📚 Literature Review

🥇 Sim Comparaison

Comparative Study of FaaS Simulators for Edge IoT Applications

The comparison criteria include:

- **Simulator**: The tool's overall purpose and design.
- **Resource Usage**: How the simulator models and tracks CPU, memory, energy, and other resources.
- **Edge Support**: Capability to model resource-constrained edge nodes and IoT workloads.
- **Network Modeling**: Ability to simulate network dynamics critical for edge-cloud interactions.
- **Configurability**: Flexibility to customize scheduling, infrastructure, and workloads.

These aspects are critical for selecting a simulator that supports IoT application simulation, energy and cost metric tracking, and real-world validation.

Comparative Table

The table below summarizes the simulators based on the specified aspects, derived from the document's details:

Simulator	**Resource Usage**	**Edge Support**	**Network Modeling**	**Configurability**
MFS	Tracks CPU, RAM, GPU usage; detailed container lifecycle (cold/warm starts); no energy metrics	Partial; basic edge support, primarily cloud-focused	Limited; basic network modeling, no dynamic topologies	Medium; supports custom scheduling but limited by simple algorithms
ServerlessSimPro	Tracks CPU, memory, energy consumption; detailed container states (cold start, idle); PM power usage	Limited; no explicit edge support, cloud-centric	Basic; minimal network modeling	High; extensive scheduling options (FirstFit, Linear Programming), customizable parameters
SimFaaS	Tracks instance utilization (CPU, memory); no energy or container lifecycle modeling	Full; supports hybrid cloud-edge environments, region-based latency	None; no explicit network modeling	High; modular, supports custom scheduling and instance provisioning (
Simulator	**Resource Usage**	**Edge Support**	**Network Modeling**	**Configurability**

CloudSimSC	Tracks CPU, memory, VM efficiency; basic container lifecycle (cold/warm starts)	Limited; basic edge extensions, not tailored for edge constraints	Basic; limited network simulation, weak for IoT/edge	High; flexible scheduling (Round Robin, Bin Packing), extensible architecture
EdgeFaaS	Tracks CPU, memory, energy consumption; supports heterogeneous resources	Full; models edge-specific orchestration, dynamic failures	Limited; no data flow or bandwidth modeling	Medium; customizable infrastructure via YAML/Prolog, but limited placement policies
faas-sim	Tracks CPU, memory, network usage; detailed resource consumption per function	Full; supports heterogeneous edge devices, IoT workloads	Flow-based; <7% error, also models geo-distributed topologies	High; modular, supports custom schedulers, topologies, and traces

Detailed Analysis

1. Simulator Overview

- **MFS**: A Python-based simulator modeling Apache OpenWhisk, focused on cloud FaaS with realistic container lifecycle simulation . Suitable for performance and cost analysis but less tailored for edge IoT scenarios.
- **ServerlessSimPro**: A comprehensive cloud-centric simulator using real-world traces (AzureFunctionsInvocationTrace2021), emphasizing energy tracking and scheduling flexibility Ideal for energy and cost metrics but lacks edge support.
- **SimFaaS**: A modular simulator for cloud-edge FaaS, focusing on QoS-aware scheduling without detailed container modeling . Good for hybrid environments but limited in energy metrics.
- **CloudSimSC**: Extends CloudSim for serverless computing, offering generalizable scheduling and scaling but lacking edge-specific fidelity . Useful for broad experimentation but not IoT-focused.
- **EdgeFaaS**: A Python-based simulator for edge FaaS orchestration, supporting dynamic infrastructure and energy tracking . Well-suited for IoT workflows and edge validation.
- **faas-sim**: A trace-driven, SimPy-based simulator for edge FaaS, modeling IoT workloads and network dynamics with high fidelity . Highly relevant for smart city and accident prevention use cases.

EdgeFaaS and faas-sim are most aligned due to their **edge focus,** supporting IoT applications. ServerlessSimPro is valuable for **energy metrics**, while MFS and CloudSimSC are less suitable due to limited edge support. SimFaaS offers hybrid flexibility but **lacks critical metrics.**

2. Resource Usage

- **MFS**: Tracks CPU, RAM, GPU usage and container lifecycle metrics (response time, rejection ratio) but omits energy consumption. Limited for energy-focused IoT studies.
- **ServerlessSimPro**: Excels in tracking CPU, memory, and energy consumption (PM power usage), with detailed container states Ideal for analyzing energy-cost trade-offs.
- **SimFaaS**: Tracks instance utilization (CPU, memory) but abstracts containers, missing energy and cold start metrics . Less comprehensive for resource analysis.

- **CloudSimSC**: Monitors CPU, memory, and VM efficiency, with basic container lifecycle support Adequate but not detailed for edge resource constraints.
- **EdgeFaaS**: Tracks CPU, memory, and energy consumption, supporting heterogeneous resources . Strong for edge IoT resource modeling.
- **faas-sim**: Tracks CPU, memory, and network usage per function, with detailed resource consumption Excellent for IoT workload analysis.

ServerlessSimPro, **EdgeFaaS, and faas-sim** provide the most comprehensive resource tracking, particularly for energy, which is critical for evaluating performance-to-cost ratios in IoT deployments. MFS and SimFaaS are less suitable due to missing energy metrics, and CloudSimSC is too general.

3. Edge Support

- **MFS**: Offers partial edge support, primarily cloud-focused, limiting its ability to model resource-constrained edge nodes
- **ServerlessSimPro**: Lacks explicit edge support, focusing on cloud environments . Not ideal for IoT edge scenarios.
- **SimFaaS**: Fully supports hybrid cloud-edge environments, modeling region-based latency and edge nodes Suitable for IoT but lacks container granularity.
- **CloudSimSC**: Provides limited edge extensions, not tailored for edge constraints or IoT workloads
- **EdgeFaaS**: Fully supports edge environments, modeling heterogeneous resources, dynamic failures, and IoT orchestrations . Highly relevant for internship goals.
- **faas-sim**: Offers full edge support, modeling heterogeneous devices (e.g., Raspberry Pi, Jetson) and IoT workloads like AI inference . Optimized for edge IoT.

EdgeFaaS and faas-sim are the strongest for edge support, directly addressing **smart city and accident prevention IoT applications.** SimFaaS is viable for hybrid setups, but MFS, ServerlessSimPro, and CloudSimSC are less applicable due to weak edge modeling.

4. Network Modeling

- **MFS**: Limited to basic network modeling, not supporting dynamic topologies or edge-specific dynamics
- **ServerlessSimPro**: Basic network modeling, insufficient for edge-cloud interactions
- **SimFaaS**: Lacks explicit network modeling, relying on region-based latency configurations
- **CloudSimSC**: Limited network simulation, inadequate for IoT or edge scenarios .
- **EdgeFaaS**: Limited; does not model data flow or bandwidth, a noted limitation
- **faas-sim**: Uses flow-based network simulation, Best for edge-cloud network dynamics. (could we model energy)

faas-sim stands out for its robust network modeling, critical for simulating IoT data transfers in smart city scenarios. Other simulators are weaker, limiting their ability to capture edge network constraints.

5. Configurability

- **MFS**: Medium configurability; supports custom scheduling but constrained by simple algorithms
- **ServerlessSimPro**: High configurability; offers extensive scheduling options (FirstFit, Linear Programming) and customizable parameters
- **SimFaaS**: High configurability; modular design allows custom scheduling and instance provisioning

- **CloudSimSC**: High configurability; supports flexible scheduling (Round Robin, Bin Packing) and extensible architecture
- **EdgeFaaS**: Medium configurability; customizable infrastructure via YAML/Prolog but limited by placement policies
- **faas-sim**: High configurability; modular design supports custom schedulers, topologies, and trace-driven workloads

ServerlessSimPro, SimFaaS, CloudSimSC, and **faas-sim** offer high configurability, enabling experimentation with **IoT workloads** and **scheduling strategies**. EdgeFaaS is moderately flexible, while MFS is less adaptable.

