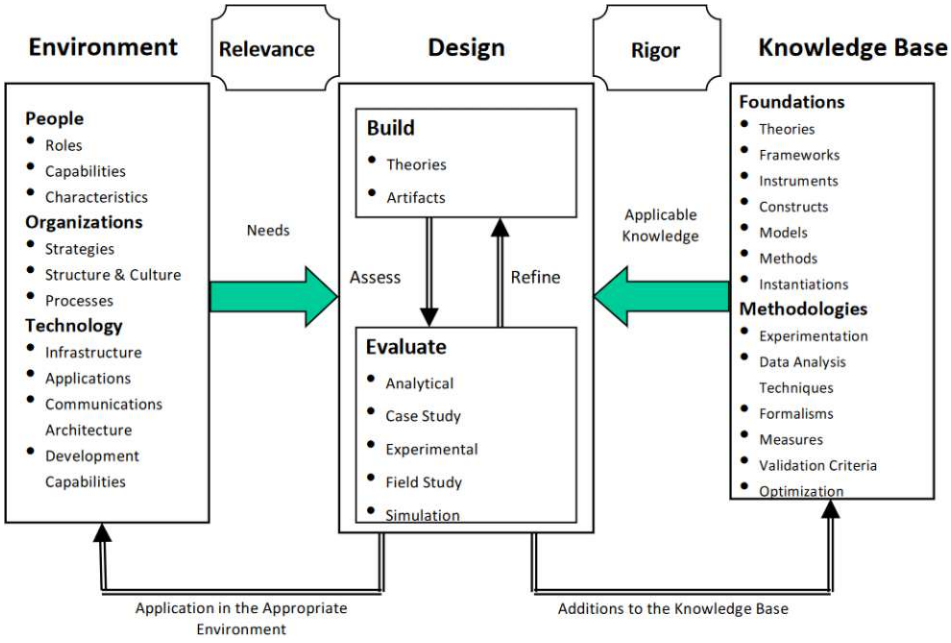


Figure 1.3: DSR methodology model [34]

1.7 Structure

The remainder of this thesis is organized as follows: Chapter 2 provides background information on serverless computing, the 3D Continuum, and related tools and technologies. Chapter 3 reviews existing work on data passing between serverless functions, stateful serverless functions, and function co-location, positioning Skylark within the research landscape. Chapter 4 presents the Skylark model, detailing its stateful serverless workflow model and the architecture of its key components: the Skylark Elect Service and Skylark SDK. Chapter 5 introduces the State Propagation and State Bundling mechanisms, explaining how they leverage Skylark and its components to improve state management in serverless workflows. Chapter 6 describes the implementation of the Skylark framework, including the technologies used and development challenges encountered. Chapter 7

evaluates Skylark through experiments that measure the performance and scalability of the proposed mechanisms. Finally, Chapter 8 concludes the thesis with a summary of findings and discusses potential directions for future research.

CHAPTER 2

Background

This Chapter introduces essential background concepts and technologies relevant to the thesis. Section 2.1 describes the 3D Continuum, highlighting how computing and storage resources can be extended into the LEO environment to form a cohesive global computing fabric. Section 2.2 reviews the fundamentals of serverless computing and its event-driven and stateless properties. It also describes three critical technologies for distributed serverless deployments: *WebAssembly*(2.2.1), which provides a lightweight, secure runtime for portable code execution; *Kubernetes*(2.2.2), an orchestration system for automating the deployment and scaling of containerized applications; and *Knative*(2.2.3), a serverless framework that builds on *Kubernetes* to streamline function management and event-driven serverless workflows.

2.1 Edge-Cloud-Space (3D) Continuum

The vision of the 3D Continuum is to combine Cloud and Edge nodes on Earth and in space such that computing workloads can be executed on each of these layers [39].

Cloud computing is a paradigm that provides on-demand access to computing resources, enabling users to utilize hardware and software services over the Internet with low upfront investments. This model offers elasticity, allowing users to scale resources dynamically based on workload demands, and follows a pay-as-you-go pricing structure, reducing costs associated with over- and underprovisioning [6].
Edge and Fog computing extend traditional Cloud computing by bringing computation, storage, and networking resources closer to data sources, enabling low-latency and context-aware processing. Edge computing refers to processing data at or near its source, often at the level of IoT devices or gateways, to minimize latency and reduce bandwidth usage. Fog computing, on the other hand, creates an intermediate layer between Edge devices and centralized Cloud data centers, offering distributed computing and storage capabilities across a wide geographical area [11].

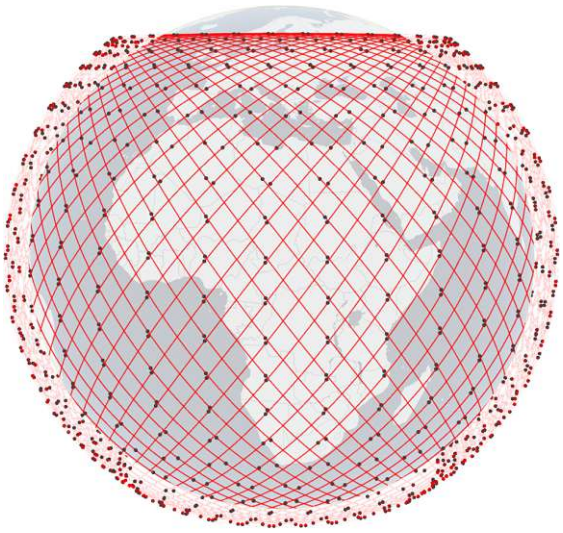


Figure 2.1: Starlink's inner LEO satellite shell at 550km [22]

Driven by advancements in global internet coverage via satellites from companies such as SpaceX [49], LEO satellites enable real-time connectivity for remote regions, mobile platforms like aircraft and ships, and urban areas with limited infrastructure. Figure 2.1 shows Starlink's inner shell of LEO satellites, traveling up to 27,500km/h 550km above Earth's surface. Edge computing is now expanding to include satellite nodes to improve the quality of service for use cases such as augmented reality and earth observation. However, Orbital Edge Computing (OEC) introduces unique challenges such as dynamic satellite movement, limited compute capacity, and lack of physical access for maintenance, necessitating specialized design considerations [36].

2.2 Serverless Computing

Serverless computing is a Cloud-based paradigm that abstracts infrastructure management, allowing developers to focus on application logic. Applications are decomposed into fine-grained, stateless, event-driven functions, often called Function-as-a-Service (FaaS). These serverless functions are executed in isolated environments and scale elastically to accommodate demand, with providers charging on a pay-per-execution basis, aligning costs with actual resource usage [29]. Stateful data is typically managed externally using Backend-as-a-Service (BaaS) platforms like Cloud object stores [30] or databases. This model is characterized by its inherent scalability and cost-efficiency, making it ideal for dynamic workloads [29]. Leading implementations, such as AWS Lambda [46], Azure Functions [31], and Google Cloud Functions [20], exemplify its widespread adoption in the Cloud environment.

The need to store function inputs and outputs in an external data store can elevate

latency and bandwidth requirements. Such overheads become particularly problematic in the 3D Continuum, where connectivity, network topology, and resource availability vary significantly across terrestrial edge nodes, central cloud data centers, and mobile satellite infrastructures. Consequently, the overall performance of serverless functions depends on the location of the function and its associated state.

Edge Functions

The advantages of Edge computing make FaaS at the Edge particularly suitable for latency-sensitive applications, such as IoT, augmented reality, and industrial automation [43]. However, stateless FaaS typically relies on external storage to persist function state, which can lead to increased latency and inefficiencies. Solutions to improve data locality, such as state-aware placement policies and distributed storage, have been proposed. For example, decentralized FaaS platforms like ServerlEdge [43] enable state co-location with functions and dynamic state migration, ensuring data proximity to computation. Additionally, SLO–aware offloading strategies help balance function placement between Edge and Cloud nodes to optimize latency while adhering to performance guarantees [33].

2.2.1 WebAssembly

WebAssembly (Wasm) is a low-level, platform-agnostic bytecode format designed to execute code at near-native speeds in secure, sandboxed environments. Initially introduced for web browsers, WebAssembly has evolved into a general-purpose runtime capable of running on various infrastructures beyond the browser. By acting as a compilation target for programming languages such as C, C++, and Rust, Wasm simplifies cross-platform software development, as developers can write code once and execute it across multiple operating systems without modification.

WebAssembly has a lightweight, isolated execution model. Unlike traditional virtualization technologies—such as full virtual machines or heavy container engines—Wasm has minimal runtime overhead and offers rapid startup times. These properties reduce cold-start latencies of serverless functions and improve scalability in dynamic environments such as the 3D Continuum. Furthermore, WebAssembly's sandboxing features help ensure security by confining code execution to a restricted environment, mitigating the risks associated with multi-tenant architectures [50].

2.2.2 Kubernetes

Kubernetes is an open-source container orchestration platform that automates containerized applications' deployment, scaling, and management across clustered computing environments [24]. By abstracting away low-level operational details such as load balancing, resource allocation, and fault tolerance, Kubernetes enables developers to focus on application logic while benefiting from robust container scheduling and dynamic scaling. Central to its design is the concept of declarative configuration, wherein operators specify

a desired application state (for example, the number of replicas or version constraints), and Kubernetes continually monitors and enforces this target state.

Beyond its widespread adoption in traditional cloud data centers, Kubernetes also supports specialized, lightweight distributions—such as MicroK8s [14]—tailored for resource-constrained edge environments. This flexibility allows Kubernetes to span a range of deployments, from powerful cluster nodes in large data centers to smaller edge nodes closer to data generation sources. Consequently, Kubernetes is well-suited for workloads in the 3D Continuum, where application components may be geographically dispersed and subject to heterogeneous network conditions.

2.2.3 Knative

Knative is an open-source extension to Kubernetes designed to support serverless workloads through automated scaling, routing, and event-driven execution [23]. It builds upon Kubernetes container orchestration capabilities by introducing higher-level abstractions for deploying and managing functions and microservices. Specifically, Knative Serving manages how workloads are scaled in response to traffic, including the ability to scale to zero during idle periods, thus reducing resource usage and operational costs. Knative Eventing provides a standardized event handling and routing mechanism, enabling developers to integrate external services and data streams easily.

Through these features, Knative offers a simplified workflow for building serverless applications on top of Kubernetes. Developers can package their code as containers, define minimal configurations, and let Knative handle instance provisioning and autoscaling. This architecture is especially beneficial in heterogeneous environments like the 3D Continuum, where computing resources must adapt to dynamic workloads and fluctuating network conditions.

CHAPTER 3

Related Work

This chapter provides an overview of existing research on stateful serverless computing, focusing on both architectural and performance aspects relevant to the 3D Continuum. Section 3.1 describes prior work on how serverless functions exchange and store data, including techniques that reduce overhead by enabling local, in-memory data sharing. Section 3.2 outlines existing methods to incorporate and manage state in serverless applications, emphasizing approaches that balance flexibility with low-latency data access. Finally, Section 3.3 discusses function co-location and bundling strategies, highlighting solutions that reduce cold-start penalties and redundant storage operations.

3.1 Data Passing between Serverless Functions

Shillaker and Pietzuch explore enhancements in serverless architectures to handle stateful applications efficiently. Traditional serverless frameworks manage state externally, leading to significant overheads from data duplication and movement. The study introduces *Faaslets*, an abstraction layer that uses software-fault isolation (SFI) to allow memory sharing directly between functions, thus minimizing data movement and duplication. This model utilizes WebAssembly for memory isolation and supports efficient state sharing across serverless functions. Their experimental results demonstrate a significant reduction in initialization times and resource consumption, alongside improvements in runtime performance for data-intensive tasks, such as machine learning training and inference [48]. However, their approach is limited to functions co-located on the same machine. In contrast, Skylark also extends State Propagation to include remote target storage, optimizing function state management across cloud, edge, and space nodes.

Cicconetti et al. explore possible solution models for function state transitions in edge networks: (i) a *pure FaaS* implementation, which adheres to the conventional stateless serverless model where each function invocation is independent, and state management is entirely the client's responsibility, without any optimizations for state handling. (ii)

StateProp, a model involving the propagation of the application state through the chain of function invocations, allowing each function to access and potentially modify the state before passing it along to the downstream function. (iii) *StateLocal*, where the state is kept local to the serverless platform executing the function, reducing data transfer overhead by fetching the state only when necessary from the local environment rather than remote locations [15]. Skylark builds on these concepts but extends State Propagation strategies to the highly dynamic 3D Continuum, accounting for satellite movement, SLOs, and function availability.

3.2 Stateful Serverless Functions

Cloud

Azure Durable Functions uses event sourcing to manage state, with the function's execution history stored in storage queues or tables. This allows the orchestration to replay events to reach the current state, supporting complex, long-running workflows. However, this approach can introduce latency due to the need to read and write the state to external storage [13]. Similar solutions by big vendors include Amazon's Step functions [47]. These solutions assume resource-rich cloud environments with centralized storage. Skylark provides local storage instances on all layers of the 3D Continuum and provides a network topology-aware State Propagation mechanism.

Barcelona-Pons et al. introduce *Crucial*, a system designed to address challenges posed by inter-function communication. This approach incorporates a Distributed Shared Object (DSO) layer, which operates on top of a low-latency in-memory data store, providing strong consistency and enabling fine-grained updates and coordination between serverless functions [8]. Skylark differs by focusing on optimizing state placement and access locality rather than maintaining shared consistency across multiple functions.

Edge

Puliafito et al. propose a dynamic model where serverless functions can switch between remote-state and local-state modes depending on the situational requirements at the edge. This flexibility allows functions to manage state more efficiently—locally when low latency and minimal network traffic are needed or remotely when broader accessibility and cost efficiency are prioritized [38]. Our work differs by integrating State Propagation with node topology and SLO awareness, ensuring that the function state remains close to the execution environment while adapting to changing network conditions.

Pfandzelter et al. introduce *Enoki*, a framework enabling stateful serverless operations over distributed edge and cloud environments by directly integrating a replicated key-value store in the FaaS nodes. This reduces latency while ensuring consistency across distributed nodes. While Enoki optimizes intra-node state access, Skylark focuses on inter-node State Propagation, ensuring efficient state transitions across a distributed, multi-layered computing environment.

Nardelli and Russo studied SLO-aware offloading of functions and introduced a state-aware data migration algorithm. The algorithm decides whether a single key part of a key-value store should be migrated to another node based on (i) data access time, (ii) migration time, (iii) SLOs, and (iv) cost minimization. Our work is different since we have to consider the change of geo-location of satellites over time [33]. Our work extends this concept by considering the changing geographic positions of satellite nodes and adapting State Propagation dynamically to maintain low-latency state availability.

3.3 Function Co-Location and –Bundling

Marcelino and Nastic highlight the inefficiencies in inter-function communication when serverless functions are co-located yet rely on remote services, which often leads to increased latency and network overhead. To combat these inefficiencies, they introduce *CWASI*, a runtime shim that implements a novel three-mode communication model, optimizing inter-function communication based on function locality. This model significantly reduces reliance on external services, decreasing virtualization overhead [28]. Skylark builds upon this approach by dynamically deciding the optimal State Propagation strategy within a single node and across distributed nodes in a dynamic topology. Co-located functions using the CWASI runtime shim can also leverage Skylark's State Bundling mechanism to decrease the number of remote storage operations.

Czentye et al. introduce function fusion, a novel method for combining interdependent functions into composite units managed as single serverless artifacts. This technique mitigates cold starts, reduces communication latency, and improves resource utilization by leveraging internal state propagation and implicit instance parallelization. The authors provide theoretical foundations for the cost and latency models underlying function fusion and propose efficient algorithms, such as bicriteria approximation and greedy heuristics, to achieve cost-optimal partitioning of serverless workflows [16]. Our work differs by applying state-aware function embedding and State Bundling to reduce redundant storage operations in multi-node, dynamic topologies.

Schirmer et al. expand on function fusion with their framework, FUSIONIZE++, which dynamically optimizes serverless applications through task inlining and infrastructure adjustments. Recognizing inefficiencies such as remote invocation overheads, cascading cold starts, and double billing during synchronous function calls, they propose an iterative optimization approach. The framework monitors application execution, identifies call patterns, and employs heuristics to reconfigure fusion groups—task clusters executed within the same function—to minimize latency and cost. Additionally, FUSIONIZE++ adjusts resource allocations like memory and CPU shares to enhance execution efficiency. This feedback-driven mechanism allows real-time adaptation to workload changes without requiring developer intervention [45]. Skylark, in contrast, focuses on optimizing state transitions independently of function placement, ensuring efficient function execution even in environments with rapidly changing node topologies.