Alignment with Internship Goals

- faas-sim and EdgeFaaS provide the most advanced edge-focused simulation, with faas-sim's trace-driven approach and EdgeFaaS's orchestration modeling offering deep insights into FaaS at the edge
- **Simulate IoT Application**: faas-sim's support for IoT workloads (AI inference, smart city scenarios) and EdgeFaaS's workflow orchestration make them ideal for simulating smart city or accident prevention applications SimFaaS is a secondary option for hybrid setups
- **Validate with production environment deployment**: faas-sim's trace-driven validation and EdgeFaaS's energy and performance metrics facilitate comparison with production environment deployment results ServerlessSimPro's energy metrics are also useful but limited by cloud focus
- **Evaluate Serverless Benefits**: faas-sim's comprehensive metrics (FET, resource usage, implied cost) and ServerlessSimPro's energy and cost tracking enable robust analysis of latency, cost, and energy efficiency . EdgeFaaS supports similar evaluations with edge-specific metrics

Key Questions Addressed

- **FaaS and Edge Integration**: EdgeFaaS and faas-sim address opportunities (low latency, reliability) and challenges (resource constraints, scheduling) by modeling edge-specific features
- **IoT Applications**: faas-sim's smart city use case and EdgeFaaS's workflow support demonstrate FaaS's effectiveness for traffic management and accident prevention, reducing latency and improving responsiveness
- **Energy and Cost**: ServerlessSimPro and EdgeFaaS track energy consumption, with faas-sim implying cost via resource usage, supporting performance-to-cost analysis

Decision

- **Primary Choice**: **faas-sim** is the most suitable due to its edge support, trace-driven fidelity, network modeling, and comprehensive metrics, ideal for IoT simulations
- **Secondary Choice**: **EdgeFaaS** is strong for edge orchestration and energy tracking, suitable if IoT workflows are complex
- **Consideration**: **ServerlessSimPro** is valuable for energy and cost metrics but requires extensions for edge support
- **Avoid**: MFS, SimFaaS, and CloudSimSC are less suitable due to limited edge support or missing critical metrics

ServlessSimPro

ServlessSimPro : A Comprehensive Serverless Simulation Platform - Review

Serverless Computing, Simulation, Scheduling

● **Closest to reality, comprehensive metrics** ● **scheduling flexibility, comparison**

Motivation:

The need of sim tools +
Limited availability of open-source simulation platforms that can :

- Emulate serverless env accurately
- Free and convenient for researchers
- Comprehensive interfaces for scheduling strategies
- Comprehensive set of monitoring metrics

The experimental results demonstrate that the scheduling algorithm of the simulator can effectively reduce latency, enhance resource utilization, and decrease energy consumption..

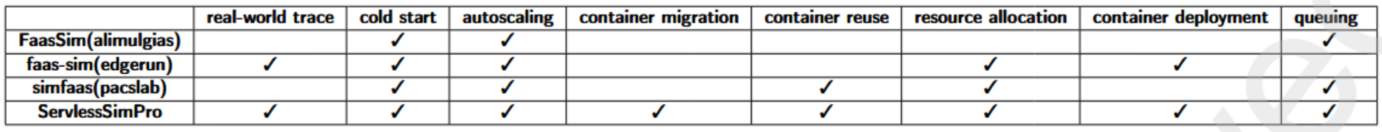
2. Methodology & System Design

- **ServlessSimPro Overview**
  - Developed to **accurately model serverless execution** using real-world traces (AzureFunctionsInvocationTrace2021 dataset).
  - **Three-tier architecture**: Physical Machines (PMs), Containers, Functions .
- **Comparison with Existing Simulators**
  - **CloudSim:** large scale ds, scalability & control + energy-awareness
  - **iCanCloud**: Cost/Performance forecasting, sim large env with custom detail

General cloud simulators lack serverless features. [statelessness of functions, event-driven architecture, automatic scalability, and the transient lifecycle of functions.]

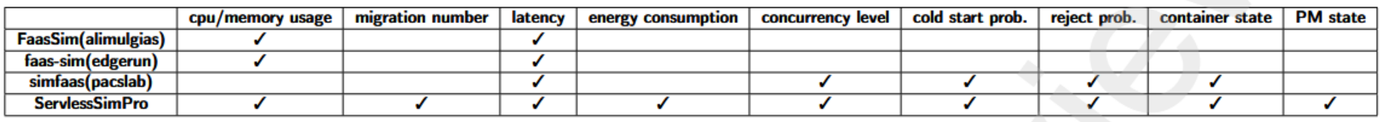
  - **FaasSim :** indep tasks, limited metrics for sless scheduling strategies
  - **faas-sim** (edgerun): trace-driven, spec: sless edge computing systems, network metrics, lack scheduling flexibility and energy-awareness.
  - **SimFaaS** (pacslab) **:** qos metrics, edge env, instances could be hosts, virtual machines, or containers, (instances running functions)=> 2 tiers (instances, functions) != 3 tiers, CPU and memory resources, migration or deployment
- **Unique Contributions of ServlessSimPro**:
  - More **scheduling algorithms** than any other simulator.
  - **Tracks energy consumption** (first to include this metric).
  - Supports **container migration and queuing**, unlike competitors

Table 1
Comparison of Serverless Simulators: System Features



	real-world trace	cold start	autoscaling	container migration	container reuse	resource allocation	container deployment	queuing
FaasSim(alimulgias)		✓	✓					✓
faas-sim(edgerun)	✓	✓	✓			✓	✓	
simfaas(pacslab)		✓	✓		✓	✓		✓
ServlessSimPro	✓	✓	✓	✓	✓	✓	✓	✓

Table 2
Comparison of Serverless Simulators: Performance Metrics



	cpu/memory usage	migration number	latency	energy consumption	concurrency level	cold start prob.	reject prob.	container state	PM state
FaasSim(alimulgias)	✓		✓						
faas-sim(edgerun)	✓		✓						
simfaas(pacslab)			✓		✓	✓	✓	✓	
ServlessSimPro	✓	✓	✓	✓	✓	✓	✓	✓	✓

3. Features & Metrics

- **Key Features** (Section 3.2 - 3.4)
    - Existing research :
        - **Concurrency Scaling:** Adjusting the concurrency level impacts performance and latency.
        - **Memory-Latency Tradeoff:** Finding an optimal balance (Saha et al.).
        - **CE-Scaling Framework:** Dynamic resource allocation for ML workloads using heuristic algorithms (Wu et al.).
    - Supports multiple scheduling strategies:
        - Resource allocation
        - **Container deployment :**
            - **First-Fit Algorithm:** Simplest approach, placing containers in the first available spot
            - **Linear Programming:** More efficient, reducing cost by ~5% and improving scalability (Boukadi et al.)
            - **Energy Optimization:** Dynamic programming-based heuristics reduce execution time and energy consumption (Lin et al.).
        - Migration :
            - **Load Balancing & Cost Reduction:**
                - Balance-Aware Placement (BACP) & Adaptive Threshold Migration (ATCM) ensure efficient load distribution (Nadgowda et al.).
            - **Resource Consolidation**: Migrating only long-running containers optimizes energy and performance (Khan et al.).
        - Reuse & Caching :
            - **Fixed Caching (AWS, Google, IBM, Azure):** Containers remain cached for 10-20 minutes.
            - **Warm Queue (FIFO Replacement):** Alternative approach to optimize cold start handling.
            - **Prewarm vs. Keep-Alive Time Windows:** Reducing cold starts using historical data.

    - **Energy-aware design**: Measures CPU/memory usage, power consumption, cold start impact.
    - **Concurrency and queuing control**: Dynamically adjusts execution limits based on function loads.
- **Metrics Evaluated** (Section 3.3)
    - **Performance**: Response time, latency, cold start probability.
    - **Resource Utilization**: CPU, memory, number of active containers.
    - **Energy Consumption**: PM power usage based on CPU lo

4. Architecture :

- **Control Layer** – Handles scheduling, automatic scaling, and event-driven execution. It manages incoming requests and collects performance metrics.
- **Task Layer** – Converts requests into tasks, processes them, and logs resource state changes. Uses a min-heap for scheduling tasks.
- **Resource Layer** – Consists of physical machines (PMs), containers, and functions. Simulates container states (cold start, running, idle, dead) and manages their resource allocation.

ServlessSimPro

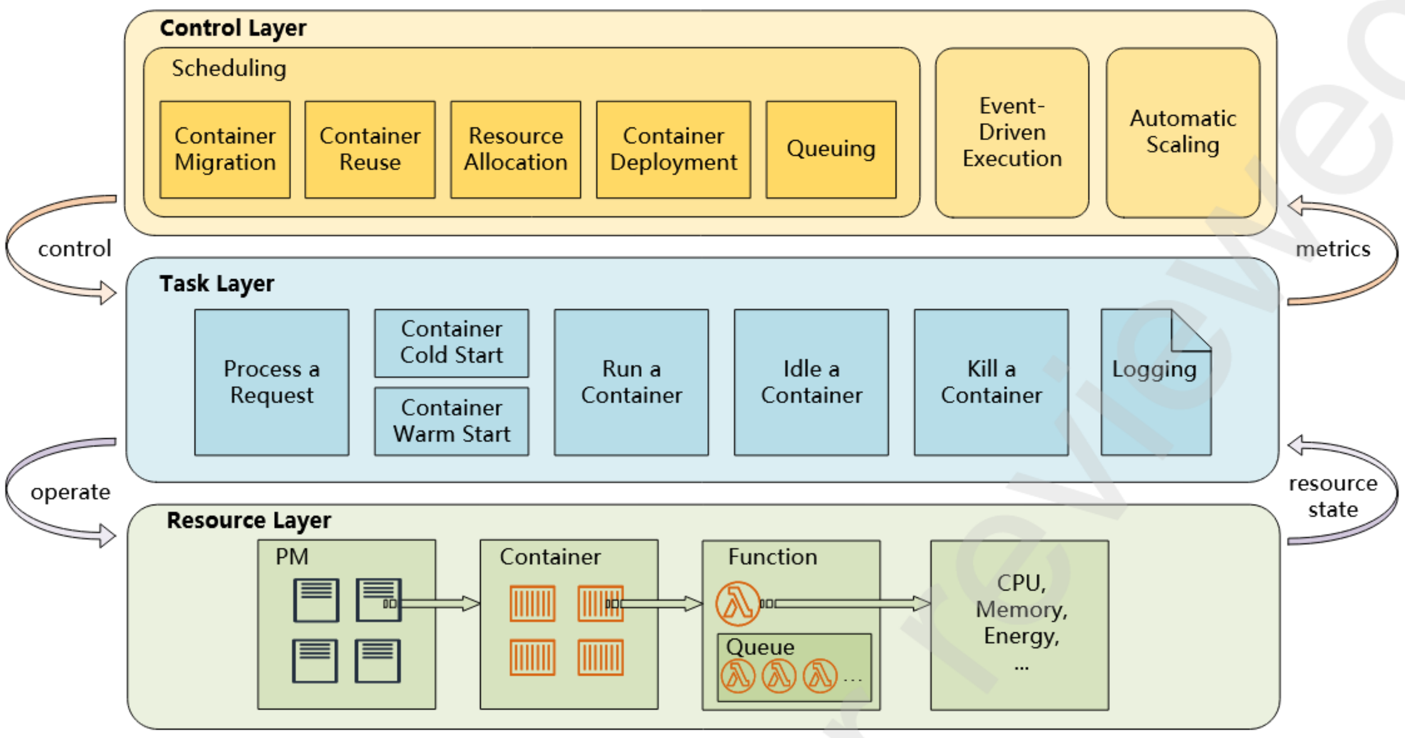


Figure 1: Architecture

Experiments 👍

- **EarliestKilled**: Prioritizes terminating the earliest allocated containers.
- **LatestKilled**: Terminates the most recently allocated containers first.
- **RandomSelection**: Randomly selects a container to terminate.

  FirstFit: Allocates functions to the first available container, optimizing resource usage.
- **MinVectorDistance**: Allocates functions based on a metric minimizing distance in a multi-dimensional resource space.

FCFS (First Come, First Served): Processes requests in arrival order, used in queuing and concurrency control.

- **Findings from the Experiments**
  - **Resource Allocation**
    EarliestKilled had the **highest energy consumption** but **lowest latency**.
    - **LatestKilled** had the **lowest energy consumption** but **highest latency**.
    - **RandomSelection** performed **between the two** in both metrics.
  - **Container Deployment**
    - **FirstFit** was more **efficient than MinVectorDistance**, leading to **lower energy consumption, better resource allocation, and fewer active containers**.
    - FirstFit optimizes the number of physical machines (PMs) in use, improving resource efficiency.
  - **Queuing and Concurrency Control (QCC)**
    - Enabling QCC **reduced energy consumption** and **decreased cold starts**, but **increased latency and rejection rates**.
    - The larger the queue thresholds, the less effective QCC became, eventually reverting to a non-QCC state.
    - **threshconc (threshold for concurrent executions) had a more significant impact than maxlen (queue length)**.
  - **Resource Consolidation**
    - Consolidation **decreased energy consumption, increased resource usage, and reduced the number of PMs needed**.
    - However, it also **increased latency and cold start occurrences**
    - A **shorter consolidation interval amplified these effects**.

4. Findings & Contributions (Section 5, Conclusion)

- **More Realistic Serverless Simulation**
  - Models real-world function executions more accurately.
  - Provides **fine-grained** monitoring and tracking of **container states, PM usage, and queuing behavior**.
- **Energy Efficiency as a Key Metric**
  - First simulator to track **energy consumption in serverless computing**.
  - Helps **optimize scheduling strategies** for cost and performance trade-offs.
- **Flexible and Extensible Design**
  - Allows researchers to **modify and test scheduling algorithms easily**.
  - Provides **customizable parameters** for fine-tuning function execution.

5. Limitations & Future Work

- **Current Limitations:**
  - **No support for edge computing** (unlike some other simulators).
  - **Does not include VM-based simulations**, only PM-container-function layers.
- **Future Enhancements:**
  - Extending the simulator to **edge computing** scenarios.
  - Integrating **virtual machines** for more diverse workload modeling.

Overall Assessment

Strengths:

- **Most comprehensive scheduling options** among simulators.
- **First simulator to track energy consumption in serverless computing.**
- **Highly configurable** with real-time monitoring.

Weaknesses:

- **No edge computing support.**
- **No direct VM-level simulation.**

MFS

MFS: A Serverless FaaS Simulator - Review

Serverless Computing, FaaS (Function-as-a-Service), Simulation , Openwhisk

🟢 **Closest to reality, comprehensive metrics** 🔴 **scheduling flexibility, comparison**

Motivation:

Serverless computing helps developers implement their codebase without worrying about server management hassles. These are the reasons why the concept of serverless computing is presented in the cloud, the edge, the fog, and the Internet of things. Research on serverless computing, on the other hand, requires extensive hardware equipment. It can also be time consuming. Simulation is needed to overcome these problems.

Methodology & System Design

- **MFS Overview**: Written in Python and based on Apache OpenWhisk, MFS provides a **detailed metrics & simulation** environment for FaaS platforms. (Section III, System Design)
- **Architecture**: MFS models Apache OpenWhisk components, including a controller (global scheduler), physical machines (local schedulers), containers, functions, and event management. (Section III, System Design)
- **Comparison with Existing Simulators**:(Section II, Related Work)
    - CloudSim (Python based, widely used but not **serverless-specific** features).
    - SimFaaS (Calculates metrics, also Python-based but Oversimplified : lacks **container support** and **function reuse**).

5. Features & Metrics

- **Supported Features**: Models container lifecycle (cold/warm starts), heterogeneous resources (CPU, GPU, TPU), and different runtime environments. (Section IV, MFS Features)
- **Metrics Evaluated**:
    - Performance: Response time, waiting time, service time, throughput.
    - Resource Utilization: CPU, RAM, GPU usage.
    - Reliability: Function rejection/missed ratio, completion ratio.
    - Cost Estimation: Based on unit pricing models from AWS Lambda.

6. Findings & Contributions (Section VI, Conclusions)

- **Realistic Serverless Simulation**: More accurate than prior simulators due to OpenWhisk-based architecture.
- **Comprehensive Metric Reporting**: Captures performance, cost, and resource usage more effectively.
- **Supports Container Lifecycle**: Unlike SimFaaS, allows container reuse, reducing cold start impact.

7. Limitations & Future Work (Section VI, Conclusions)

- **Simple Scheduling Algorithms**: Current strategies do not consider function chains or dependencies.
- **Potential Enhancements**: Future work aims to improve resource allocation and *introduce advanced scheduling policies*.