CHAPTER 4

Skylark Serverless Workflow Model and Architecture Overview

Skylark is a topology- and SLO-aware serverless state management framework. It reduces serverless workflow latency by placing the function input state close to the local execution environment and by enabling State Bundling for Embedded Functions. Skylark achieves this by providing local storage instances on nodes throughout the 3D Continuum and by introducing a distributed service we call Skylark Elect Service for Target Storage selection. In addition, we introduce Skylark SDK, which enables developers to fetch function input state and propagate output state in collaboration with the Skylark Elect Service. Skylark SDK further helps reduce the number of necessary state operations by offering State Bundling functionality for Embedded Functions.
This Chapter has two Sections. Section 4.1 describes the Skylark Serverless Workflow Model, and Section 4.2 describes the architecture of the main components of Skylark.

4.1 Skylark Serverless Workflow Model

This section describes the Skylark Serverless Workflow Model, defining how serverless functions are composed, executed, and coordinated throughout the 3D Continuum. By specifying the function sequences' structure and how data is propagated, this model provides a framework for managing state and ensuring low-latency, SLO-compliant operation across a distributed environment characterized by a dynamic network topology.

Network Topology

A serverless cluster in the 3D Continuum consists of 3 types of nodes: Cloud nodes $N_C \subseteq N$, terrestrial edge nodes $N_E \subseteq N$, and space satellite nodes $N_S \subseteq N$. Figure 4.1 depicts all node types deployed within the 3D Serverless Platform. A LEO space node

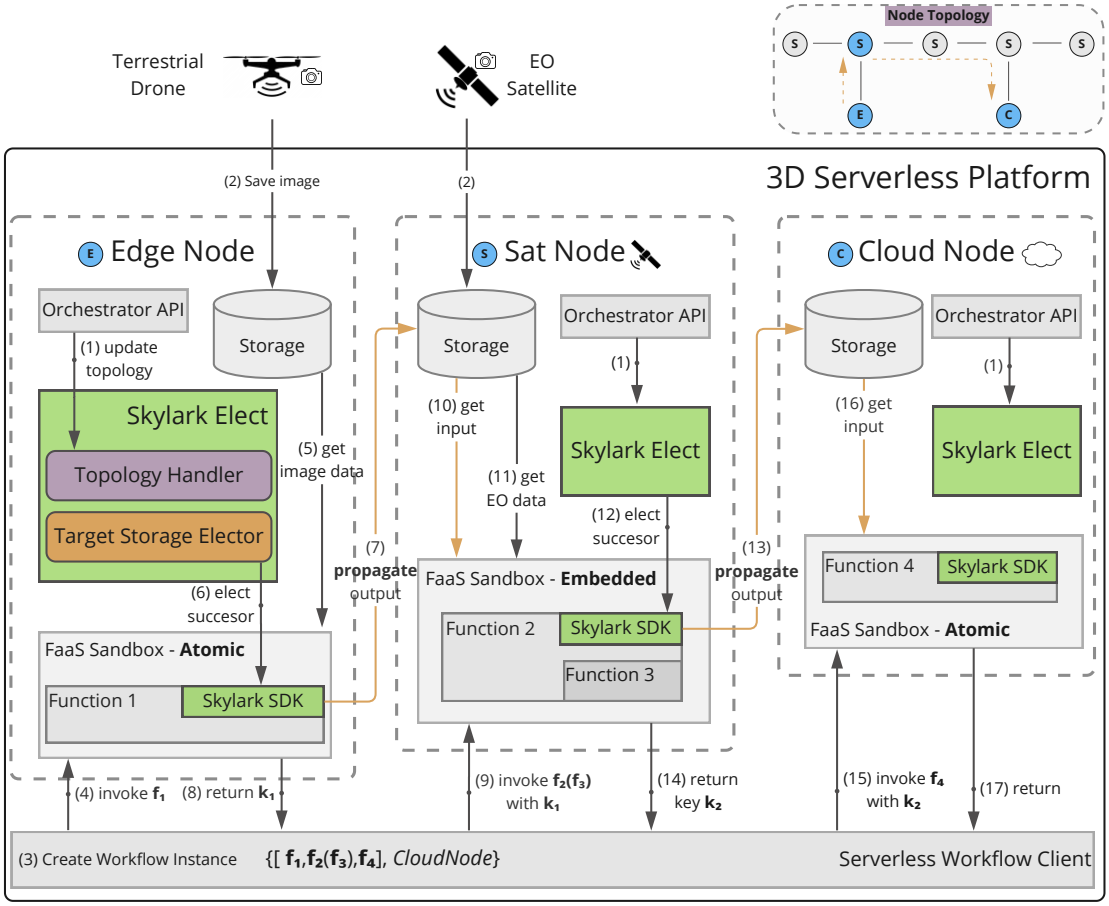


Figure 4.1: Skylark Serverless Workflow Model

must travel around 27,000km/h to maintain an orbit 550km above Earth's surface [9]. Therefore, the cluster Node Topology changes over time t, as illustrated in Figure 4.2. At $t = 0$, the Cloud Node can only see Sat Node 5. At $t = 1$, Sat Node 6 is visible. Similarly, at $t = 0$, the Edge Node can only see Sat Node 2. At $t = 1$, Sat Node 3 is visible. From the perspective of a Sat Node, neighboring satellite nodes in the same orbit and trajectory don't change over time. Additionally, compared to the edge and cloud nodes, satellite nodes have a constrained power supply, fixed computing capabilities, and experience temperature fluctuations as they move in and out of Earth's shadow [27]. In contrast to edge and satellite nodes, cloud nodes can perform resource-intensive tasks such as machine learning model inference. A node $n \in N$ provides the execution environment for functions. We denote the connection of nodes $n_i, n_j \in N$ via edges $e_{i,j} \in E$. The network delay or Round-Trip-Time (RTT) and bandwidth characterize $e_{i,j}$. Therefore, the Node Topology is the composition of nodes and their connections at a time.

Symbol	Description
N	Set of cloud, edge, and satellite nodes
$e_{i,j}$	Network latency and bandwidth between $i, j \in N$
F	Set of functions
f_i	Atomic Function, running alone in a sandbox
$f_i(f_j)$	Embedded Function, sharing a sandbox
$\|f\|$	The number of functions in a sandbox (Depth)
D_A^f	Atomic output state of $f \in F$
D_E^f	Embedded output state of $f \in F$
$\|D\|$	State size
k^f	SkylarkKey for the output state of a function $f \in F, k^f \in D^f$
W	Serverless workflow

Table 4.1: Main notation adopted in the thesis

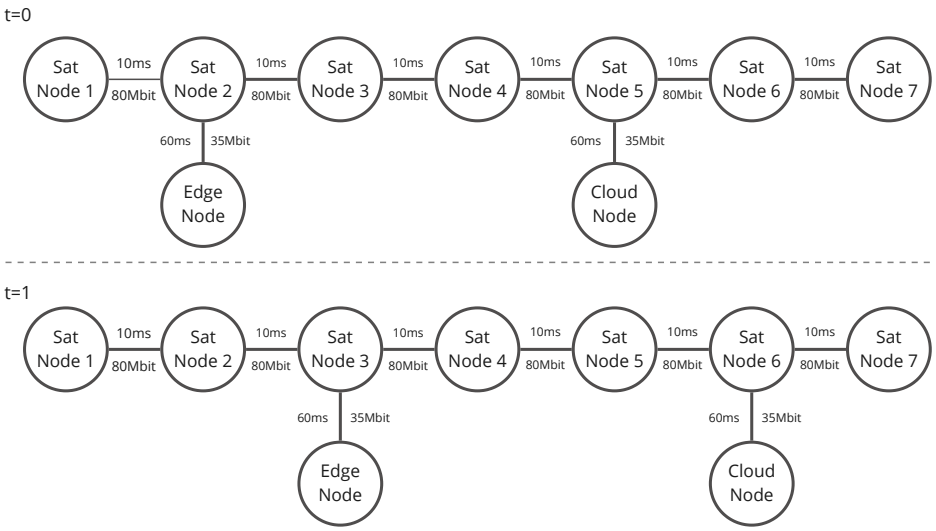


Figure 4.2: The changing Node Topology of LEO satellite nodes relative to terrestrial nodes over time

Serverless Workflow

A serverless workflow consists of a sequence of functions expecting an input state and producing an output state. We define functions and state as follows: Let F be the Set of all deployed serverless functions on a cluster. A serverless function $f \in F$ represents a

single computational step inside a serverless application. The lifecycle of f is i) invocation, ii) execution, and iii) termination [29]. f is stateless insofar as it doesn't retain state after termination and, therefore, relies on external storage services for state management. However, f expects the input state upon invocation and returns the output state upon termination. A Set of functions $f_i, f_j, f_k \in F$ can either run alone in their sandbox environment $\{f_i, f_j, f_k\}$ (Atomic Function) or together $\{f_i(f_j, f_k)\}$, sharing a single sandbox (Embedded Function). We call the number of functions placed inside a sandbox its depth and denote it as $|f|$. An Atomic Function f_i has a depth of 1 ($|f_i| = 1$). Conversely, an Embedded Function with one parent and two children $f_i(f_j, f_k)$ has a depth of 3 ($|f_i| = 3$).

Let the function output state be a key-value pair $D := \{k, v\}$ and $f_1, f_2 \in F_S$ a sequence of two Atomic Functions. f_1 is the predecessor of f_2 and f_2 is the successor of f_1. f_1 produces function output state $D_A^{f_1}$, which f_2 expects as function input state upon invocation. We refer to a storage instance chosen for storing the function output state henceforth as Target Storage.
We denote the bundled state of an Embedded Function as $D_E^{f_i}$ (Embedded State), therefore representing the states of multiple functions as a single entity, where f_i is the parent function. We denote the state size as $|D|$.

Let a Serverless Workflow be a tuple $W := \{F_S, n_T \in N_C\}$ consisting of a sequence of functions $f_1, ..., f_n \in F_S \subseteq F$ and a Tail Node $n_x \in N_C$. Given a sequence of two Atomic Functions $\{f_i, f_j\} \in F_S$, the workflow first invokes f_i and only after its termination invokes its successor function f_j. Hence, f_i is the predecessor function of f_j. The output state produced by f_i is the input state of f_j. The last function of the sequence $f_n \in F_S$ is assumed to be a resource-intensive task, which has to be executed on one of the cloud nodes N_C. The Tail Node $n_x \in N_C$ represents a node on which the last node of the sequence f_n is in the warm queue. Henceforth, we refer to the successor function as the Target Function.

4.1.1 Constraints

Service Level Objective (SLO)

SLOs define upper bounds on performance metrics such as network latency for data retrieval and propagation. In Skylark, these objectives constrain the Set of feasible storage nodes for each state transition. By specifying, for example, a maximum allowable round-trip time, Skylark ensures that function output data is only placed where the SLO can be upheld. This targeted placement keeps data within a suitable distance to the function's execution environment, thereby limiting retrieval delays and preventing the overhead of storing state on nodes that cannot meet the required latency guarantees.

Data Locality

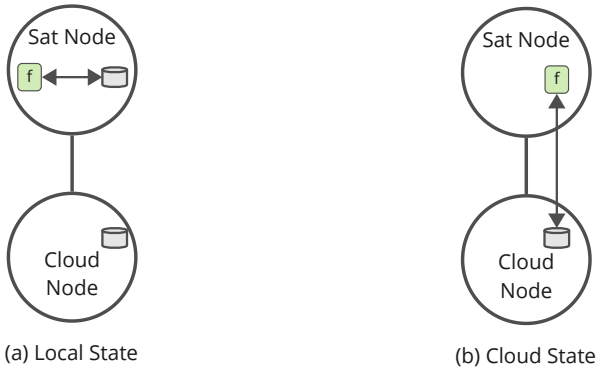


Figure 4.3: Local State vs. Global State

Data locality in Skylark ensures that functions retrieve and store state near their execution environments, lowering latency and reducing network overhead. Instead of relying on centralized cloud storage for state management (Figure 4.3b), Skylark proactively places function output in local or nearby storage nodes (Figure 4.3a). This approach helps maintain efficient data access in dynamic edge, cloud, and satellite deployments, where physical distances and resource constraints can otherwise introduce significant performance penalties. By aligning function input state placement with its actual execution environment, Skylark increases data locality.

Function Availability

Function availability in Skylark refers to whether a function instance is actively running (warm) or scaled to zero (cold). Cold functions conserve resources but require additional time to start, which can introduce significant latency. To address this, Skylark prioritizes storing output data where a successor function remains warm, thereby reducing the need to spin up new instances. In doing so, State Propagation and placement decisions align with function availability, ensuring minimal startup delays and efficient resource utilization across the 3D Continuum.

4.2 Skylark Architecture

This Section describes the architecture of Skylark's main components, Skylark Elect Service and Skylark SDK, in the context of a serverless platform in the 3D Continuum. Edge, space, and cloud nodes run local storage instances by default. However, due to resource restrictions, an edge node may be unable to host a storage instance. Therefore, individual nodes that do not host storage instances are allowed.

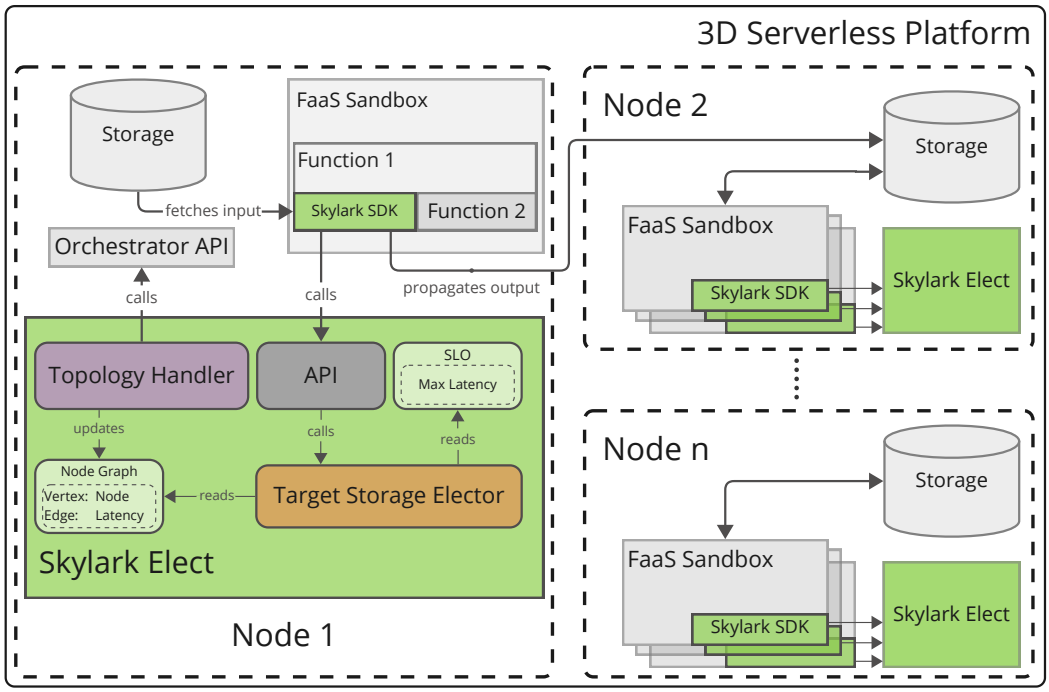


Figure 4.4: Architecture of the Skylark Elect Service

4.2.1 Skylark Elect Service

Figure 4.4 provides an overview of the Skylark Elect Service architecture. The Skylark Elect Service is platform-agnostic and modular, enabling seamless and distributed integration into the 3D Continuum as a stand-alone service running in an isolated and lightweight Wasm sandbox. It enables serverless functions, part of a serverless workflow sequence, to manage function output state, thus increasing input state data locality for Target Functions. Skylark Elect Service is composed of three main modules: *Topology Handler*, *Target Storage Elector*, and *API*.

Topology Handler

The Topology Handler maintains an accurate and up-to-date representation of the dynamic cluster Node Topology by periodically querying the orchestrator API. We call this internal representation `NodeGraph`. It contains information about nodes such as name, IP, and type (*Cloud, Edge* and *Satellite*). Moreover, `NodeGraph` represents real-time network properties such as RTT and bandwidth between nodes (*Edge*). The Topology Handler enables the Target Storage Elector to access accurate and up-to-date Node Topology information.

Target Storage Elector

Called by a serverless workflow function via the *API*, the Target Storage Elector chooses the storage location of the function output state based on the `NodeGraph` and SLOs. The `NodeGraph` is maintained by the Topology Handler, and the user provides the SLOs. Specifically, the Target Storage is elected based on:

- `NodeGraph`: An up-to-date representation of all cluster nodes. A cluster in the 3D Continuum contains thousands of nodes. Most of them would violate the SLO requirements. We, therefore, reduce the solution space to the nodes part of the shortest path between the current node and the Tail Node of the serverless workflow. We call these nodes candidate nodes.
- SLO: The maximum network RTT between the local node and the node hosting a potential Target Storage.
- Target Function: The successor of the caller function. By contacting the orchestrator, the Target Storage Elector considers whether the Target Function is warm or cold on the candidate node.

The Elector chooses the Target Storage by finding the closest candidate node to the Tail Node, satisfying the maximum network RTT and having the Target Function in the warm queue. The host information of the Target Storage is returned to the API.

API

Passes calls from functions to the Target Storage Elector and returns the result to the caller. When a serverless workflow function finishes its execution tasks, it wants to know where to store the function output state. The caller also provides workflow metadata such as the Tail Node and SLOs.

4.2.2 Skylark SDK

Skylark SDK enables State Bundling for Embedded Functions by leveraging function embedding properties such as locality, trust, and shared state. Trusted functions are grouped into one sandbox, allowing them to share resources such as shared memory, heap memory, and registers. Developers can utilize the Skylark SDK to fetch the function input state, propagate the function output state, or delete the input/output state.
Additionally, Skylark SDK plays a role in function output State Propagation within serverless workflows, as described in Section 4.1. When a function finishes its task, the output state is passed to Skylark SDK to initiate State Propagation. Skylark SDK contacts the local Skylark Elect Service to elect a Target Storage close to the execution environment of the Target Function. Skylark SDK then creates a SkylarkKey (key) and stores the output state (value) as a key-value pair in the Target Storage. The function

returns the generated SkylarkKey upon termination to the workflow client. The workflow client supplies the SkylarkKey as input to the Target Function to access its input state.

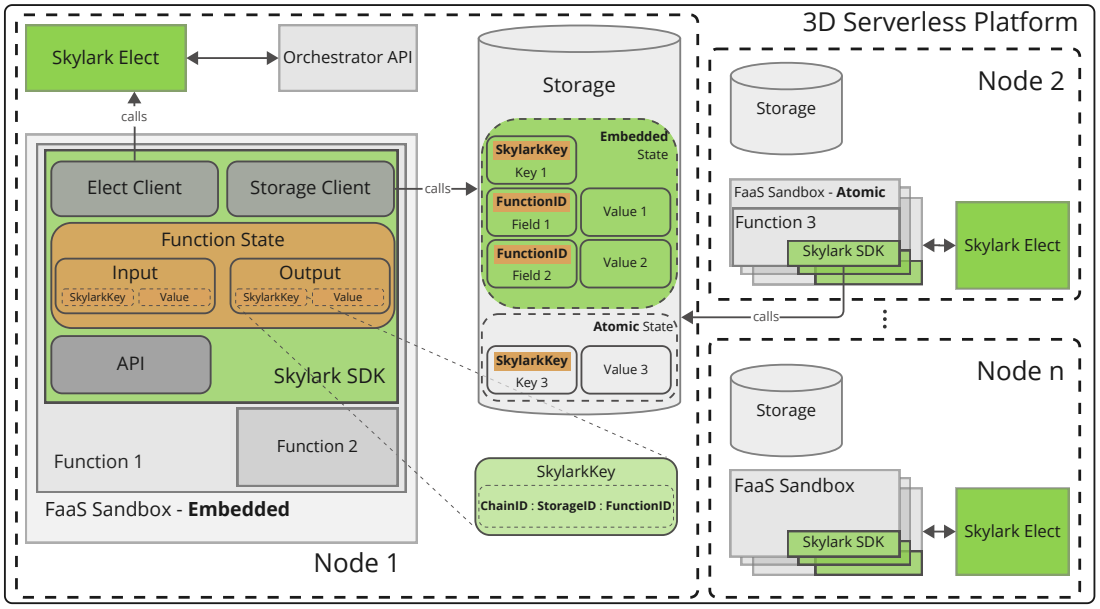


Figure 4.5: Architecture of Skylark SDK

Figure 4.5 shows the Skylark SDK architecture integrated into the 3D Continuum. It is composed of four main modules: *Function State*, *API*, *Elect Client*, and *Storage Client*.

Function State

This module holds the input state from the predecessor and the output state produced by the function. Both are passed through the API module. The output state is propagated to some Target Storage elected by the Skylark Elect Service. It then serves as the input state for the Target Function. Each state object is identified by a SkylarkKey $k := \{k_C, k_S, k_F\} \in D$. It consists of three parts:

1. `ChainID` (k_C): Identifies the Workflow instance. Given two workflows $W_1 = \{F_1, n \in N\}, W_2 = \{F_2, n \in N\}$, k_C is identical for all $f \in F_1$. However, keys of $f \in F_2$ have a different ChainID.

2. `StorageID` (k_S): Each local storage hosted on a node in the 3D Continuum can be addressed with a unique identifier. We encode this identifier as k_S into the SkylarkKey to enable Target Functions to access their input state even if it is unavailable on the local storage. This is important since the Skylark Model doesn't guarantee local state availability; rather, it increases local state availability through the State Propagation mechanism explained in the next Chapter.

3. `FunctionID` (k_F): Identifies a single function. Given two workflow instances with identical function sequences $W_1 = \{F_1, n \in N\}, W_2 = \{F_2, n \in N\}, F_1 = F_2$, two functions on the same position in the sequence $f_j = f_i, f_i \in F_1, f_j \in F_2$ have the same FunctionID $k_F^{f_j} = k_F^{f_i}$.

Function input and output states can be Atomic or Embedded. Given a Atomic Function and a Embedded Function $f_1(f_2), f_3 \in F$ as illustrated in the storage of Node 1 in Figure 4.5:

- `Embedded State`: A state object holding multiple functions' input or output state in a Set-like data structure. In this data structure, a SkylarkKey identifies the Set, which contains pairs of `FunctionIDs` and state String values. f_1 is the parent, and therefore, its SkylarkKey k^{f_1} is used to identify the Set. f_1 and f_2 store their state into this set, using their corresponding `FunctionID` $k_F^{f_1}, k_F^{f_2}$ for value identification.
- `Atomic State`: A single key-value pair. f_3 creates a SkylarkKey to store its Atomic State into the storage. A SkylarkKey is used as a key k^{f_3}, and the value is a String like in Embedded State.

The Function State module interacts with all other modules of the Skylark SDK. It uses the Elect Client module to interact with the Skylark Elect Service for State Propagation. It passes function input and output state to the Storage Client to execute storage operations such as get, set, and delete. The Function State module exposes its functionality via the API module.

API

This module exposes the core functionality of the Skylark SDK to the developer. It accepts calls to the following state operations:

- `fetch input`: Expects the SkylarkKey k of the predecessor function as input. Fetches the output state of the predecessor function from the storage encoded in the SkylarkKey $k_S \in k$. Returns the input state to the caller.
- `propagate output`: Expects a state value (Atomic) or a list of state FunctionID-value pairs (Embedded) as input. Passes the state to the Function State module, which elects a Target Storage using the Skylark Elect Service. After successfully propagating the function output state to the Target Storage using the Storage Client, the SkylarkKey is returned to the caller.
- `delete`: Expects a SkylarkKey and state type as input. Deletes the state from the location encoded in the key.

The API exposes this functionality for Embedded State and Atomic State and passes the requests to the Function State module.

Storage Client

This module is responsible for executing storage operations such as **get**, **set**, and **delete** for Embedded State and Atomic State. The Function State module calls the Storage Client to execute storage operations on local (i.e., on the same node as the function) and remote (i.e., on another node in the cluster) key-value stores such as Redis [42].

Elect Client

This module enables the Skylark SDK to interact with the Skylark Elect Service via a standard interface. The Function State module calls the Elect Client and expects workflow metadata such as Target Function, SLOs, and Tail Node, which are then passed to the Skylark Elect Service. The elected Target Storage is finally returned to the Function State module.

4.2.3 Assumptions

SLO-aware Scheduler

The Skylark Framework assumes the presence of an SLO-aware scheduler, such as HyperDrive [39], Vela [41], or Polaris [40]. These schedulers are designed to dynamically allocate and manage serverless functions across distributed nodes based on SLOs, network conditions, and resource availability. HyperDrive offers topology-aware scheduling by continuously evaluating node status and network metrics to optimize function placement. Vela implements a multi-phase scheduling mechanism that ensures low-latency execution by leveraging real-time data from edge and cloud nodes. Polaris enhances microservices scheduling by incorporating SLO constraints and dynamic topology changes.

Although Skylark doesn't include a scheduler, its mechanisms depend on such a scheduler to ensure that serverless functions are placed efficiently. The assumed SLO-aware scheduler complements Skylark's State Propagation and State Bundling mechanisms. Without an SLO-aware scheduler, Skylark's ability to efficiently propagate and bundle state would be significantly hindered, as the framework chooses Target Storages based on accurate, up-to-date information about function availability and network SLOs.

Function embedding

Skylark also assumes a function embedding mechanism, such as the CWASI runtime shim [28], which enables multiple functions to share a single sandbox environment. This assumption is crucial for Skylark's State Bundling mechanism, which optimizes state management by treating co-located function states as a single entity. Without such embedding mechanisms, the State Bundling feature of Skylark would not be feasible, as its efficiency hinges on the co-location of functions within a shared execution environment.

CHAPTER 5

Skylark Mechanisms

The mechanisms introduced in this chapter build upon the Skylark model presented in Chapter 4 to enhance stateful serverless workflows in the 3D Continuum. Skylark employs two core mechanisms: the Skylark State Propagation mechanism and the Skylark State Bundling mechanism. The State Propagation mechanism, described in Section 5.1, ensures that the function state is placed close to its execution environment by leveraging the Skylark Elect Service and local storage instances, thereby reducing latency and improving data locality. The state bundling mechanism, described in Section 5.2, enables co-located functions within a shared sandbox by leveraging the Skylark SDK to minimize storage operations. Both mechanisms operate within the dynamic node topology of the 3D Continuum, where function placement, state management, and inter-node communication must adapt to changing connectivity and performance constraints. By integrating these mechanisms, Skylark addresses key challenges of stateful serverless execution, reducing data transfer overhead and enhancing workflow responsiveness across the 3D Continuum.

5.1 State Propagation Mechanism

To achieve high local state availability of the function input state, Skylark leverages local storage, the Skylark SDK, and the Skylark Elect Service to propagate the function output state close to the Target Function 's execution environment.

Function output State Propagation has 4 phases, as illustrated in Figure 5.1: In ①, a function leverages the Skylark SDK API to call the Skylark Elect Service to INITIALIZE the State Propagation process. The Skylark Elect Service reads the current node topology and reduces the solution space only to include candidate nodes in the path between the Head Node (i.e., the current node) and the Tail Node. In the next phase in ②, two filters are applied to IDENTIFY all viable Target Storages. The first filter removes all nodes violating SLOs provided by the user. The second filter removes all nodes on which the Target Function is cold. In ③, the Skylark Elect Service ELECTs the Target Storage by

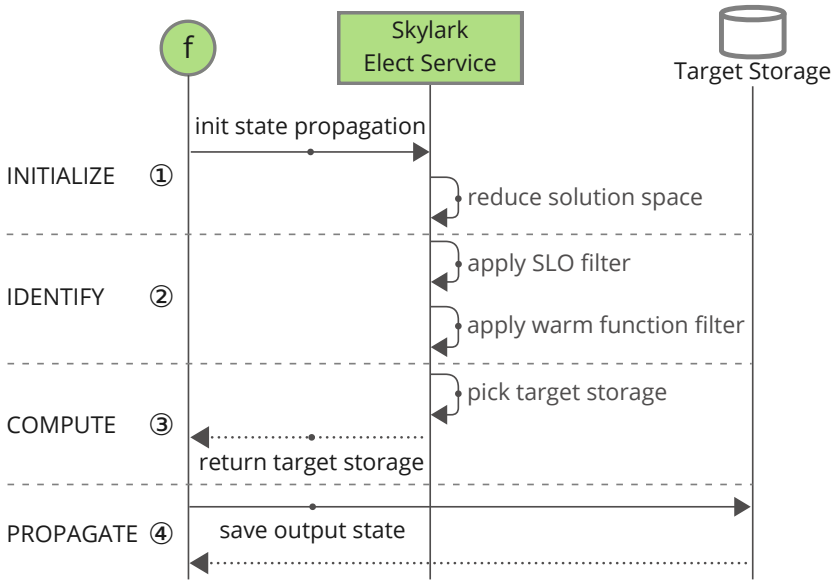


Figure 5.1: Skylark State Propagation

picking the storage closest to the Tail Node and returns its host information to the caller function. In ④, the PROPAGATE phase, the Skylark SDK stores the function output state to the elected Target Storage, thereby finishing the State Propagation process. Next, we illustrate the State Propagation mechanism with an example workflow.

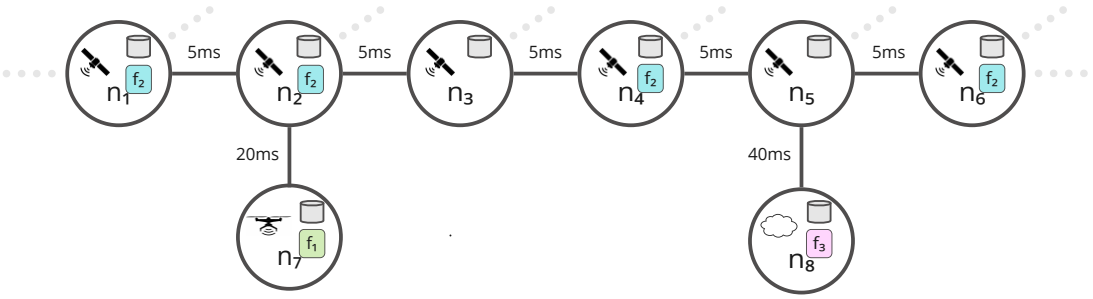


Figure 5.2: Example node topology containing edge, space, and cloud nodes with warm functions and local storage

Figure 5.2 shows part of an example node topology within the 3D Continuum using the Skylark Framework. The network delay between the nodes is also depicted. $n_1, ..., n_6$ are satellite nodes, n_7 is an edge node and n_8 is a cloud node. All nodes have local storage instances, and warm functions are marked. Consider a serverless workflow with a sequence of 3 functions $W = \{[f_1, f_2, f_3], n_8\}$, where the last function of the sequence

f_3 is a resource-intensive task requiring a cloud environment. f_3 is warm on n_8 when the workflow instance is created, therefore n_8 is the Tail Node of the serverless workflow. Additionally, consider a network RTT SLO of $T_{max}^{RTT} = 60ms$. This example demonstrates the Skylark State Propagation mechanism by examining the state transition of $f_1 \rightarrow f_2$. f_1 is executed on the edge node n_7 (Head Node in the first state transition). After finishing its main execution task, f_1 uses the `propagate output` interface of the Skylark SDK to initiate the State Propagation process (phase ①), calling the Skylark Elect Service. There, the Target Storage Elector first reads the whole node topology of the cluster as depicted in Figure 5.2. All of the nodes are capable of storing the output state of f_1. Since f_2 relies on this state as input, the mechanism aims to store it close to the local execution environment of f_2 by considering SLOs and warm functions. In this phase, the nodes n_1, n_2, n_4 seem viable as Target Storage picks since f_2 is warm and all candidates fulfill the SLO requirements (n_6 has f_2 in warm-queue but violates the SLO). However, since we know that f_3 will likely be executed on n_8, we also want to minimize the state transition distance from f_2 to f_3. Therefore, placing the output state of f_1 on n_1 is less costly than on n_4, but the subsequent state transition $n_1 \rightarrow n_8$ is far more expensive than the state transition $n_4 \rightarrow n_8$.