Overall Assessment

- **Strengths**:
    - Realistic modeling of serverless environments (Base Platform, Containers support).
    - Comprehensive metrics for performance evaluation.
- **Weaknesses**:
    - Limited scheduling flexibility.
    - Lack of extensive comparisons with alternative simulators.

Lifecycle of a Simulation:

- **Initialization:**
  - **Input Preparation:** Functions are generated with defined attributes (e.g., submission time, resource requirements, deadlines).
  - **Event List Creation:** All function arrival events are created based on their submission times, using a priority queue structure.
- **Simulation Loop:**
  - **Event Fetching:** The simulator enters a loop where it fetches the next event (e.g., function arrival, function completion) from the event list.
  - **Global Scheduling:**
    - The **Controller** (global scheduler) receives function arrival events.
    - It selects an appropriate **P**hysical **M**achine (PM) based on a chosen scheduling strategy (e.g., load balancing or utilization).
  - **Local Scheduling:**
    - The selected PM (representing a worker node) invokes its **local scheduler**.
    - The local scheduler checks for available warm containers to run the function.
    - If no warm container is available, a new cold container is instantiated.
  - **Function Execution:**
    - The function is executed within the chosen container.

        - Metrics such as response time, waiting time, and resource utilization are updated during execution.
    - **Event Update:** Once a function is executed, the corresponding completion event is added to the event list, and container status is updated (e.g., remains warm for a specified period).
- **Termination:**
  The simulation continues until the event list is empty, meaning all function events have been processed.
    - Overall metrics (e.g., average response time, throughput, cost) are then calculated and reported.

SimFaas

SimFaaS: A Simulator for Serverless Computing in Cloud-Edge Environments - Review

Serverless Computing, Simulation, Scheduling, Edge Computing

🟢 **Modular, QoS-aware, supports multiple environments (cloud & edge)**

🔴 **Lacks container-level granularity, no energy metrics, less dynamic queuing**

Motivation:

- Rise of Function-as-a-Service (FaaS) and challenges in evaluating strategies under diverse environments.
- Existing simulators (like CloudSim, iCanCloud, EdgeSim) are not tailored for the ephemeral, event-driven, stateless nature of serverless computing.
- Need for a flexible simulation tool to:
  - Model hybrid cloud-edge FaaS environments.
  - Evaluate QoS-aware scheduling strategies.
  - Allow configurable simulation components for rapid prototyping.

2. System Design

General Architecture

- Two-tier system: Function Requests → Execution Instances
- Execution instances may represent:
  - Containers
  - Virtual Machines
  - Edge Nodes
- Instances are abstract—no deep modeling of container lifecycle (no cold start, reuse, etc.)
- Function requests include: resource requirements, deadlines, origin (edge/cloud), etc.

Scheduling Support

- Central component: the Scheduler
  - Decides request placement, instance provisioning, and termination.
- Implemented strategies are modular—developers can inject custom policies.

Environment Configuration

- Supports defining multiple Regions, each with different latency, capacity, and pricing.
- Simulates heterogeneous infrastructure, suitable for hybrid scenarios.

3. Features & Metrics

QoS Constraints (Section II.C)

- Allows simulation of request deadlines, resource needs, and origin types.

- Simulation outputs include success/failure of requests based on QoS violations.

Extensible Components (Section II.D)

- Users can plug in:
  - Custom scheduling algorithms
  - New function types
  - Specific instance provisioning rules
- Designed for rapid experimentation.

Metrics Tracked (Section II.E)

- Request Success Ratio
- Average Execution Delay
- Scheduler Decision Overhead
- Instance Utilization
- No explicit energy consumption tracking
- No modeling of container cold start or caching

4. Experiments & Findings (Section III. Evaluation)

Experiment Setup

- Compared centralized vs decentralized (cloud-edge) schedulers
- Load scenarios: Poisson-distributed functions across different regions
- Variables: deadlines, CPU/mem requirements, latency

Findings

- Decentralized scheduling improved success ratio under high latency and strict deadlines
- Centralized scheduling suffered due to bottlenecks and increased response time
- SimFaaS could simulate the performance tradeoffs of different infrastructures (edge, cloud, hybrid)

5. Contributions (Section I, II, III)

Modular and Pluggable Simulator

- Extensible architecture for experimenting with FaaS scheduling in hybrid environments
- Flexible workload modeling (requests with QoS, variable resource demand)

QoS-Aware Scheduling Evaluation

- Simulates deadline-sensitive function dispatching
- Offers insights into centralized vs decentralized function placement

Cloud-Edge Support

- One of few simulators that support hybrid deployment models
- Includes latency and region configuration for realistic edge-cloud behavior

6. Limitations & Future Work (Section IV. Conclusion)

Limitations

- No energy consumption metrics
- Abstracts containers as "instances" → lacks modeling of container cold starts, caching, or migration
- Limited real-world trace support (mainly synthetic experiments)

Future Work

- Incorporating realistic traces from existing FaaS platforms

- Modeling transient behavior (e.g., cold starts)
- Support for energy-aware scheduling policies

Overall Assessment

Strengths:

- Modular simulator with strong QoS focus
- Supports hybrid cloud-edge infrastructure
- Good for researchers testing scheduling under varying deadlines and resource constraints

Weaknesses:

- No fine-grained container modeling (cold starts, reuse, etc.)
- Lacks energy tracking or migration capabilities
- Simpler compared to more comprehensive simulators like ServerlessSimPro

CloudSimSC

CloudSimSC: A Serverless FaaS Simulator - Review

Serverless Computing, FaaS (Function-as-a-Service), Simulation, CloudSimSC

🟢 **Generalizable architecture, customizable policies** 🔴 **Limited real-world execution fidelity, lacks provider-specific execution models**

Motivation:

Serverless computing simplifies software deployment by abstracting infrastructure concerns, allowing developers to focus on code rather than server management. However, research in serverless environments requires access to real-world infrastructures, which can be expensive and time-consuming. Simulation frameworks like CloudSimSC aim to bridge this gap by providing a flexible, scalable, and extensible simulation environment.

Methodology & System Design

- **CloudSimSC Overview:**

CloudSimSC extends the widely used CloudSim framework to model serverless platforms. It introduces FaaS-specific elements, including function execution, auto-scaling policies, and scheduling algorithms. (Section III, System Design)

- **Architecture:**

CloudSimSC integrates components such as:

- **ServerlessController:** Manages request handling and resource provisioning.
- **RequestLoadBalancer:** Routes incoming function calls based on policies.
- **ServerlessDatacenter:** Represents the infrastructure layer, including virtual machines and containers.
- **FunctionScheduler:** Selects execution nodes using predefined policies (e.g., Round Robin, Bin Packing).
- **FunctionAutoScaler:** Implements horizontal and vertical scaling strategies.

Comparison with Existing Simulators:

	Key Features	**Limitations**
CloudSim	General-purpose cloud simulation	Not designed for serverless computing
SimFaaS	Models FaaS execution	Lacks detailed container lifecycle management
OpenDC	Focuses on datacenter-level modeling	Does not consider serverless function execution styles

Features & Metrics

- **Supported Features:**
- Simulates function execution environments (cold/warm starts).
- Supports dynamic resource allocation and scaling policies.
- Provides flexible scheduling algorithms (Round Robin, Bin Packing, First Fit).

Metrics Evaluated:

1. **Performance:** Function response time, execution latency, scheduling delay.
2. **Resource Utilization:** CPU, memory, and VM efficiency.
3. **Cost Estimation:** Infrastructure costs based on function execution.

Findings & Contributions (Section VI, Conclusions)

🔹 **Generalizable Simulation Framework:** Supports multiple execution styles (scale-per-request, request concurrency).
🔹 **Configurable Scheduling & Scaling:** Allows different auto-scaling strategies for realistic workload handling.
🔹 **Extensibility:** Provides modular design, enabling integration with future scheduling algorithms.

Limitations & Future Work (Section VI, Conclusions)

🔸 **Lacks Real-world Execution Fidelity:** Does not fully replicate cloud provider execution behaviors (e.g., AWS Lambda, Google Cloud Functions).
🔸 **Simplified Cost Models:** Does not include provider-specific billing mechanisms.
🔸 **Limited Network Simulation:** Requires improvements for IoT and edge computing scenarios.

Overall Assessment

Strengths:

- Supports various scheduling and scaling policies.
- Provides a flexible, extensible architecture.
- Enables provider-independent simulation for generalizability.

Weaknesses:

- Lacks execution fidelity for cloud-provider-specific implementations.
- Does not model network constraints affecting serverless execution.

Lifecycle of a Simulation:

- **Initialization:**
  - Define the simulation environment (e.g., number of datacenters, hosts, VMs, containers).
  - Load function execution parameters (cold start, warm start, concurrency settings).
  - Configure scheduling and auto-scaling policies.
- **Workload Generation:**
  - Inject function invocation requests based on workload traces or synthetic models.
  - Distribute requests across available resources via the load balancer.
- **Scheduling & Execution:**
  - Assign function execution requests to available compute resources.
  - Apply scheduling policies (e.g., Round Robin, First Fit).
  - Trigger cold/warm starts depending on container availability.
- **Scaling & Resource Management:**
  - Monitor system metrics (CPU, memory usage, execution latency).
  - Trigger horizontal or vertical scaling based on workload changes.
- **Performance Evaluation:**
  - Record execution times, function latency, resource utilization, and cost estimations.
  - Generate logs for analysis and debugging.
- **Simulation Termination:**
  - Conclude execution upon reaching predefined simulation time or workload completion.
  - Output results for comparative evaluation.

EdgeFaaS

Simulating FaaS Orchestrations In The Cloud-Edge Continuum - Review

Computing methodologies, Modeling and simulation, Networks

🟢 **Closest to reality, comprehensive metrics** 🔴 **scheduling flexibility, comparison**

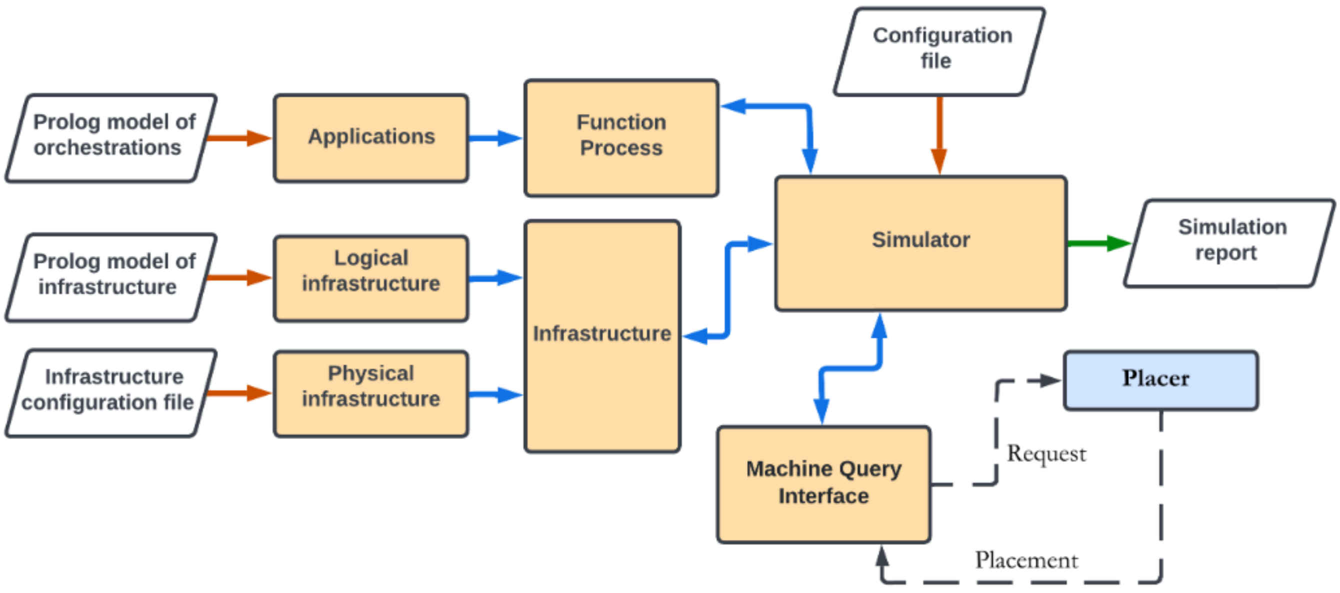
Motivation:

The paper addresses the challenge of deploying Function-as-a-Service (FaaS) orchestrations in a heterogeneous Cloud-Edge continuum. Real infrastructures for such experiments are expensive and time-consuming so a simulation environment is essential.

- **Need for Simulation:** To evaluate management proposals for FaaS applications considering their ephemeral, on-demand, and pay-per-use characteristics.
- **Research Gap:** Existing simulators focus mostly on cloud-only scenarios and do not adequately cover orchestrations across both cloud and edge resources, nor do they account for dynamic network topologies and energy consumption.

Methodology & System Design

- **Overview**: EdgeFaaS is a Python-based simulator that models FaaS application orchestration across a distributed, heterogeneous infrastructure.
- **Architecture**:



Phases: Alternates between (i) failure/migration and (ii) placement/execution phases (Section 3.3)

Modeling:

- FaaS Orchestrations: Declared as workflows (seq, if, par) with hardware/software/latency requirements (Section 3.2.1).
- Infrastructure: Nodes, links, and services are configurable via YAML/Prolog, with dynamic failure/resurrection support (Section 3.2.2).

Comparison with Existing Simulators:(Section 5)

- DFaaSCloud [11]: Supports Cloud-Edge but lacks orchestration and energy modeling.
- SimFaaS [14]: Focuses on scalability but ignores Edge and dynamic topologies.
- FaaSSim [17]: Simulates Edge but does not model energy or partial orchestration migrations.
- Unique contributions of λContSim:
  - Orchestration-aware placement.

        - Dynamic infrastructure changes.
        - Energy consumption tracking.

5. Features & Metrics

- Key Features:
    - Ephemeral function states (WAITING, RUNNING, CANCELED, etc.) (Section 3.2.1).
    - Customizable infrastructure (randomized or user-defined) (Section 3.2.2).
    - Partial re-deployment after failures (Section 3.3.1).
- Metrics Collected:
    - Performance: Placement service time (Q1, Fig. 4).
    - Reliability: Success/failure rates of placements (Q2, Fig. 5).
    - Energy: Infrastructure consumption (Q3, Fig. 6).

6. Findings & Contributions

- Realistic Evaluation: Demonstrated via a case study with 700 experiments, varying infrastructure sizes (Section 4).
- Placement Strategy Insights:
    - Service time fluctuates (60–180 ms) independent of infrastructure size (Fig. 4).
    - Success rates improve with larger infrastructures (28% to 84%) (Fig. 5).
    - Energy consumption diverges from baseline as infrastructure scales (Fig. 6).

7. Limitations & Future Work (Section VI, Conclusions)

- Data Abstraction: Does not model data flow or bandwidth (Section 6).
- Enhanced Placement Policies: Plans to compare other methodologies (Section 6).

Lifecycle of a Simulation:

- **Initialization:**
    - **Input Preparation:** Functions are generated with defined attributes (e.g., submission time, resource requirements, deadlines).
    - **Event List Creation:** All function arrival events are created based on their submission times, using a priority queue structure.
- **Simulation Loop:**
    - **Event Fetching:** The simulator enters a loop where it fetches the next event (e.g., function arrival, function completion) from the event list.
    - **Global Scheduling:**
        - The **Controller** (global scheduler) receives function arrival events.

        - It selects an appropriate **P**hysical **M**achine (PM) based on a chosen scheduling strategy (e.g., load balancing or utilization).
    - **Local Scheduling:**
        - The selected PM (representing a worker node) invokes its **local scheduler**.
        - The local scheduler checks for available warm containers to run the function.
        - If no warm container is available, a new cold container is instantiated.
    - **Function Execution:**
        - The function is executed within the chosen container.
        - Metrics such as response time, waiting time, and resource utilization are updated during execution.
    - **Event Update:** Once a function is executed, the corresponding completion event is added to the event list, and container status is updated (e.g., remains warm for a specified period).
- **Termination:**
  The simulation continues until the event list is empty, meaning all function events have been processed.
    - Overall metrics (e.g., average response time, throughput, cost) are then calculated and reported.