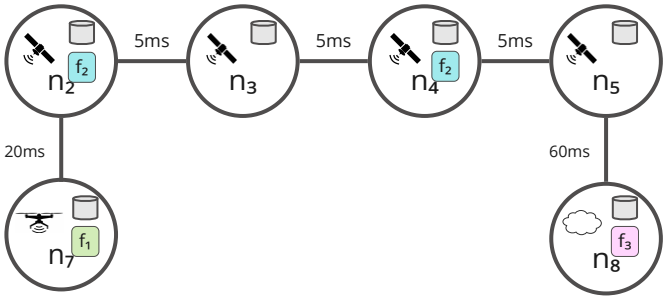
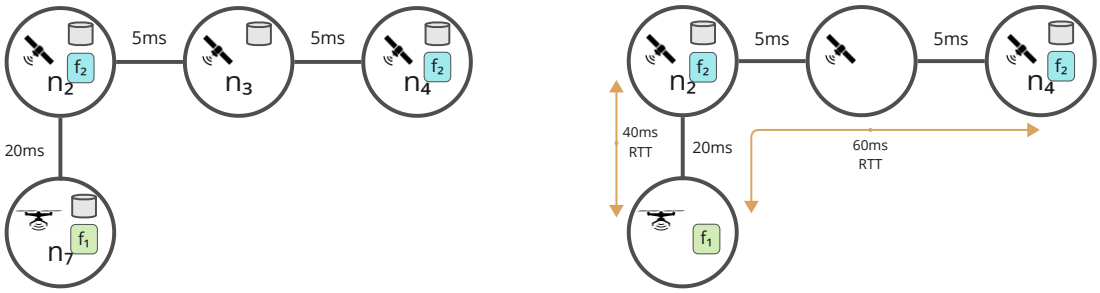


Figure 5.3: Candidate nodes after calculating the shortest path from Head Node to Tail Node in phase ① INITIALIZE

By reducing the solution space to only include nodes between Head Node and Tail Node as depicted in Figure 5.3, we optimize state transitions not in a narrow view reduced to two subsequent functions, but in a holistic view considering the complete workflow. Calculating the shortest path between Head Node and Tail Node concludes phase ①.

Figure 5.4 depicts the filtering of candidate nodes based on SLOs and warm function availability in phase ②. Figure 5.4a shows the resulting SLO-compliant nodes. n_5 would violate the SLO since the RTT from n_5 is $70ms$ and is, therefore, not included in this set. From the four remaining nodes (n_2, n_3, n_4, n_7), two have the Target Function f_2 available in the warm queue, reducing the pool of candidate nodes to n_2 and n_4 as shown in Figure 5.4b, concluding phase ②. In phase ③, the Target Storage is elected. Both are viable in the narrow view regarding SLOs and function availability. Arguably, n_2 is the best pick regarding the migration time. However, the holistic view favors



(a) Nodes fulfilling SLOs

(b) Nodes with warm Target Function

Figure 5.4: Candidate nodes after SLO and warm function filtering in phase ② IDENTIFY

propagating the state further towards the Tail Node since, at the end of the workflow, the resource-intensive functions need the state in this vicinity. Therefore, the storage instance hosted on n_4 is elected as Target Storage by the Target Storage Elector and returned to f_1, concluding phase ③. Next, in phase ④, f_1 generates a SkylarkKey based on the workflow, function, and storage and saves the output state to the Target Storage on n_4. The SkylarkKey is returned to the workflow client, concluding phase ④ and, therefore, this example.

Note that we made two simplifications for this example: First, Figure 5.2 simplifies the actual node topology since the network size in a 3D Continuum cluster may include thousands of nodes in a real-life scenario. Figure 5.2 hints at this fact with the dotted lines next to the space nodes. The vast number of nodes in the network underlines the need to reduce the number of Target Storage candidates. Additionally, depending on how tight the SLOs are set, most nodes would exceed T_{max}^{RTT}. Second, we only consider one state transition and omit the dynamic nature of satellite nodes floating above the stationary terrestrial edge and cloud nodes. When f_2 finishes its computation and performs State Propagation, the node topology might have changed compared to when the Skylark Elect Service picked the Target Storage for the output state of f_1. The Skylark Model treats the second state transition as a new calculation, performing all phases described above, utilizing the Skylark Elect Service deployed on the new Head Node n_4.

Algorithm 5.1 depicts the steps the Target Storage Elector applies for a single state transition. It relies on several inputs: The *nodeGraph* holds the information about the current node topology and its network characteristics. The *targetFunction* is the successor function and recipient of the state transition. The maximum network RTT SLO T_{max}^{RTT} limits the distance to the Target Storages and helps align with SLO-aware scheduler decisions. The Head Node n_H hosts the function initiating the state transition. The Tail Node n_T is the likely execution environment of the last function of the workflow function sequence.
In line 4, the shortest path calculation of phase ① is performed by employing Dijstra's algorithm [17] between n_H and n_T with the RTT between nodes used as edge weights. In lines 6-13, the resulting candidate nodes are filtered for SLO compliance and function

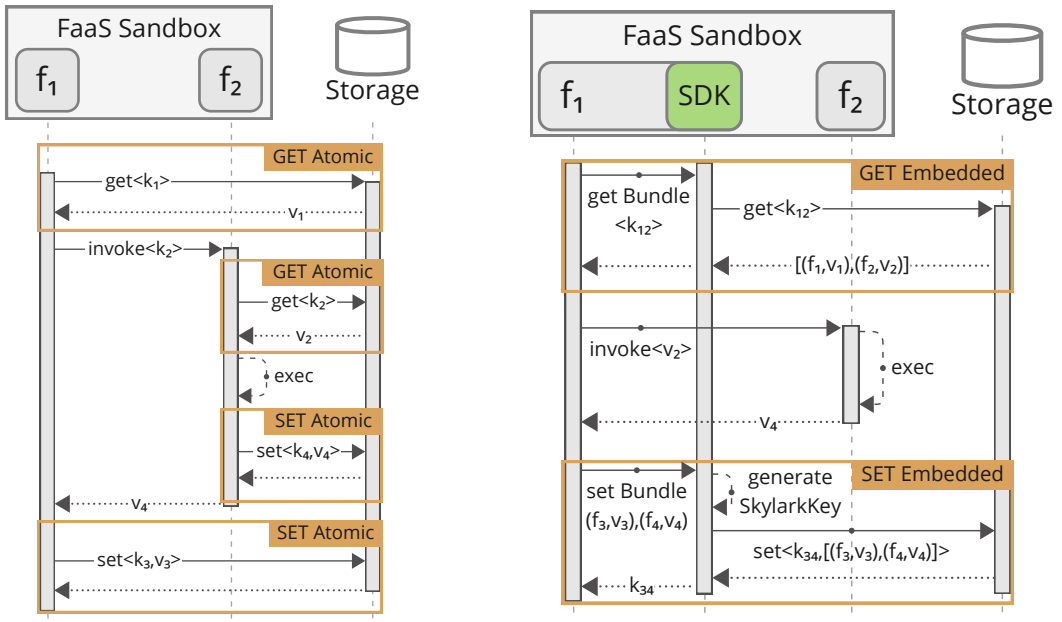
Algorithm 5.1: Target Storage Election Algorithm

```
   Input: nodeGraph  // Vertex:Node, Edge:RTT
   Input: targetFunction
   Input: T_max^RTT  // SLO
   Input: n_H  // Head Node
   Input: n_T  // Tail Node
   Output: s_E  // Elected Target Storage
 1 filteredSLO = new Stack()
 2 s_E = null
 3 // Reduce solution space
 4 storageCandidates = dijkstra(nodeGraph, n_H, n_T)
 5 // Apply SLO filter and warm function filter
 6 for each candidateNode, t^RTT in storageCandidates do
 7 |   if t^RTT ≤ T_max^RTT then
 8 |   |   filteredSLO.push(candidateNode.storage)
 9 |   |   if targetFunction in candidateNode.warmFunctions then
10 |   |   |   s_E = candidateNode.storage
11 |   |   end
12 |   end
13 end
14 // Pick Target Storage
15 if s_E == null then
16 |   s_E = filteredNodesSLO.pop()
17 end
18 return s_E
```

availability as per phase ②. Each node fulfilling the SLO is pushed on a stack. If it also hosts the Target Function in the warm queue, the nodes' storage instance is temporarily assigned as the elected Target Storage s_E. If a node later in the path satisfies both conditions, s_E is overwritten, ensuring that viable Target Storage is selected while moving the state toward the workflow goal vicinity. Lines 15-18 depict phase ③, where the Target Storage is elected. If s_E has already been set inside the last block, it is returned as is. The other case indicates that no node complying with SLO requirements has the Target Function in the warm queue. Therefore, the storage hosted by the SLO-compliant node furthest toward n_T is elected and returned.

5.2 State Bundling Mechanism

The bundling mechanism leverages the Skylark SDK to enable Embedded Functions. These functions optimize state management by reducing the number of state operations performed on external storage. We achieve this by providing the developer with interfaces where state operations such as **get**, **set**, and **delete** can be performed atomically for



(a) Embedded Functions performing get and set operations one-by-one

(b) Embedded Functions performing get and set operations using the state bundling mechanism

Figure 5.5: Comparing the number of state operations needed to get and set the state of two Embedded Functions with and without the state bundling mechanism

multiple functions.

Figure 5.5 compares two scenarios where Embedded Functions perform storage operations with and without the Skylark State Bundling mechanism. Both scenarios include two serverless functions embedded into a single sandboxed environment $f_1(f_2) \in F$. Both functions rely on the input state to perform their task and produce the output state when their computation finishes. Since serverless functions are stateless, the state is stored externally. A part of f_1's execution depends on the function output state of f_2.

Figure 5.5a depicts the baseline scenario where both functions perform get and set storage operations independently. First, f_1 fetches its state $D_A^{f_1}$ from the external storage using key k_1 and then invokes f_2 since it needs the output for its computation. f_2 also performs a storage get operation using its key k_2 to retrieve the input state value v_2. After f_2 finishes its tasks, the output state $D_A^{f_2} = \{k_4, v_4\}$ is saved using a set operation to the external storage. The state value v_4 is returned to f_1, which can now finish its task. In the same fashion as f_2, f_1 saves it output state $D_A^{f_1} = \{k_3, v_3\}$ using a set operation and terminates. In total, 4 state operations have been performed.

By contrast, when using the Skylark State Bundling mechanism (Figure 5.5b), the two functions can perform the same work with only 2 storage operations. f_1 as the parent function leverages the Skylark SDK to treat the function state of both functions as a

single entity. Using the key for the Embedded State k_1, the SDK fetches the input state for f_1 and f_2 with a single storage get operation. Next, f_1 invokes f_2 and directly supplies its input state as part of the request, therefore allowing f_2 to start task execution immediately. After f_2 finishes its task, the output state value v_4 is returned to f_1, which can now finish its task. Finally, f_1 again uses the SDK to create an Embedded State object for both functions' output state $D_E^{f_1} = \{k_2, [(f_1, v_3), (f_2, v_4)]\}$. The SDK accepts the function output states as tuples of `FunctionID` and state value (k_F, v) and aggregates them into a state bundle. A SkylarkKey is created to identify the bundle, and finally, the Embedded State is stored in the external storage with a single state operation. In total, 2 state operations have been performed.

Using the baseline method, the number of storage operations grows with a factor of n, where n is the number of functions embedded in a sandbox $|f|$. Employing the Skylark State Bundling mechanism results in constant storage operations regardless of $|f|$, therefore minimizing storage operation overhead.

Algorithm 5.2: State Bundling Algorithm

```
   Input: D  // State Tuples [(k_F, v)]
   Input: targetStorage  // StorageID k_S
   Input: fnId  // FunctionID k_F of parent
   Input: chainId  // Optional ChainID k_C
   Output: k  // SkylarkKey
 1 D_E= new EmbeddedState()
 2 if chainId == null then
 3 |  chainId = generateNewId()
 4 end
 5 k = new SkylarkKey(chainId, fnId, targetStorage)
 6 D_E.setKey(k)
 7 // Map Key-Value pairs to State object
 8 for each k_f, v in D do
 9 |  D_E.setField(k_f, v)
10 end
11 // migrate state to Target Storage
12 storageClient.setEmbedded(D_E, k.StorageID)
13 return k
```

Algorithm 5.2 depicts the state bundling mechanism for the case in which Embedded Functions want to store multiple state values to a Target Storage with a single state operation from the perspective of the Skylark SDK. It expects multiple inputs: D is an Array of state tuples holding the actual state of the functions and their corresponding identifiers. The *targetStorage* indicates the storage instance on which the state will be stored. This can be local storage or remote storage on another node in the cluster. The *fnId* identifies the parent function using the SDK. The optional *chainId* identifies the workflow instance.

In lines 2-4, a new *chainId* is generated if none was provided. In line 5, a SkylarkKey is created, composed of k_C, k_F, k_S, and assigned as key for the Embedded State in line 6. In lines 8-10, the state tuples provided by the parent functions are mapped to the Embedded State object D_E. In line 12, the Embedded State is saved to the Target Storage, and the new SkylarkKey k is returned to the parent function in line 13.

CHAPTER 6

Prototype Implementation

This Chapter describes the implementation process and technologies used for the Skylark Framework. We realized Skylark Elect Service as a stand-alone Rust [44] service. Skylark SDK is realized as a Rust Library. Local storage instances are provided using Redis key-value stores. We also provide an overview of the GitHub repository [21] structure and conclude with a description of the challenges we faced during development and how we addressed them.

6.1 Environment and Tech Stack

Programming Language: Rust [44]. Reducing serverless workflow overhead is one of the primary objectives of our mechanisms. By choosing a low-level programming language, we avoid introducing language-related performance overhead. Rust offers native memory safety and powerful async capabilities compared to languages like C or C++. It has out-of-the-box support for compiling to Wasm targets. Table 6.1 shows the key Rust libraries we used. See Section 6.2 for a detailed description.

WebAssenbly Runtime: WasmEdge [54]. Wasm applications integrate well into heterogeneous edge environments since they are portable across different platform ar-

Name	Used by	Description
reqwest	Elect, SDK	HTTP communication as client
tokio	Elect, SDK	Asynchronous operations and concurrency
hyper	Elect	HTTP communication and exposing an API
serde	Elect, SDK	Serializing and de-serializing data to/from internal models
redis	SDK	Storage operations as client

Table 6.1: Key Rust Libraries used for the Skylark Prototype

chitectures. Its sandboxed execution environment offers additional safety bounds on a multi-tenant, serverless platform. Wasm binaries have smaller footprints and faster start-up times than traditional containers. WasmEdge supports Rust out of the box and integrates well with Kubernetes through the Kwasm add-on.
Containerization: Docker [18]. We used Dockerfiles to build and containerize the Wasm binary in two steps:

1. *Build*: The rust image for the `wasi/wasm` platform is used as build environment. Additionally, `clang` is installed. The cargo target `wasm32-wasmp1` is added, and the `RUSTFLAGS` `wasmedge` and `tokio_unstable` are set.

2. *Containerize*: The second step creates a minimal `scratch` image, places the compiled binary in the root folder, and marks it as `ENTRYPOINT`.

We published images to DockerHub [19] via `docker push`.
Orchestrator: Microk8s [14] There are several Kubernetes forks suitable for resource-constrained environments, such as the LEO edge (OpenFaaS [26], k3s [37]). We chose Microk8s for its high compatibility with the chosen OS (Ubuntu Server 24.04.1 LTS) running on our hardware (Raspberry Pis). However, our prototype is platform-agnostic.
Serverless Framework: Knative [23]. State-of-the-art framework and high compatibility with the remaining tech stack. It was added using the Microk8s Knative plugin.
Storage: Redis [42]. For data storage, we chose a fast and lightweight key-value store. Redis allows for simple String key-value pairs and more sophisticated storage types like HashMaps. This is important since we use HashMaps to model Embedded State for Embedded Functions. We deploy one Redis instance via DaemonSet to enable local storage on each node.

Next, we describe the implementation workflow of the prototype from development to deployment:

1. **Develop**: Create software artifact using the Rust programming language.

2. **Build**: Calling `docker buildx build` compiles the code and creates a Docker image.

3. **Push**: We publish the Docker Image to the DockerHub container registry via `docker push`.

4. **Deploy**: The Image is deployed via `kubectl apply` using a YAML script. Skylark Elect Service is deployed as DaemonSet across all cluster nodes. Skylark SDK is deployed as part of a serverless function as a Knative Service.

The prototype requires setting the environment variables shown in Table 6.2.

Variable	Used by	Description
LOCAL_NODE_HOST	Elect, SDK	IP of the local node
NODE_INFO_PORT	Elect	Port of the local NodeInfo service for topology data
NODE_REFRESH_INTERVAL	Elect	Refresh interval of the Topology Handler
SKYLARK_ELECT_PORT	SDK	Port of the local Skylark Elect Service

Table 6.2: Environment variables accessed by the prototype

6.2 Tools and Libraries

Here, we describe the tools and libraries used by the prototype.

Threads

Skylark Elect Service initializes on start-up and spawns two threads: `Node Topology Handler` and `HTTP Client Handler`. We use the `tokio` crate and utilize its thread tooling.