🟧 Faas-Sim Sless + Edge

faas-sim : A trace-driven simulation framework for serverless edge computing platforms - Review

Computing methodologies, Modeling and simulation, Networks

🟢 **Closest to reality, comprehensive metrics, Configurable** 🔴 **profiling overhead, learning curve**

Motivation

Serverless edge computing extends FaaS to edge environments, enabling low-latency, scalable IoT applications. However, evaluating such platforms is challenging due to the lack of standardized benchmarks, reference architectures, and real-world testbeds for edge computing, unlike cloud-centric systems. Existing simulators (CloudSim) often use oversimplified resource models, failing to capture edge-specific challenges like device heterogeneity, dynamic network conditions, and spatio-temporal workloads. The authors address these gaps by developing faas-sim, a flexible, trace-driven simulator that supports:

- **Platform Design**: Evaluating architectures (centralized, hybrid, decentralized) for edge-cloud FaaS.
- **Function Adaptation**: Testing scheduling, scaling, and routing strategies.
- **Resource Planning**: Estimating hardware needs for workloads.
- **Performance Optimization**: Analyzing latency, throughput, and cost for IoT scenarios.

Methodology

Simulation Setup

faas-sim is a Python-based, discrete-event simulator built on SimPy, using trace-driven data from real-world edge testbeds. Key components include:

- **Trace-Driven Workloads**: Profiles from edge devices (Raspberry Pi, Nvidia Jetson, Intel NUC) and workloads (AI inference, speech-to-text, matrix multiplication) ensure realistic simulations.
- **Network Simulation**: Integrates Ether, a flow-based network topology synthesizer, to model geo-distributed edge-cloud networks, balancing fidelity and performance.
- **FaaS System**: A high-level interface inspired by OpenFaaS and Kubernetes, allowing function deployment, invocation, scaling, and removal.
- **Environment**: Manages simulation components (scheduler, Load Balancer, Resource Monitor) with modular, user-configurable objects.
- **Function Simulator**: Models function life cycles (deploy, startup, setup, invoke, teardown) using trace data for accurate execution time and resource usage.

The setup supports heterogeneous edge devices, dynamic topologies, and workloads like AI-based object classification, relevant to smart city and IoT scenarios.

Real-World Validation

faas-sim’s accuracy was validated by replicating experiments on real-world testbeds:

- **Basic Data Transfer**: Compared node-to-node transfers (1MB–200MB files) on a Raspberry Pi testbed against ns-3 and Ether, achieving low error rates (<7% for sequential transfers).
- **Geo-Distributed Scenario**: Replicated an EMMA MQTT middleware experiment across cloud regions, modeling client-broker latencies with coarse-grained accuracy, capturing general system behavior.
- **Use Case Evaluations**: Applied faas-sim to resource planning, adaptation strategies, and co-simulation, using traces from devices like Jetson NX and workloads like Resnet50 inference.

This hybrid approach (simulation + testbed validation) ensures practical relevance

Evaluation Metrics

faas-sim provides comprehensive metrics, critical for internship goals:

- **Function Execution Time (FET)**: Measures latency, including cold starts and invocation delays.
- **Resource Usage**: Tracks CPU, memory, and network consumption per function invocation.
- **Data Throughput**: Evaluates system capacity for data-intensive workloads.
- **Network Usage**: Assesses data transfer performance in edge-cloud scenarios.
- **Cost**: Estimates operational expenses (implied through resource usage).
- **Performance Degradation**: Quantifies multi-tenancy interference in edge nodes.

These metrics were evaluated across scenarios like smart city and industrial IoT, comparing adaptation strategies (e.g., Skippy, Vanilla) and topologies.

Proposed Framework: faas-sim

Architecture

faas-sim's modular architecture supports flexible simulation of serverless edge platforms:

- **FaaS System**: Acts as the front-end, managing function deployment, invocation, and scaling, inspired by real-world platforms like OpenFaaS.
- **Environment**: Centralizes simulation components (Scheduler, Autoscaler, Topology), enabling customization via APIs.
- **Function Simulator**: Simulates function life cycles using trace-driven data, supporting diverse workloads and hardware.
- **Ether Integration**: Provides flow-based network simulation for realistic edge-cloud topologies.
- **Trace-Driven Model**: Uses real-world profiling data to model FET and resource usage, ensuring fidelity.

Features and Design Choices

1. **Trace-Driven Simulation**:
   - **Feature**: Relies on real-world traces from edge devices and workloads
   - **Why**: Ensures high fidelity by using empirical data, avoiding oversimplified models in CloudSim derivatives.
2. **Modular and Extensible Design**:
   - **Feature**: Allows users to replace components (like Scheduler, Load Balancer) or add custom resource models.
   - **Why**: Supports evolving edge FaaS research, enabling experimentation with new adaptation strategies.
   - **Impact**: Facilitates rapid prototyping, as seen in co-simulation-driven optimizations.
3. **Ether Network Simulation**:
   - **Feature**: Flow-based network model balances performance and fidelity, simulating geo-distributed topologies.
   - **Why**: Faster than packet-level simulators suitable for large-scale edge scenarios.
   - **Impact**: Achieves <7% error in data transfer experiments, supporting realistic network modeling.
4. **Flexible Function Modeling**:
   - **Feature**: Models OpenFaaS watchdog modes (HTTP, Fork) and supports custom function simulators.

    - **Why**: Captures real-world FaaS behaviors, like caching in HTTP mode, enhancing simulation realism.
    - **Impact**: Enables accurate simulation of diverse workloads, from AI inference to I/O-heavy tasks.
5. **Co-Simulation Support**:
    - **Feature**: Integrates with real-world systems for runtime optimization (tuning Skippy scheduler weights).
    - **Why**: Addresses dynamic edge environments by simulating future scenarios.
    - **Impact**: Improves adaptation strategies, as shown in optimized FET for AI workloads.

Results and Analysis

Accuracy

- **Results**: faas-sim replicated test experiments with high accuracy:
    - Sequential data transfers: <7% error vs ns-3 and testbed.
    - EMMA experiment: Modeled geo-distributed latencies, capturing system behavior despite coarse granularity.
- **Why**: Trace-driven data and Ether's flow-based model ensure realistic FET and network performance.
- **How**: Profiling on diverse devices (e.g., Jetson Nano, Xeon GPU) and workloads (e.g., Mobilenet inference) provides accurate inputs.

Performance

- **Results**: Simulated a smart city topology (15 edge clusters, 37,500 requests) on a developer machine (i7, 32GB RAM) in ~8 minutes, using ~2GB memory.

Use Case Evaluations

- **Resource Planning**: Estimated hardware needs for smart city and industrial IoT, showing scalable deployments with Skippy scheduler (higher throughput vs. Vanilla).
- **Adaptation Strategies**: Evaluated scheduling and load balancer placement, reducing FET by optimizing multi-tenancy interference.
- **Co-Simulation**: Optimized Skippy weights for AI workloads, improving FET by balancing data and computation movement.

Contributions

- **Novel Simulator**: faas-sim is the first trace-driven simulator for serverless edge computing, offering high fidelity and flexibility.
- **Comprehensive Metrics**: Provides FET, resource usage, throughput, network, and implied cost, surpassing cloud-centric simulators.
- **Modular Design**: Supports custom components and topologies, enabling diverse research use cases.
- **Real-World Validation**: Demonstrates accuracy via testbed replication, supporting internship validation goals.
- **Open-Source Ecosystem**: Integrates with Edgerun and Galileo, providing traces and tools for researchers.

Limitations and Future Work

- **Trace Dependency**: Requires profiling for new devices/functions, increasing setup effort.
- **Network Fidelity**: Flow-based Ether model may lack precision in complex scenarios vs. packet-level simulators.
- **Scalability**: Memory usage (~2GB) may limit very large simulations without logging optimizations.
- **Future Directions**: Embedding faas-sim in real-time control loops, integrating more IoT protocols, and enhancing network fidelity.

Critical Assessment

Strengths

- **High Fidelity**: Trace-driven approach ensures realistic edge FaaS simulations.
- **Flexibility**: Modular design supports diverse use cases and custom models.
- **Comprehensive Metrics**: Covers FET, resource usage, network, and implied cost, ideal for IoT research.
- **Open-Source**: Edgerun integration provides accessible tools and traces.

Weaknesses

- **Profiling Overhead**: Trace collection for new scenarios is resource-intensive.
- **Learning Curve**: Configurability requires familiarity with SimPy and Python.

Comparison with Prior Reviewed Simulators

Differences from MFS

- **Scope**: MFS, based on Apache OpenWhisk, focuses on cloud FaaS with partial edge support, simulating response time, cost, and rejection ratio. faas-sim targets edge FaaS, offering trace-driven FET, resource usage, and network metrics.
- **Methodology**: MFS uses synthetic workloads, while faas-sim leverages real-world traces, ensuring higher fidelity for edge scenarios.
- **Metrics**: MFS omits energy and network usage, whereas faas-sim includes these, critical for edge IoT applications.
- **Choices and Results**: MFS's static models limit adaptability, while faas-sim's modular design and co-simulation yield dynamic optimizations (e.g., reduced FET for AI workloads).
- **Contributions**: MFS provides a cloud FaaS baseline, while faas-sim advances edge research with realistic simulations.

Differences from Other Simulators

- **ServerlessSimPro**: Cloud-focused, with energy and cost metrics but limited edge support. faas-sim's edge-centric design and trace-driven approach offer broader applicability.
- **SimFaaS**: Supports cloud FaaS cost and performance but lacks edge focus and configurability. faas-sim's topology generation and adaptation support are superior.
- **EdgeFaaS**: Models edge orchestration but lacks comprehensive metrics. faas-sim's detailed FET and resource usage are more robust.
- **CloudSimSC**: General cloud simulator with basic edge extensions, oversimplifying resource models. faas-sim's trace-driven fidelity is more accurate.

Comparison with EdgeServe

- **Scope**: EdgeServe is a deployable FaaS framework, while faas-sim is a simulation tool. faas-sim complements EdgeServe by enabling pre-deployment evaluation.
- **Methodology**: EdgeServe uses iFogSim and testbeds, while faas-sim uses SimPy and Ether, offering higher configurability but no real-world deployment.
- **Metrics**: Both provide latency, cost, and energy, but faas-sim's trace-driven FET and network usage are more granular.

- **Choices and Results**: EdgeServe’s ML-based placement achieves 68–82% latency reduction, while faas-sim’s flexible adaptations optimize FET dynamically.
- **Contributions**: EdgeServe offers a practical solution, while faas-sim provides a research platform for designing such frameworks.

Edge meets FaaS

When Edge Meets FaaS: Opportunities and Challenges - Review

Computing Methodologies, Modeling and Simulation, Networks

Motivation:

The authors propose Function-as-a-Service (FaaS) as a promising abstraction for edge computing, emphasizing its potential to deliver:

- **Faster Responses:** By processing data closer to its source, FaaS reduces latency, leveraging small, resource-efficient functions that fit edge devices' limited capabilities.
- **Better Privacy:** FaaS isolates functions, minimizing the exposure of sensitive user data compared to monolithic applications.
- **Higher Productivity:** It abstracts hardware heterogeneity, enabling developers to utilize diverse edge resources effectively.
- **More Reliability:** Sandboxed functions limit the impact of failures, critical for edge devices interacting with the physical world.
- **Lower Costs:** Local execution reduces reliance on expensive cloud resources.

While FaaS platforms like AWS Lambda excel in cloud settings, they are not optimized for edge computing's unique challenges—such as resource constraints, heterogeneity, and the need for low-latency responses. The paper identifies a gap in comprehensive evaluations of FaaS in edge contexts, particularly regarding:

- The transition from application-based to function-based architectures.
- The trade-off between performance and isolation in sandbox mechanisms.
- The complexity of distributed scheduling across edge and cloud tiers.

Existing studies and platforms lack detailed insights into these areas, leaving the feasibility and optimization of FaaS-based edge computing underexplored.

Methodology

The authors evaluate FaaS-based edge computing by analyzing two platforms: AWS IoT Greengrass and OpenFaaS . Experiments are conducted on real hardware—Raspberry Pi 3B+ devices and an edge server—to assess practical performance. The study focuses on three challenges:

Sandbox Mechanisms:

The paper tests three AWS Greengrass sandbox options:

- **Greengrass Container (GGC):** Lightweight, using cgroups and namespaces.
- **Docker Container (DOCK):** Heavier, offering stronger isolation.
- **Greengrass No Container (GNC):** Minimal overhead, minimal isolation.
  These are evaluated for runtime overhead and isolation under stress tests.

Distributed Scheduling:

A custom OpenFaaS prototype implements scheduling strategies:

- **Edge-Only:** Scales functions across local devices.
- **Edge-Cloud Cooperative:** Offloads a proportion of requests to a server.
- **Cloud-Only:** Executes all functions on a server.

Experiments measure latency under varying loads.

Experiments :

- **Sandbox Mechanisms:** GGC, DOCK, and GNC are tested for overhead and isolation.
- **Function Chaining:** Sequential execution overhead, including cold starts, is assessed.
- **Distributed Scheduling:** A prototype explores edge-only, cooperative, and cloud-only strategies.

Metrics Collected

- **Performance:**
    - Sandbox overhead (3.8% for GNC, 78.3% for DOCK).
    - Cold start impact (5.3x runtime increase) vs. warm containers (negligible overhead).
    - Scheduling latency (0.86s for edge-only vs. 0.44s for cloud-only).
- **Isolation:**
    - GGC doubles runtime under stress; DOCK reduces interference to 40%.
- **Scheduling:**
    - Latency decreases as cloud offloading increases (25% to 75%).

Findings & Contributions

Realistic Evaluation

Using real edge devices and servers, the paper provides concrete insights into FaaS performance, avoiding the abstractions of simulation.

Key Insights

- **Sandbox Trade-offs:** Lightweight GGC and GNC offer low overhead (3.8–4.2%) but poor isolation; DOCK's 78.3% overhead ensures better security.
- **Function Chaining:** Cold starts significantly inflate runtime (5.3x), while warm containers perform efficiently.
- **Scheduling Benefits:** Offloading to the cloud reduces latency (from 0.86s to 0.44s), demonstrating vertical scaling's value.

Opportunities

- **Improved Latency and Bandwidth Efficiency:** Co-locating compute with data sources reduces round-trip time and traffic to the cloud.
- **Context-Awareness and Locality:** Utilizing local contextual information for better service personalization and quality.
- **Elastic and Event-Driven Edge Processing:** FaaS abstracts lifecycle management and autoscaling, which could simplify deployment at the edge.
- **Heterogeneous Resource Utilization:** Function granularity allows optimal usage of diverse and constrained edge resources.

Challenges

Resource Management:

- Edge resources are limited and heterogeneous, unlike the cloud.
- Issues include fine-grained scheduling, pla cement, and elasticity across dynamic topologies.

Function Deployment and Orchestration:

- Cold start latency is much worse at the edge.
- Function migration and orchestration across multiple edge nodes are complex due to decentralization.

Security and Privacy:

- New attack surfaces arise from multi-tenancy and limited isolation.

Programming Model and Abstractions:

- Lack of programming models tailored for distributed, dynamic, and latency-sensitive FaaS at the edge.

- Trade-offs between usability (for devs), granularity, and deployment complexity remain unresolved.

Monitoring and Debugging:

- Fine-grained observability is hard due to ephemeral, distributed function executions.

Limitations & Future Work

- **Data Abstraction :** The evaluation omits data flow and bandwidth modeling, critical for network-constrained edge scenarios.
- **Enhanced Scheduling Policies:** The basic scheduling prototype suggests potential, but advanced algorithms could optimize latency and cost further.
- **Isolation Mechanisms:** The performance-isolation trade-off remains unresolved, especially for time-sensitive applications requiring both speed and security.

⭐ FaaS Edge Cloud FW

Serverless Computing in the Edge-Cloud Continuum : Challenges, Opportunities, and a Novel Framework - Review

Edge-Cloud Continuum, Serverless Computing, Distributed Function Placement, IoT Application Optimization, Adaptive Resource Management

Validates the idea of the internship in future trends anticipated

🟢 **Comprehensive framework & test + Case studies** 🔴 **Complex implementations**

Motivation:

IoT devices and the need for **real-time processing** have exposed the limitations of cloud-centric serverless models, which suffer from **high latency** and **costs** due to data transfers. Edge computing addresses these by processing data locally, but existing serverless platforms (AWS Lambda…) are not optimized for edge constraints like **device heterogeneity**, **intermittent connectivity**, and limited resources. The Author identifies critical challenges:

- **Resource Management**: Coordinating diverse edge and cloud resources.
- **Data Consistency**: Maintaining state in distributed environments.
- **Function Placement**: Optimizing function execution locations.
- **Security and Privacy**: Safeguarding data in distributed settings.