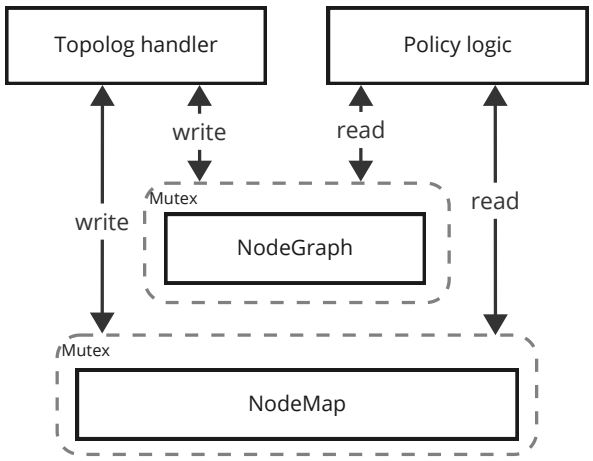


Figure 6.1: Skylark Elect Service threads and shared data

We use Mutex guards provided by the Rust `std::sync` library to restrict concurrent access to the shared objects.

SkylarkKey generation

We use the `uuid` library to generate `ChainID` and `FunctionID`.

Figure 6.2: Anatomy of a SkylarkKey

Figure 6.2 provides an example of a serialized SkylarkKey, where colons separate the three parts.

(De-)Serialization

Using HTTP and Redis clients requires serializing and de-serializing data, which is moved across system layers. We used the JSON standard for string conversion. The Rust libraries `serde` and `serde_json` were leveraged to enable this functionality via `#[derive()]`. The SkylarkKey is also converted to and from a String by leveraging `serde`. This allows developers to use the String type when interacting with the Skylark SDK.

HTTP Client and Server

We used the `reqwest` library to perform HTTP requests and the `hyper` library to serve the HTTP GET endpoint `/storage-node` in the Skylark Elect Service.

Redis Client

The Redis Client interfaces with the Skylark SDK State module and the cluster's Redis instances. It exposes functionality for reading, writing, and deleting Embedded State and Atomic State. Redis Hashes store the former, and regular Key-Value pairs store the latter.

6.3 GitHub Repository

This section describes the repository structure and links the relevant packages. Each package has a `README.md` file in its root folder with a short description and instructions for building and deployment. Link to the repository: https://github.com/polaris-slo-cloud/skylark

- `skylark_elect`: Skylark Elect Service implementation
- `skylark_sdk`: Skylark SDK implementation

- `experiments`: Includes scripts for setup and execution and introductions for the topology simulation.
- `examples`: This directory includes example functions implementing the Skylark SDK and the NodeInfo service.

6.4 Challenges and Solutions

This Section describes two challenges we faced during development and how we solved them.

6.4.1 Serverless and Networking

Challenge. As our work aims to embrace the serverless paradigm, the first version of the Skylark Elect Service was deployed as a Knative service. This introduced two problems: First, calling the service while in the cold queue caused significant overhead. Second, service discoverability from outside the Knative environment was a challenge.
Solution. Deploying Skylark Elect Service as a Daemonset across nodes. This prevents scale-to-zero without paying a high price in computing resources since the memory footprint of the Wasm image is tiny compared to images using standard target platforms (Linux, Windows). Networking becomes trivial since using `hostNetwork: true` in the deployment spec enables client services to reach the Skylark Elect Service via the node IP and a static port.

6.4.2 Separation of concerns

Challenge. The second version of the Skylark Elect Service took the whole state as input and took care of node election and State Propagation. In this version, the SkylarkKey didn't encode the Target Storage. This led to high coupling with the client and made Skylark Elect Service a bottleneck for parallel executions. Also, a non-ephemeral state must be stored to track state locations. **Solution.** Separate the concerns of node election and state operations. Skylark SDK executes state operations and calls the Skylark Elect Service only with necessary information. This further allows for flexibility in Skylark Elect Service's deployment. The service can be deployed on multiple nodes, but it could also be deployed as a centralized service.

CHAPTER 7

Evaluation

This Chapter evaluates the Skylark model and its mechanisms for State Propagation and State Bundling within the 3D Continuum. Our experimental cluster consists of 8 nodes, simulating dynamic network conditions using latency and bandwidth control tools. We conducted experiments to assess i) workflow State Propagation performance (Section 7.2), ii) target election algorithm scalability (Section 7.3), and iii) workflow scalability (Section 7.4). For State Bundling, we iv) conducted a performance experiment (Section 7.5). We conclude the Chapter by discussing the findings in Section 7.6. Results indicate that Skylark's State Propagation mechanism reduces workflow latency by up to 33% and state retrieval time by up to 66% compared to baseline approaches while increasing workflow throughput by up to 91%. The State Bundling mechanism decreases function latency by up to 24% by minimizing storage operations. These findings demonstrate the effectiveness of Skylark in optimizing stateful serverless execution across the 3D Continuum.

7.1 General Experimental Setup

7.1.1 Infrastructure

All experiments were conducted on a cluster of 8 nodes, divided into 1 cloud node and 7 satellite nodes.

Hardware. The cloud node and 3 satellite nodes are Raspberry Pi (RPI) 5 Model B Rev 1.0 (8GB RAM, 4x2.4GHz CPU), 3 satellite nodes are RPI 4 Model B Rev 1.4 (8GB RAM, 4x1.8GHz CPU), and 1 satellite node is a RPI 4 Model B Rev 1.1 (2GB RAM, 4x1.5GHz CPU). All devices have a WiFi chip connected to a consumer-grade Router (Zyxel DX-3101-BO) supporting 5GHz/10Gbit. Figure7.1 shows the Raspberry setup during the experiment. To help the RPIs with passive heat syncs with heat dissipation, 2 12V ventilators were installed on a cartage contraption to produce air circulation. The

fans drew power directly from 2 RPI's General Purpose Input/Output pins. Though these only supply 5V, the resulting air current was sufficient to keep core temperatures below 60°C throughout the experiments.

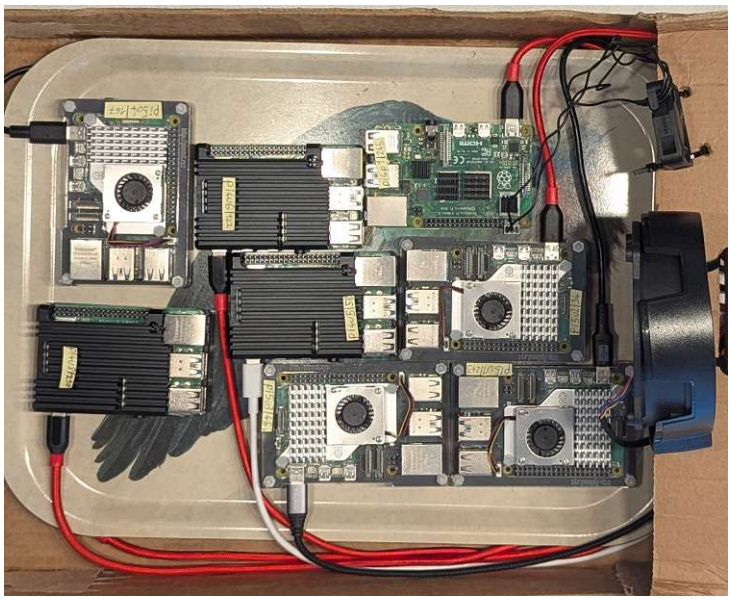


Figure 7.1: Testbed setup with 8 Raspberry Pis

Software. The experimental functions are written in Rust v1.83.0 with Cargo edition 2021 (using Skylark SDK as a dependency) and compiled with WasmEdge, using the `wasm32-wasip1` target. Docker-Desktop, enabled with the containerd image store, was used to store/push Wasm images on/to DockerHub. We chose Knative Serving v1.8.1 as a serverless platform to pull, serve, and scale our serverless functions. The Node-Info-Service, simulating an orchestrator API providing up-to-date node topology data, was written in Python (Docker image target: `python:3.9-slim linux/arm64`). All nodes run the Ubuntu Server 24.04.1 LTS operating system (OS). We chose Microk8s v1.31.3 as the orchestrator, a lightweight Kubernetes fork that integrates well with Ubuntu.

Networking. We used the `tc` tool to control latency (netem delay) and bandwidth (htb rate) between nodes. We set the simulation parameters based on similar work on Edge-Cloud-Space communication [33]. Satellite-to-satellite latency was uniformly sampled from [1,20] ms and [45,75] for satellite-to-cloud communication. Using cronjobs to simulate a changing network topology, we change the `tc` configuration over time. Cronjobs allow for testing different topology-change periods.

Functions

The functions are deployed as Knative services. To enable scheduler simulation, we deploy node-specific functions, increase the scale-down time to 10 minutes, and discount the first run to remove bias from cold starts. Each function uses the Skylark SDK as a dependency for storing and retrieving state data. They expose an HTTP endpoint for function invocation and output their result as an HTTP Response. All functions are written in Rust and compiled as Wasm binaries.
Each function takes a SkylarkKey as input, retrieves the predecessor output state, does some computation, and stores a new output state of the same size via Skylark SDK. They return experiment metrics and the new SkylarkKey.

7.1.2 Storage

Our experiments use Redis as the key-value store to migrate and retrieve data. Each node in our cluster has a local Redis (community edition) instance running inside a Microk8s pod. We developed and used a helper function to store values of different but specific sizes. It is written in Rust and exposes an HTTP endpoint, which expects the target node and state size as input. The function generates a random string of this size and stores it in the specified Redis instance.

7.1.3 Scripts

We use shell scripts to prepare and execute the experiments.

State Propagation

- `setup_propagate_performance.sh`: generates data with sizes 1MB-50MB and stores it in the given storage.
- `setup_propagate_scalability.sh` generates a given amount of key-value pairs with fixed size and stores them in the given storage.
- `run_propagate_performance.sh`: Runs performance experiment.
- `run_propagate_scalability.sh`: Runs scalability experiment.
- `run_propagate_workflow.sh`: Runs a single workflow. It is called by `run_propagate_scalability.sh`.

State Bundling

- `setup_bundled.sh`: Generates and stores data for atomic and Embedded Functions of depth 2-5 in the given local and global storage.
- `run_bundled.sh`: Runs the performance experiment.

Name	Node-Type	Description
Preprocess	All	Gets the path to the raw image as input, opens it, and performs processing operation, stores data, returns key.
Filter	All	Gets key from Preprocess as input, loads data, performs filter operation, stores data, returns key.
Detect	All	Gets key from Filter as input, loads data, performs detection operation, stores data, returns key.
Alarm	Cloud	Gets key from Detect as input, loads data, performs resource-intensive inference tasks, returns result.

Table 7.1: Serverless functions used in the workflow based on our EO use case

Baseline	Storage location	Description
Stateless	Cloud centralized	Workflow functions store and retrieve state from a centralized global storage
Random	Local on all nodes	Picks a random node from the initial path as Target Node for each state transition

Table 7.2: Experiment Baselines

7.2 Experiment: State Propagation Workflow Performance

This experiment evaluates the performance of the Skylark State Propagation mechanism against two baseline policies in a serverless workflow comprised of 4 chained functions. Table 7.1 shows the example functions we implemented for the experiment, inspired by the use case scenario outlined in Chapter 1.

7.2.1 Experiment Definition

Let W be a serverless workflow with 4 chained functions $W = \{[f_1, f_2, f_3, f_4], n_8\}$. Let $n_{1..7} \in N_S$ be satellite nodes, and $n_8 \in N_C$ a cloud node holding f_4 in the warm queue and, therefore, chosen as Tail Node. The experiment is conducted and repeated with increasing state sizes (5MB,10MB,...,50MB). The client invoking W is located near n_1 and acts as a workflow handler. All functions utilize Skylark SDK to store the output state and retrieve the input state. The function finally returns the SkylarkKey to the client. The RTT SLO is set to $60ms$. We evaluate Skylark against 2 baselines defined in Table 7.2.

In each test run, we gather the following metrics:

- Workflow latency T_W: The total time of a workflow execution from the time f_1 is invoked until f_4 returns its output.
- State migration time $T_f^{mig(v)}$: The time it takes f to store output state value v.

- State retrieval time $T_f^{ret(v)}$: The time it takes f to retrieve input state value v.
- State retrieval distance d_{n_v,n_f}: The distance between the node hosting the Target Storage n_v of input state value v and the actual node executing f (0 if the state is locally available).
- Local state availability L_f: Binary value depicting whether the input state was available locally. We calculate the arithmetic mean over all runs to find the average local state availability in %.

Note that f_4 is scheduled on n_8 in all scenarios, simulating a node-type specific function as per our use case, demanding a cloud environment for resource-intensive tasks. Therefore, we don't consider d_{n_k,n_f} and L_f for this function when evaluating the workflow performance. The experiment was repeated 10 times per policy and state size to ensure consistency and avoid bias.

7.2.2 Experiment Setup

We set up the infrastructure and networking simulation described in Section 7.1.1. In this experiment, nodes $n_{1..3}, n_8$ are RPI 5, $n_{4..6}$ are RPI 4 with 8GB RAM, and n_7 is a RPI 4 with 2GB RAM. All Redis instances, excluding n_1, are cleared before each workflow execution. We wrote a shell script simulating the client and scheduler, calling the functions defined above. The shell script also performs the timing for T_W. The example functions supply $d_{n_k,n_f}, T_f^{mig(v)}, T_f^{ret(v)}, L_f$ as part of their output. The shell script outputs all metrics for each experimental run to a log file.

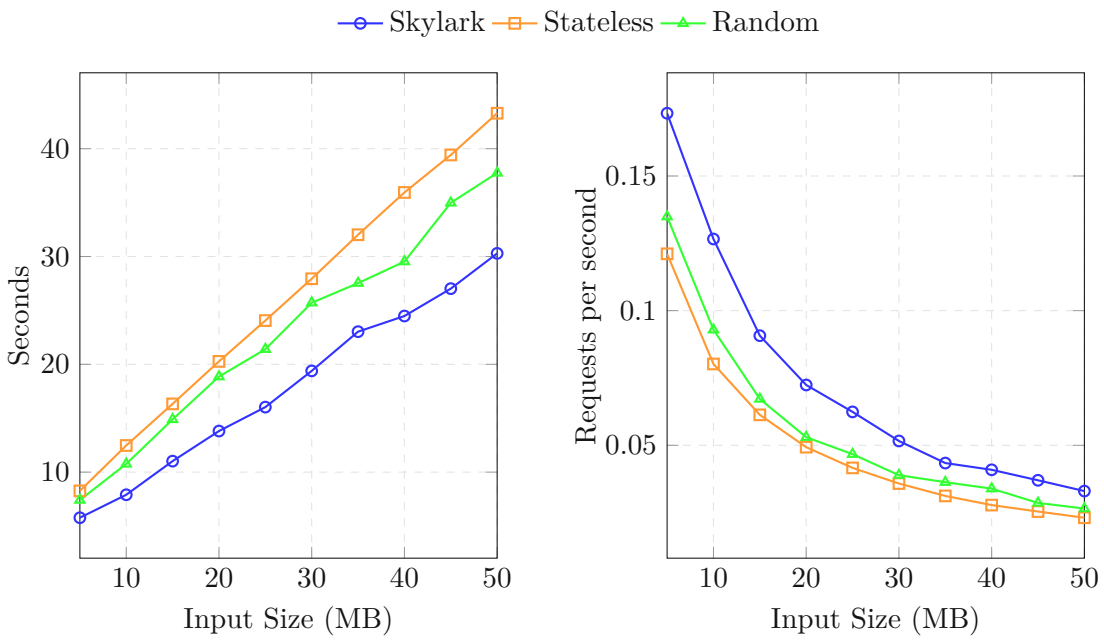
NodeInfo Setup

This Python pod is an orchestrator API wrapper that exposes factual information via HTTP endpoints, such as node metadata and simulated node topology information. It loads the simulated topology via a JSON file and changes the inter-node latencies within the defined deviation ranges. The service queries the Kubernetes API for node specifications, such as name and IPs. It is deployed on each node as a Daemonset.

7.2.3 Experiment Result

Figure 7.2 reports the workflow latency and throughput result of the performance experiment for Skylark compared to the Stateless and Random baselines. Figure 7.2a shows the average workflow execution time T_W on the y-axis and the state size on the x-axis. On average, Skylark decreases workflow latency by 33% compared to Stateless and by 22% compared to Random. Figure 7.2b reports on the throughput. Skylark has a consistently higher throughput than the other policies. Skylark improves throughput by 50% compared to Stateless and 29% compared to Random.

We hypothesized that propagating the output state close to its Target Function's local execution environment reduces workflow latency. Figure 7.3 reveals the average read



(a) Latency **(b) Throughput**

Figure 7.2: Total Workflow Execution Time

($T_f^{ret(v)}$) and write ($T_f^{mig(v)}$) times (y-axis) of workflow functions of varying input sizes (x-axis). Figure 7.3a plots $T_f^{ret(v)}$ and shows a significant decrease for Skylark compared to Stateless (66%) and Random (62%). However, as shown in Figure 7.3b, Skylarks average $T_f^{mig(v)}$ is 9% lower than Stateless and 4% higher than Random. We conclude that Skylark significantly reduces state retrieval times while performing similarly regarding state migration time. Next, we look at local state availability L_f and state proximity d_{n_k,n_f}. Recall that state is locally available if the function can access it through the local storage. Conversely, the state distance is determined by the number of nodes between the function and the state.

The left chart of Figure 7.4 illustrates that, indeed, Skylark has a lower average distance (0.21) between the function execution environment and its state compared to Stateless (4) and Random (2.16). Random leverages local storage on satellite nodes; therefore, the average distance is lower than the stateless approach. On average, Skylark has a local state availability of 79% compared to 12% with the Random policy. We conclude that Skylark increases local state availability and reduces network strain compared to the baselines.

An essential aspect of electing successor nodes for State Propagation is the scheduler's decision. One way of predicting where the scheduler invokes a function is SLO awareness. The RTT SLO is essential to Skylark's election decision. Figure 7.5 reports on the SLO

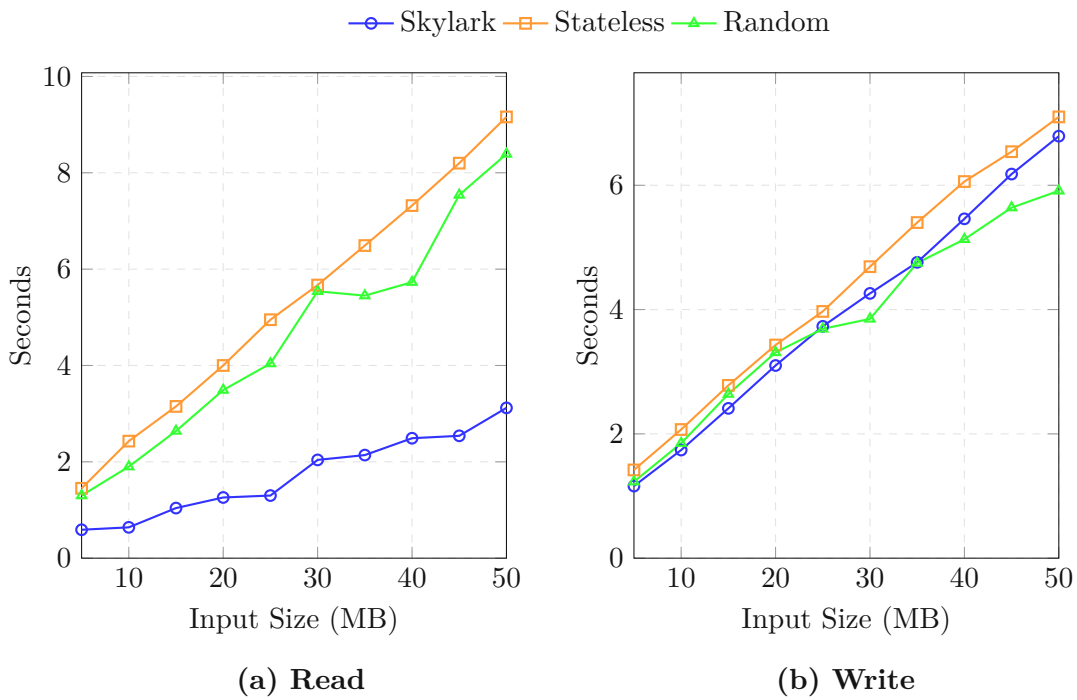


Figure 7.3: Average Retrieval and Migration Times

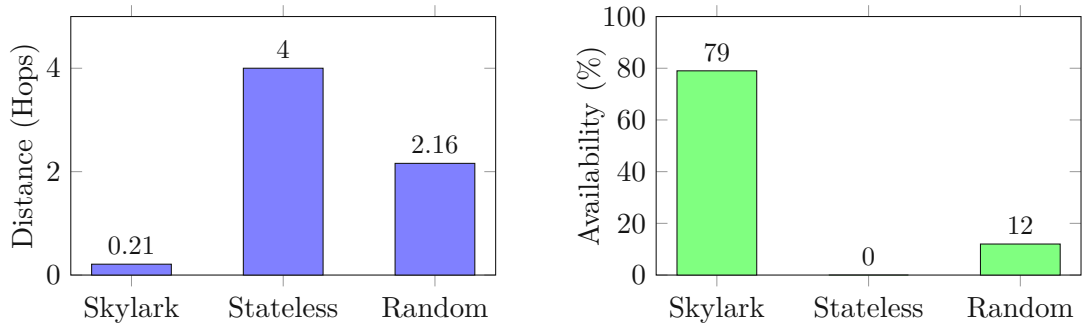


Figure 7.4: Average State Read Distance in Hops and Local State Availability

violations during the experiment. Skylark never violates the SLO since the algorithm prohibits it. The Random policy has a high variation in SLO violations, explained by the nature of picking random nodes in the path. The Stateless setup has many SLO violations due to the high cost of migration and retrieval from the central storage.

Figure 7.6 reports average resource usage over all nodes during experiment execution. Throughout the experiment, both CPU usage and RAM allocation remain stable. Skylark, on average, requires 3.5% more RAM than Stateless.

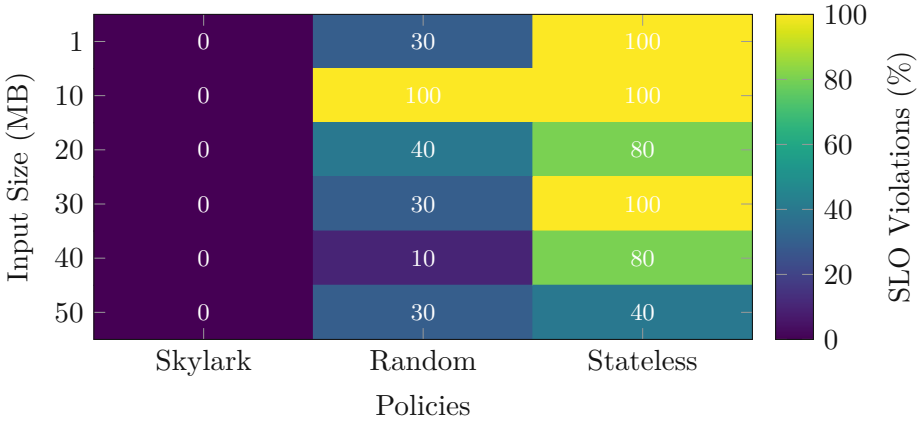
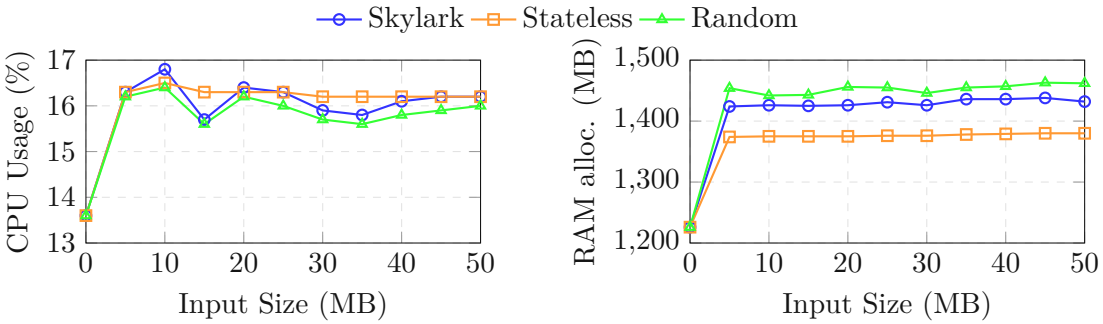


Figure 7.5: Data Migration Time SLO Violation Map



(a) CPU **(b) RAM**

Figure 7.6: Resource Usage during State Propagation Performance Experiment

7.3 Experiment: State Propagation Policy Scalability

LEO satellite networks consist of thousands of satellites. To ensure our model is viable for topology sizes of this scale, we benchmark the implementation of the Skylark policy.

7.3.1 Experiment Definition

Based on multiple input variables, the Skylark policy decides if and where to propagate the state. See Algorithm 5.1 for details. We consider node topologies of size 10, 100, 1,000, and 10,000. The topology size's average policy runtime is the only metric collected in this experiment. The experiment was repeated 15 times for each node topology size.

7.3.2 Experiment Setup

We set up the infrastructure and networking simulation described in Section 7.1.1. We used a RPI 5 node. We implemented a `"/benchmark"` endpoint in our Skylark Elect

Service prototype to eliminate bias from function overhead or invocation latency. It loads the graphs from JSON files and elects the worst-case nodes as Head and Tail Nodes. Timing is done directly in the endpoint function and returned as an HTTP response.

7.3.3 Experiment Result

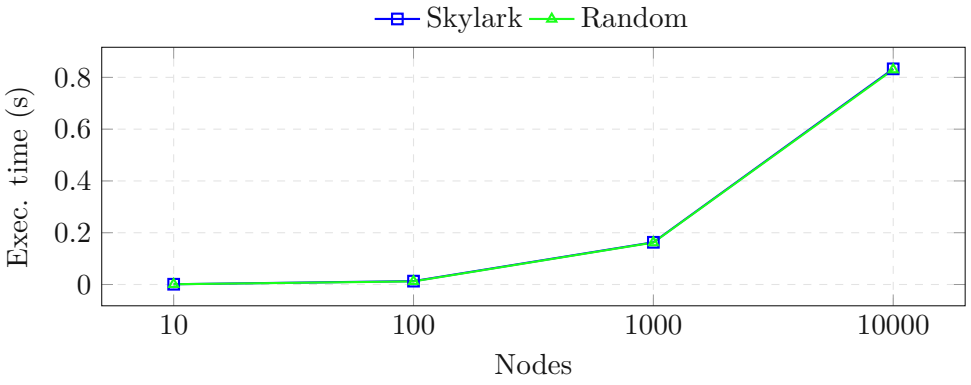


Figure 7.7: Simulation of Skylark state propagation node election for 10-10,000 nodes

Figure 7.7 shows the result of our load test. While with 10 and 100 nodes, the policy runtimes are negligible, the computational complexity of Dijkstra's algorithm $O(V + ElogV)$, where V is the number of vertices and E the edges, is noticeable with topology sizes 1,000 and 10,000. Comparing Skylark to the Random policy shows that SLO awareness adds minimal overhead. Specifically, Skylark is slower than the Random policy on average by 1 ms at $V = 1.000$ and by 3 ms at $V = 10.000$.
We see two options for future work. First, in most use cases, we expect serverless workflows to be geographically restricted. A local and, therefore, smaller graph for policy decisions would make the current implementation feasible. If not, using an alternative data structure, like a Binary Heap with the algorithm, would improve the runtime complexity. Approximation algorithms and distributed solutions could also be potential avenues for optimization.

7.4 Experiment: State Propagation Workflow Scalability

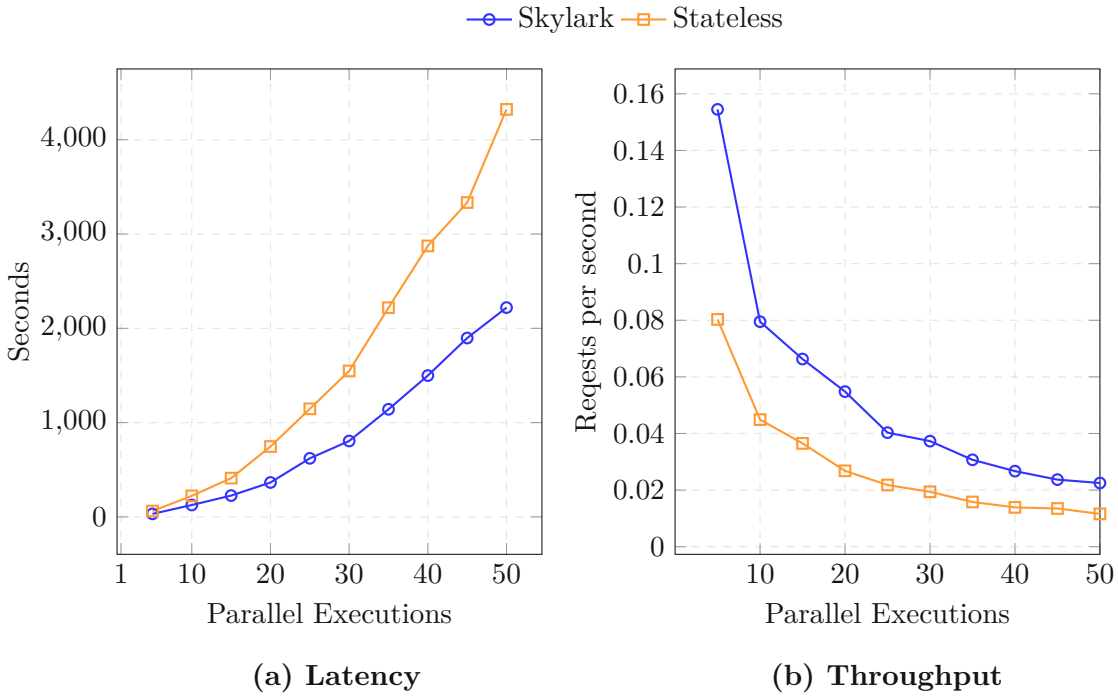
In Section 7.2, we tested how workflow latency and throughput change when varying the input size to determine the performance of Skylark Elect Service. However, we also want to know how these metrics are affected by different numbers of parallel workflow execution requests.

7.4.1 Experiment Definition

The Experiment definition is analogous to the performance experiment (see Section 7.2.1). However, we use a fixed state size of 2MB instead of the varying input sizes. The number

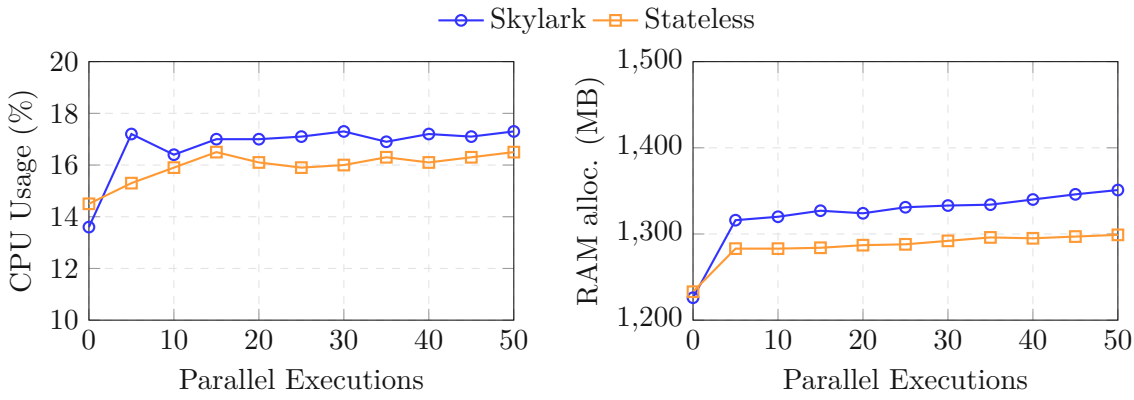
of parallel executions (fan-out degree) ranges from 5 to 50. We measure the workflow latency metric T_W. This is when f_1 is invoked until f_4 returns its output. The experiment was repeated 10 times for each parallel execution step.

7.4.2 Experiment Result



(a) Latency **(b) Throughput**

Figure 7.8: Skylark Elect Scalability with Parallel Workflow Executions



(a) CPU **(b) RAM**

Figure 7.9: Resource Usage during State Propagation Scalability Experiment

Figure 7.8a reports on the workflow latency of the State Propagation scalability experiment, where the x-axis represents the fan-out degree and the y-axis the sum of workflow latencies in seconds. Using the Skylark policy results in slower latency growth when the number of parallel executions increases compared to the Stateless policy. On average, Skylark reduces workflow latency by 47% in comparison with Stateless. In Figure 7.8b, we see throughput as requests per second on the y-axis while representing fan-out degree on the x-axis as before. Skylark achieves higher throughput than Stateless consistently. On average, the Skylark policy improves throughput by 91%. We attribute this to the reduced read time caused by providing a function's input state close to its local execution environment.

Figure 7.9 reports on the resource usage during experiment execution. Throughout the experiment, both CPU usage and RAM allocation remain stable. Skylark, on average, requires 3.1% more RAM and 5% more CPU than Stateless.

7.5 Experiment: State Bundling Performance

This experiment evaluates the performance of our State Bundling mechanism proposed in Section 5.2.

7.5.1 Experiment Definition

We evaluate the performance of the State Bundling mechanism for Embedded Functions $f_E \in F$ sharing a single sandbox with different bundle sizes and storage placements. We compare our mechanism to a baseline where the same functions are placed in separate sandboxes (Atomic) $f_A \in F$. Let $n_S, n_C \in N$ be a satellite and a cloud node and $T^l_{n_S,n_C}$ the network latency between n_S and n_C. We consider embedded and Atomic Functions with depths 2-5. Let $T_f^{total(v)}$ be the total time f takes to retrieve and store state value v, where $|v| = 10MB$ in this experiment. A run with $|f_E| = i, i \in [1..5]$ entails f_E making 1 call to retrieve and 1 call to store state with $|v| = i * 10MB$ and f_A making i calls to retrieve and i calls to store state with $|v| = 10MB$. We summarize the experimental configurations in Table 7.3. We gathered the following metrics during the experiment:

- State migration time $T_f^{mig(v)}$: The time it takes f to store state value v.
- State retrieval time $T_f^{ret(v)}$: The time it takes f to retrieve state value v.
- $T_f^{avg(k)}$: The arithmetic mean of a functions' $T_f^{mig(v)}$ and $T_f^{ret(v)}$.
- Total state latency $T_f^{total(v)}$: The total function execution latency.

$$T_f^O = T_f^{total(v)} - T_f^{ret(v)} - T_f^{mig(v)} \tag{7.1}$$

The experiment was repeated 10 times for each state configuration and function depth.

Configuration	Handle	$\|f\|^{max}$	State Location	State Size	Network Delay
Atomic-Global	ASG	1	n_C	$10MB$	45ms
Atomic-Local	ASL	1	n_S	$10MB$	0ms
Embedded-Global	ESG	5	n_C	$\|f\| * 10MB$	45ms
Embedded-Local	ESL	5	n_S	$\|f\| * 10MB$	0ms

Table 7.3: Serverless functions used in the State Bundling performance experiment

7.5.2 Experiment Setup

We set up the infrastructure and networking simulation described in Section 7.1.1. In this experiment, nodes n_S and n_C are RPI 5. We wrote 3 shell scripts:

- `setup_storage_mechanism.sh`: Takes the state size in bytes as input and stores a generated string of that size to the Redis stores of n_C and n_S for both Atomic and Embedded Functions of depth 1-5. Prints the generated keys to stout. These are pasted into the next script.
- `run_bundled.sh`: Takes $|f|$ as input and calls the functions in all configurations a given number of times. Does timing for $T_f^{total(v)}$ and also fetches $T_f^{mig(v)}$ and $T_f^{ret(v)}$ as return value from the functions. Logs all metrics to stout.

Function Setup

The functions are deployed and configured as described in Section 7.2.2. Two example functions have been implemented for this experiment:

1. **Embedded.** The invocation endpoint takes the Target Storage and SkylarkKey as input. It fetches the Embedded State and stores it as a new value to the same storage instance.
2. **Atomic.** Same as Embedded, but fetches and stores the state individually.

7.5.3 Experiment Result

Figure 7.10 reports on $T_f^{total(v)}$ of varying bundle sizes. We can observe that the function latency delta between Atomic and Embedded configurations grows as $|f|$ increases in both Local and Global state modes. On average, Embedded-Global (ESG) reduces latency by 20% compared to Atomic-Global (ASG). In the Local configuration, Embedded-Local (ESL) improves latency by 19% compared to Atomic-Local (ASL).

Figure 7.11 presents the same results in a line chart. The latency is shown on the y-axis, and the function depth is displayed as input size on the x-axis. State Bundling decreases latency by up to 24% for Global State and by up to 23% for Local State.

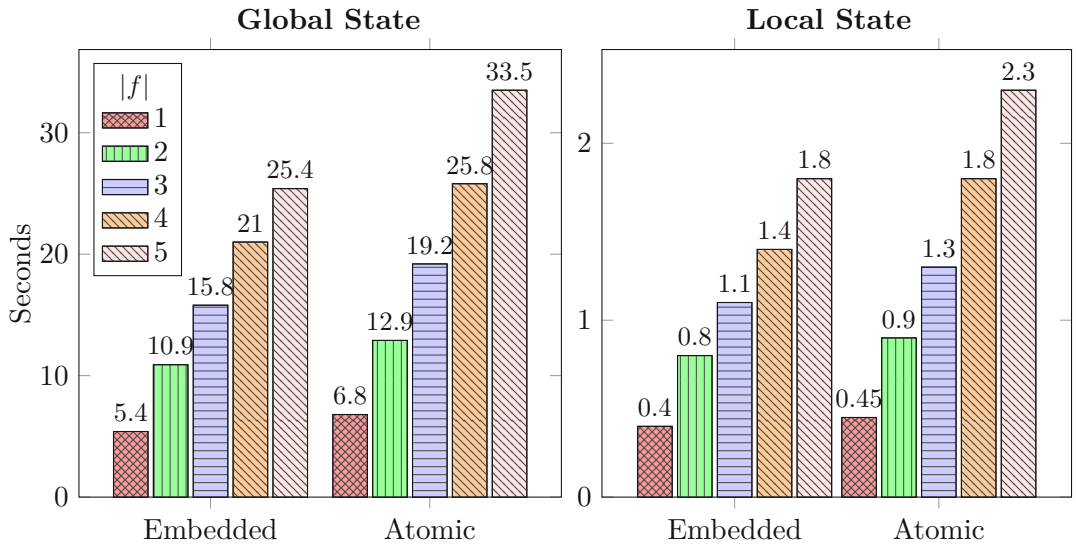


Figure 7.10: Average total Function Latency with varying function depth Embedded vs. Atomic

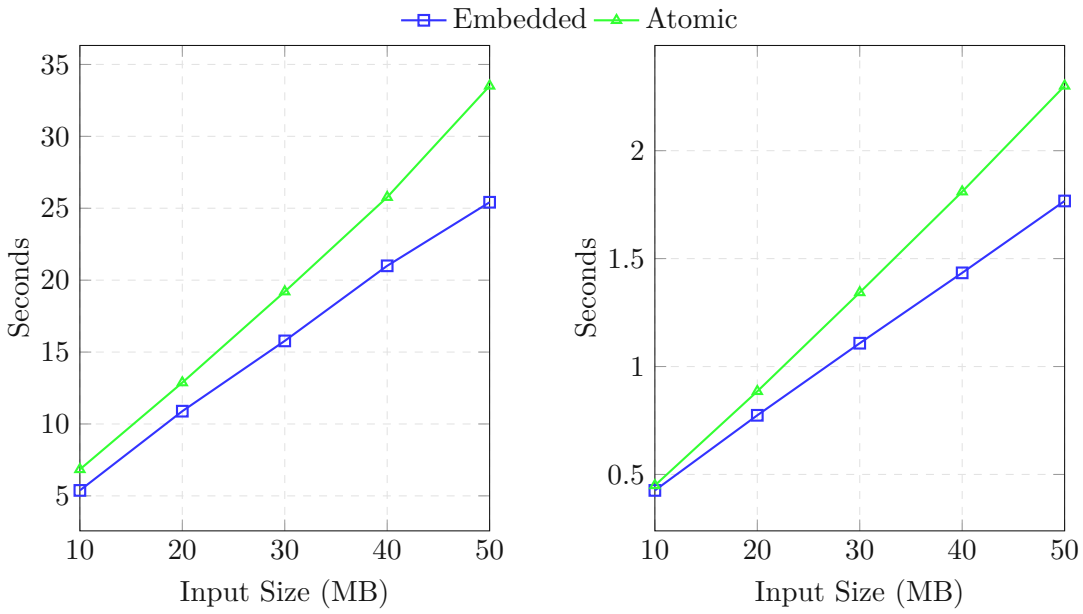
Figure 7.12 compares the networking overhead between the baseline and the State Bundling mechanism. Recall that we defined the overhead by discounting the time used for reading and writing the state (see Equation 7.1). What remains is the overhead introduced by performing the storage operation itself. Due to the additional storage operations the Atomic Function has to perform, the number of functions and overhead increases. Using State Bundling results in a stable overhead since the number of storage operations stays the same regardless of the number of functions.

7.6 Threats to Validity

The experiments highlight the inherent trade-offs in resource-constrained environments such as the 3D Continuum. For instance, while resource usage (CPU and memory) remains stable across experiments, the physical limitations of hardware like Raspberry Pis and the network devices used in the testbed may not reflect the capabilities of more powerful real-world edge nodes or satellites utilizing ISLs. This gap emphasizes the need to evaluate diverse hardware platforms further to generalize the findings.

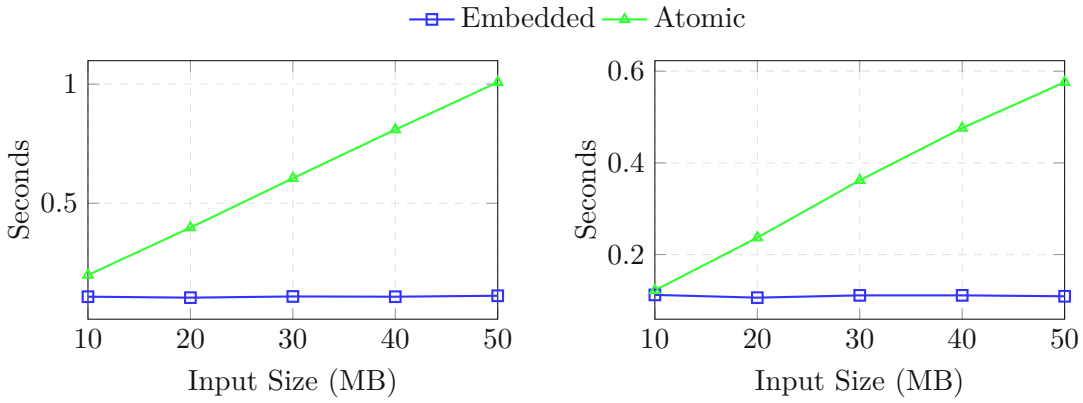
Larger state bundles may decrease scalability while reducing the number of operations. Therefore, it is necessary to develop mechanisms to decide when to embed function states in future work.

Another key insight is the importance of balancing local and global optimizations. While the Skylark mechanisms prioritize local state availability and latency reduction for serverless workflow instances, scenarios with frequent inter-node migrations may require adaptive policies that dynamically weigh local efficiency against global coordination costs.



(a) Global State **(b) Local State**

Figure 7.11: Function Latency Atomic vs. Embedded



(a) Global State **(b) Local State**

Figure 7.12: Function Overhead Atomic vs. Embedded

CHAPTER 8

Conclusion

This thesis introduces Skylark, a serverless framework designed to optimize function state management for serverless workflows in the 3D Continuum. The dynamic nature of this Continuum, characterized by heterogeneous resources and changing network topology, introduces significant challenges for stateful serverless functions. Traditional approaches rely on centralized storage, leading to increased workflow latency and inefficient State Propagation. Skylark addresses these challenges through a topology- and SLO-aware model that improves local state availability and reduces the overhead of state operations.

The Skylark Model introduces a structured approach to stateful serverless workflow execution, integrating two key components to aid in function-to-function state transitions: the Skylark Elect Service and the Skylark SDK. These components work together to dynamically elect optimal storage locations for function input and output state while providing developers with an interface to manage function state efficiently. By providing local storage on edge and space nodes, Skylark minimizes reliance on distant, high-latency, and centralized cloud storage, ensuring that the function output state is available close to the execution environment of the function dependent on it.

Two state management mechanisms were introduced, which leverage the Skylark Model to decrease workflow latency. The State Propagation mechanism optimizes state placement by dynamically electing successor storage locations based on network topology, SLOs, and function readiness, reducing retrieval latency and improving data locality. The State Bundling mechanism groups the state of co-located functions into a single entity, reducing redundant storage operations and execution overhead.

We developed a prototype of Skylark and evaluated it using a wildfire detection use case. Experimental results demonstrated that Skylark's State Propagation mechanism reduced workflow latency by up to 33% and state retrieval time by up to 66% compared to traditional approaches relying on centralized cloud storage or random state placement while increasing throughput by up to 91%. Additionally, the state bundling mechanism

reduced function execution latency by up to 24%, showing its effectiveness in minimizing state management overhead. Skylark enhances the feasibility of deploying stateful serverless workflows in highly dynamic environments by integrating node topology and SLO-aware storage selection, proactive State Propagation, and state bundling. These improvements make serverless computing viable beyond the static terrestrial edge and cloud, supporting efficient function execution in the 3D Continuum.

The main contributions of this work are: (1) the Skylark Model, which provides a structured framework for managing function state across cloud, edge, and satellite environments, (2) the Skylark Elect Service, a topology- and SLO-aware component that dynamically selects the optimal storage location for function output state, improving data locality and thereby retrieval delays, (3) the Skylark SDK, which enables seamless interaction with function state, allowing developers to efficiently store, retrieve, and migrate Embedded State within serverless workflows, (4) the State Propagation mechanism, which proactively migrates function output state to nodes where it is most likely to be accessed next by the successor function, minimizing network overhead and reducing workflow execution time, and (5) the State Bundling mechanism, which aggregates the state of co-located functions into a single entity, reducing redundant storage operations and improving execution efficiency.

8.1 Research Questions

In this section, we revisit and answer the research questions based on our findings.

- **RQ1: How can a serverless computing model be designed to support stateful function execution inside serverless workflows across the 3D Continuum?** To address this question, this thesis introduced Skylark, a topology- and SLO-aware serverless framework that provides local storage on edge and space nodes while dynamically managing function input and output state. The Skylark Elect Service selects optimal state storage locations based on network conditions and function availability constraints, ensuring the state is close to where it will most likely be accessed next. The Skylark SDK further provides an interface for functions to store, retrieve, and migrate state seamlessly. Evaluation results demonstrated that the model successfully reduces state retrieval latency and optimizes function execution across a dynamic and heterogeneous computing fabric.

- **RQ2: To what extent can reducing the overhead of state operations improve the efficiency of co-located serverless functions?** To answer this question, the State Bundling mechanism was introduced, allowing multiple co-located functions to manage their state as a single entity, which minimizes redundant storage operations. Experimental results demonstrated that function execution latency decreased by up to 24% when state bundling was applied, confirming that reducing the number of independent state operations improves execution efficiency.

- **RQ3: To what extent can increasing local state availability and reducing state retrieval latency improve the performance of serverless workflows?** This question was addressed through the State Propagation mechanism, which dynamically places the function output state close to the execution environment of the successor function, minimizing delays in state retrieval by increasing local state availability. Experimental results showed that workflow latency was reduced by up to 33% and state retrieval time by up to 66% compared to a stateless approach that relies on centralized storage. Furthermore, Skylark improves workflow throughput by up to 91%. These findings confirm that improving local state availability improves overall workflow efficiency.

8.2 Future Work

Skylark provides a scalable and performant framework for stateful serverless execution in the 3D Continuum. Our implementation is open-source and can be used by developers. However, some areas remain open for further exploration and optimization, as partially discussed in Section 7.6. Future work can build upon Skylark by refining its state placement and bundling mechanisms, deployment strategies, and decision heuristics to enhance performance and adaptability.

A potential avenue for future work concerns the optimization of state bundles. Our State Bundling mechanism currently does not consider scalability. Arbitrarily growing the bundle size would lead to a scalability bottleneck. Future work could develop heuristics for balancing the number of state operations with the bundle size.

One promising direction is enhancing the Target Storage Elector with heuristic-based decision-making for cold functions. As mentioned, our current approach picks a storage based only on SLOs when no warm function is present. Future work could incorporate heuristics such as historical execution data and orbit distance. By doing so, Skylark could make more informed state placement decisions, improving state availability even in scenarios where no warm function candidate exists.

Another potential enhancement to Skylark's State Propagation mechanism is extending its search scope beyond the shortest path between the current and tail nodes. In cases where no warm function is detected along the shortest path, neighboring nodes of those already in the path could be considered alternative candidates for State Propagation. This additional search layer could increase the likelihood of finding warm functions, placing the function state closer to the actual execution environment and reducing workflow latency.

The deployment strategy of the Skylark Elect Service presents a trade-off between centralization and decentralization. Deploying an instance on each node ensures minimal lookup latency but increases resource consumption and management overhead. Conversely, a fully centralized instance could become a bottleneck in a cluster containing thousands of nodes. A hybrid approach, inspired by Air Traffic Control systems, where Skylark

Elect instances are geographically distributed across Earth and Low Earth Orbit, could ease these concerns. This strategy would provide regional state election nodes, ensuring efficiency and scalability in global-scale, latency-sensitive workflows.

APPENDIX A

Overview of Generative AI Tools Used

I declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. No content in this thesis was purely generated via prompt and directly copied without significant modifications.

A.1 ChatGPT

The generative AI models ChatGPT-4o[1] and ChatGPT-o1[2] aided in creating the introductory paragraphs of Chapters 2-6. ChatGPT-o1 was further utilized to provide a starting point for the contents of Chapter 2.

A.2 DeepL

DeepL[3] aided in translating the Abstract from English to German.

A.3 Gammarly

Gammarly[4] aided in correcting spelling and grammar issues. The AI tool for text generation was not used in this thesis.

[1] ChatGPT-4o: https://openai.com/index/hello-gpt-4o/
[2] ChatGPT-o1: https://openai.com/index/introducing-openai-o1-preview/
[3] DeepL: https://www.deepl.com/
[4] Gammarly: https://app.grammarly.com/

List of Figures

List of Tables

List of Algorithms

Acronyms

EO Earth Observation. 1–3, 44, 61, 63

FaaS Function as a Service. 1

GEO Geostationary Orbit. 3

ISLs Inter-Satellite Links. 1, 53

LEO Low Earth Orbit. xi, 1, 3–6, 9, 10, 17, 19, 36, 48, 61

MEO Medium Earth Orbit. 2

OEC Orbital Edge Computing. 10

RTT Round-Trip-Time. 18, 22, 23, 29, 30, 44

SLO Service Level Objective. 4–6, 11, 14, 15, 17, 20, 23, 26, 27, 29–31, 44, 46, 47, 49, 55–57, 61

Glossary

Atomic Function A standalone function that runs in an isolated execution environment, maintaining its own state independently, following the traditional serverless function execution model. 19, 20, 25, 51, 53

Atomic State A state representation where each function maintains its own independent key-value state, stored and retrieved separately. 25, 26, 38

Embedded Function A function that runs within a shared execution sandbox alongside other functions, allowing direct memory sharing and optimized state management. 17, 19, 20, 23, 25, 31–33, 36, 43, 51, 52, 61

Embedded State A state representation where multiple co-located functions share and manage their state as a single entity within a shared execution environment. 20, 25, 26, 33, 34, 36, 38, 52, 56

Global State A storage model where the output state of a serverless function is stored in a centralized cloud-based repository. While providing global accessibility and reliability, cloud state introduces higher latency and bandwidth costs compared to localized storage options. 21, 61

Head Node Refers to the local execution environment of a function. Serves as a reference point in the process of electing a Target Storage for State Propagation insofar as it is the first node of the shortest path to the Tail Node. 27, 29–31, 61

Local State A storage model where a function's output state is retained within its execution environment or on a nearby node, reducing access latency and improving data locality. 21, 61

Node Topology The arrangement and connectivity of cloud, edge, and space nodes within the dynamic Edge-Cloud-Space 3D Continuum, where nodes interact based on network latency, bandwidth, and mobility constraints. 18, 19, 22, 61

Skylark A topology- and SLO-aware serverless framework designed to optimize state management in the Edge-Cloud-Space 3D Continuum. It introduces state propagation and bundling mechanisms to reduce workflow latency, improve data locality, and enhance serverless execution efficiency across dynamic network topologies. ix–xi

Skylark Elect Service A distributed service deployed on each node in the Edge-Cloud-Space 3D Continuum that dynamically selects target storage locations based on node topology, SLOs, and function placement. 4, 6, 7, 17, 21–27, 29, 30, 35–39, 48, 49, 55–57, 61

Skylark SDK A software development kit that provides state management functionality for serverless workflows in the Edge-Cloud-Space 3D Continuum. It enables functions to fetch input state, propagate output state, and manage bundled state for embedded functions. 4, 6, 7, 17, 21, 23–29, 31–33, 35, 36, 38, 39, 42–44, 55, 56, 61

SkylarkKey A unique identifier generated by the Skylark SDK to manage function state in serverless workflows. It encodes the workflow instance, target storage, and function ID. 19, 23–25, 30, 33, 34, 38, 39, 43, 44, 52, 61

State Bundling The process of aggregating the state of multiple, co-located functions into a single unit, called Embedded State. 4, 6, 7, 15, 17, 23, 26, 27, 32, 33, 41, 43, 51–53, 55–57, 63, 65

State Propagation The process of transferring the output state of a serverless function to a designated storage location (Target Storage), ensuring availability for its successor function. 3–7, 13–15, 21, 23–30, 39, 41, 43, 44, 46, 51, 55–57, 61

Tail Node The final execution node in a serverless workflow, typically hosting the last function in a sequence. In the Edge-Cloud-Space 3D Continuum, the tail node is often a cloud node, where resource-intensive tasks are performed, and it serves as a reference point for state propagation and function placement decisions. 20, 23, 26–31, 44, 49, 61

Target Function A serverless function that is the designated successor in a workflow sequence, relying on the output state of a preceding function as its input. 6, 20, 22–24, 26, 27, 29–31, 45

Target Storage A designated storage instance selected for holding the output state of a serverless function, ensuring optimal accessibility for the Target Function. In the Skylark model, target storage is dynamically elected based on node topology and service level objectives (SLOs) to maximize data locality and minimize retrieval overhead. xi, 17, 20, 22–31, 33, 34, 39, 45, 52, 65

Bibliography

[1] European Space Agency. *European data relay satellite system (edrs) overview*. Accessed: 2025-02-22. 2024. URL: https://connectivity.esa.int/european-data-relay-satellite-system-edrs-overview.

[2] European Space Agency. *European Space Agency-funded Projects Reach New Performance Level In Groundwork For Optical LEO To GEO Data Relays*. Accessed: 2025-02-22. 2024. URL: https://connectivity.esa.int/news/european-space-agencyfunded-projects-reach-new-performance-level-groundwork-optical-leo-geo-data-relays.

[3] European Space Agency. *Sentinel-2 Operations*. Accessed: 2025-02-22. 2024. URL: https://www.esa.int/Enabling_Support/Operations/Sentinel-2_operations.

[4] European Space Agency and EUMETSAT. *Sentinel High Level Operations Plan (HLOP) (COPE-S1OP-EOPG-PL-15-0020, Issue 3 Rev. 1)*. Accessed: 2025-02-27. 2021. URL: https://sentiwiki.copernicus.eu/web/document-library#DocumentLibrary-SENTINEL-2Documents.

[5] Airbus. *Airbus built Sentinel-2C satellite successfully launched*. Accessed: 2025-02-22. 2024. URL: https://www.airbus.com/en/newsroom/press-releases/2024-09-airbus-built-sentinel-2c-satellite-successfully-launched.

[6] Michael Armbrust et al. „A view of cloud computing“. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672.

[7] AWS. *Amazon S3*. Accessed: 2024-09-20. 2024. URL: https://aws.amazon.com/de/s3/.

[8] Daniel Barcelona-Pons et al. „On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures“. In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 41–54. ISBN: 9781450370097. DOI: 10.1145/3361525.3361535.

[9] Debopam Bhattacherjee and Ankit Singla. „Network topology design at 27,000 km/hour“. In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. CoNEXT '19. Orlando, Florida: Association for Computing Machinery, 2019, pp. 341–354. ISBN: 9781450369985. DOI: 10.1145/3359989.3365407.

[10] Debopam Bhattacherjee et al. „In-orbit Computing: An Outlandish thought Experiment?“ In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. HotNets '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 197–204. ISBN: 9781450381451. DOI: 10.1145/3422604.3425937.

[11] Flavio Bonomi et al. „Fog computing and its role in the internet of things“. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 13–16. ISBN: 9781450315197. DOI: 10.1145/2342509.2342513.

[12] Jan vom Brocke, Alan Hevner, and Alexander Maedche. „Introduction to Design Science Research“. In: Sept. 2020, pp. 1–13. ISBN: 978-3-030-46780-7. DOI: 10.1007/978-3-030-46781-4_1.

[13] Sebastian Burckhardt et al. „Durable functions: semantics for stateful serverless“. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485510.

[14] Canonical. *The effortless Kubernetes*. Accessed: 2025-01-26. 2025. URL: https://microk8s.io/.

[15] Claudio Cicconetti, Marco Conti, and Andrea Passarella. „On Realizing Stateful FaaS in Serverless Edge Networks: State Propagation“. In: *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2021, pp. 89–96. DOI: 10.1109/SMARTCOMP52413.2021.00033.

[16] János Czentye and Balázs Sonkoly. „Serverless application composition leveraging function fusion: Theory and algorithms“. In: *Future Generation Computer Systems* 153 (2024), pp. 403–418. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.12.010. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23004648.

[17] George B. Dantzig. *Linear Programming and Extensions*. Princeton: Princeton University Press, 1963. ISBN: 9781400884179. DOI: doi:10.1515/9781400884179.

[18] Inc. Docker. *Docker - Develop faster. Run anywhere*. Accessed: 2025-02-03. 2025. URL: https://www.docker.com/.

[19] Inc. Docker. *Dockerhub*. Accessed: 2025-02-03. 2025. URL: https://hub.docker.com/.

[20] Google. *Cloud Run functions*. Accessed: 2025-01-24. 2025. URL: https://cloud.google.com/functions.

[21] Leonard Guelmino. *Skylark GitHub Repository*. Accessed: 2025-02-15. 2025. URL: https://github.com/polaris-slo-cloud/skylark.

[22] Simon Kassing et al. „Exploring the "Internet from space" with Hypatia“. In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 214–229. ISBN: 9781450381383. DOI: 10.1145/3419394.3423635.

[23] Knative. *Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications*. Accessed: 2025-01-26. 2025. URL: https://knative.dev/docs/.

[24] Kubernetes. *Production-Grade Container Orchestration*. Accessed: 2025-01-26. 2025. URL: https://kubernetes.io/.

[25] Vincent Lannurien et al. „Serverless Cloud Computing: State of the Art and Challenges“. In: *Serverless Computing: Principles and Paradigms*. Ed. by Rajalakshmi Krishnamurthi et al. Cham: Springer International Publishing, 2023, pp. 275–316. ISBN: 978-3-031-26633-1. DOI: 10.1007/978-3-031-26633-1_11. URL: https://doi.org/10.1007/978-3-031-26633-1_11.

[26] OpenFaaS Ltd. *OpenFaaS - Serverless Functions, Made Simple*. Accessed: 2025-01-13. 2024. URL: https://www.openfaas.com/.

[27] Xiao Ma et al. „Visions of Edge Computing in 6G“. In: *5G Edge Computing: Technologies, Applications and Future Visions*. Singapore: Springer Nature Singapore, 2024, pp. 179–202. ISBN: 978-981-97-0213-8. DOI: 10.1007/978-981-97-0213-8_9. URL: https://doi.org/10.1007/978-981-97-0213-8_9.

[28] C. Marcelino and S. Nastic. „CWASI: A WebAssembly Runtime Shim for Inter-Function Communication in the Serverless Edge-Cloud Continuum“. In: *2023 IEEE/ACM Symposium on Edge Computing (SEC)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2023, pp. 158–170. DOI: 10.1145/3583740.3626611. URL: https://doi.ieeecomputersociety.org/10.1145/3583740.3626611.

[29] Garrett McGrath and Paul R. Brenner. „Serverless Computing: Design, Implementation, and Performance“. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[30] Microsoft. *Azure Blob Storage*. Accessed: 2024-09-20. 2024. URL: https://azure.microsoft.com/de-de/products/storage/blobs/.

[31] Microsoft. *Azure Functions*. Accessed: 2024-09-19. 2024. URL: https://azure.microsoft.com/en-us/products/functions/.

[32] Sergio Moreschini et al. „Cloud Continuum: The Definition“. In: *IEEE Access* 10 (2022), pp. 131876–131886. DOI: 10.1109/ACCESS.2022.3229185.

[33] Matteo Nardelli and Gabriele Russo Russo. „Function Offloading and Data Migration for Stateful Serverless Edge Computing“. In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. London, United Kingdom: Association for Computing Machinery, 2024, pp. 247–257. DOI: 10.1145/3629526.3649293.

[34] Ken Peffers et al. „A design science research methodology for information systems research“. In: *Journal of Management Information Systems* 24 (Jan. 2007), pp. 45–77.

[35] Tobias Pfandzelter. „Serverless Abstractions for Edge Computing in Large Low-Earth Orbit Satellite Networks“. In: *Proceedings of the 24th International Middleware Conference: Demos, Posters and Doctoral Symposium*. Middleware '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 3–6. DOI: 10.1145/3626564.3629088.

[36] Tobias Pfandzelter, Jonathan Hasenburg, and David Bermbach. „Towards a Computing Platform for the LEO Edge“. In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. EuroSys '21. ACM, Apr. 2021. DOI: 10.1145/3434770.3459736. URL: http://dx.doi.org/10.1145/3434770.3459736.

[37] K3s Project. *K3S - Lightweight Kubernetes*. Accessed: 2025-01-13. 2024. URL: https://k3s.io/.

[38] Carlo Puliafito et al. „Stateful Function as a Service at the Edge“. In: *Computer* 55.9 (Sept. 2022), pp. 54–64. ISSN: 1558-0814. DOI: 10.1109/mc.2021.3138690. URL: http://dx.doi.org/10.1109/MC.2021.3138690.

[39] Thomas Pusztai, Cynthia Marcelino, and Stefan Nastic. *HyperDrive: Scheduling Serverless Functions in the Edge-Cloud-Space 3D Continuum*. 2024. arXiv: 2410.16026 [cs.DC]. URL: https://arxiv.org/abs/2410.16026.

[40] Thomas Pusztai et al. „Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge“. In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. 2022, pp. 61–70. DOI: 10.1109/UCC56403.2022.00017.

[41] Thomas Pusztai et al. „Vela: A 3-Phase Distributed Scheduler for the Edge-Cloud Continuum“. In: *2023 IEEE International Conference on Cloud Engineering (IC2E)*. 2023, pp. 161–172. DOI: 10.1109/IC2E59103.2023.00026.

[42] Redis. *Redis IO*. Accessed: 2025-02-03. 2025. URL: https://redis.io/docs/latest/get-started/.

[43] Gabriele Russo Russo et al. „Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum“. In: *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. 2023, pp. 131–140. DOI: 10.1109/PERCOM56429.2023.10099372.

[44] Rust. *Rust - A language empowering everyone to build reliable and efficient software.* Accessed: 2025-02-03. 2025. URL: https://www.rust-lang.org/.

[45] Trever Schirmer et al. „FUSIONIZE++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization“. In: *IEEE Transactions on Cloud Computing* 12.4 (2024), pp. 1172–1185. DOI: 10.1109/TCC.2024.3451108.

[46] Amazon Web Services. *AWS Lambda.* Accessed: 2024-09-19. 2024. URL: https://aws.amazon.com/lambda.

[47] Amazon Web Services. *AWS Step Functions Features.* Accessed: 2024-04-29. 2024. URL: https://aws.amazon.com/step-functions/features/?pg=ln&sec=hs.

[48] Simon Shillaker and Peter Pietzuch. „Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing“. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, July 2020, pp. 419–433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[49] SpaceX. *Starlink Technology.* Accessed: 2024-09-23. 2024. URL: https://www.starlink.com/technology.

[50] Mayank Tiwary et al. „Data Aware Web-Assembly Function Placement“. In: WWW '20. Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 4–5. ISBN: 9781450370240. DOI: 10.1145/3366424.3382670.

[51] Achilleas Tzenetopoulos et al. „FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms“. In: *High Performance Computing.* Ed. by Heike Jagode et al. Cham: Springer International Publishing, 2021, pp. 428–438. ISBN: 978-3-030-90539-2.

[52] Ihsan Ullah et al. „Optimizing task offloading and resource allocation in edge-cloud networks: a DRL approach“. In: *Journal of Cloud Computing* 12.1 (July 2023), p. 112. ISSN: 2192-113X. DOI: 10.1186/s13677-023-00461-3. URL: https://doi.org/10.1186/s13677-023-00461-3.

[53] Bo Wang et al. „Mobile Edge Computing for LEO Satellite: A Computation Offloading Strategy Based Improved Ant Colony Algorithm“. In: *Proceedings of the 11th International Conference on Computer Engineering and Networks.* Ed. by Qi Liu et al. Singapore: Springer Nature Singapore, 2022, pp. 1664–1676. ISBN: 978-981-16-6554-7.

[54] WasmEdge. *WasmEdge - Bring the cloud-native and serverless application paradigms to Edge Computing.* Accessed: 2025-02-03. 2025. URL: https://wasmedge.org/.

[55] Peng Zhang et al. „General Comparison of FY-4A/AGRI With Other GEO/LEO Instruments and Its Potential and Challenges in Non-meteorological Applications“. In: *Frontiers in Earth Science* 6 (2019). ISSN: 2296-6463. DOI: 10.3389/feart.2018.00224. URL: https://www.frontiersin.org/journals/earth-science/articles/10.3389/feart.2018.00224.