Methodology

Proposed Framework: EdgeServe

Architecture

EdgeServe's three-layer architecture was designed to leverage the strengths of edge, fog, and cloud:

- **Edge Layer**: Includes IoT sensors and local servers for low-latency processing.
- **Intermediate Layer**: Fog nodes (e.g., 5G base stations) for regional coordination.
- **Cloud Layer**: Data centers for compute-intensive tasks.

A distributed orchestrator, inspired by prior hybrid models (AWS Greengrass), coordinates execution, chosen for its ability to handle dynamic workloads across heterogeneous environments.

Key Components and Design Choices

1. **Resource Management**:
   - **Method**: Combines static profiling (initial resource assessment) and dynamic monitoring (real-time updates) to allocate resources.
   - **Why :** Static profiling ensures baseline compatibility, while dynamic monitoring adapts to workload changes, addressing edge device variability.
   - **Impact**: Enables balanced workload distribution, reducing resource waste and improving throughput.
2. **Data Consistency**:
   - **Method**: Uses multi-level caching and a lightweight consensus protocol, supporting strong or eventual consistency.
   - **Why**: Caching minimizes latency, and the consensus protocol ensures reliability without heavy network overhead, suitable for intermittent edge connectivity.
   - **Impact**: Maintains state consistency, critical for IoT applications like traffic management.

3. **Function Placement**:
   - **Method**: **Employs a machine learning (ML) algorithm** to predict optimal function placement based on proximity, network conditions, and latency requirements.
   - **Why** : ML enables adaptive, data-driven decisions, learning from past executions to optimize future placements, unlike static rules in prior works.

    - **Impact**: Reduces latency by placing functions closer to data sources, as seen in the 82% latency reduction for smart city tasks.
4. **Security/Privacy**:
    - **Method**: Implements end-to-end encryption, secure enclaves, differential privacy (ε=0.1), and distributed authentication.
    - **Why** : Enclaves and encryption address edge vulnerabilities, while differential privacy balances data utility and protection, critical for smart city data.
    - **Impact**: Achieves zero security breaches and robust privacy, enhancing trust in distributed systems.
5. **Development Tools**:
    - **Method**: Provides a unified API, simulation environment, and monitoring dashboard, integrated with CI/CD pipelines.
    - **Why** : Simplifies development for diverse edge-cloud scenarios, aligning with serverless principles of abstraction.
    - **Impact**: Streamlines deployment, supporting rapid prototyping for IoT applications.

Simulation Setup to test performance under different conditions

EdgeServe's performance is evaluated using *iFogSim* for modeling IoT, edge, and fog environments. The simulation includes:

- **Diverse Devices**: Smartphones, Raspberry Pis, and industrial IoT sensors to reflect edge heterogeneity.
- **Network Topologies**: Urban (high connectivity) and rural (limited connectivity) scenarios to test adaptability.
- **Workloads**: Lightweight sensor data processing to compute-intensive machine learning tasks, ensuring comprehensive performance assessment.

The choice of iFogSim over other simulators (CloudSim) was driven by its ability to model edge-specific constraints like network variability and device limitations, critical for realistic IoT simulations.

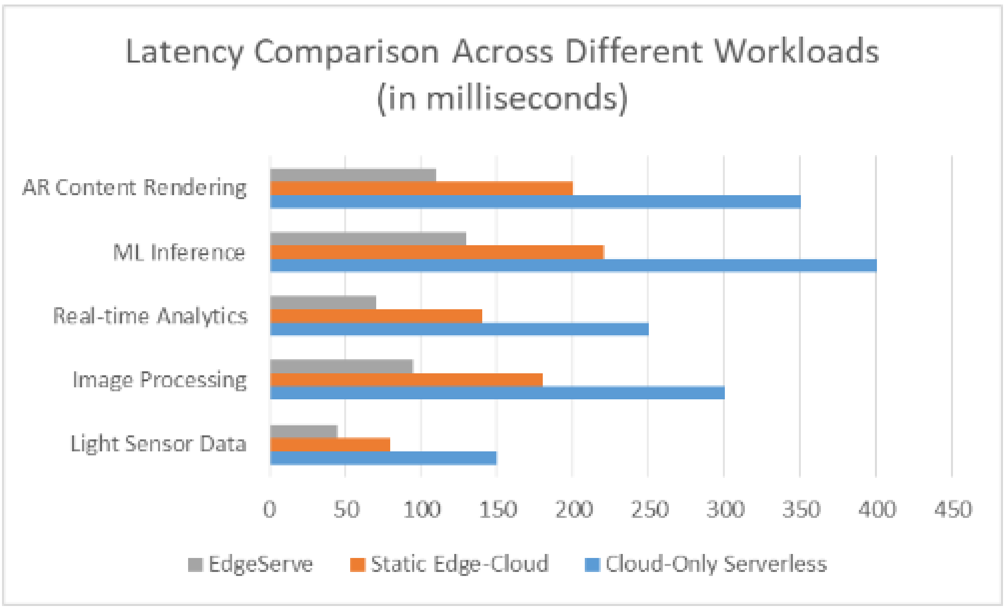
Real-World Case Studies

Three case studies validate EdgeServe in practical scenarios:

1. **Smart City Traffic Management**: Processes real-time sensor data for traffic light control, prioritizing low latency.
2. **Industrial IoT Predictive Maintenance**: Monitors machinery health, emphasizing scalability.
3. **Mobile Augmented Reality Gaming**: Delivers immersive experiences, focusing on user responsiveness.

Results and Analysis:

Latency :



- **Why**: The ML-based function placement algorithm prioritizes edge execution for latency-sensitive tasks, minimizing data transfer delays. For example, traffic light control functions were executed on local servers, avoiding cloud round-trips.
- **How**: Dynamic monitoring ensured functions were reassigned to fog or cloud nodes during network congestion, maintaining responsiveness.

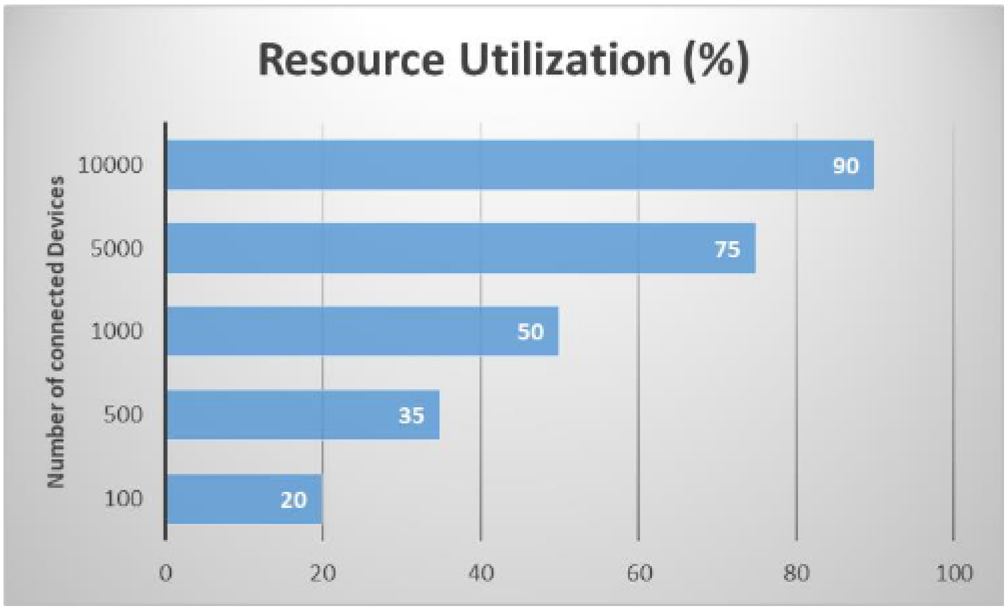
Scalability

- **Results**: Handled a 10-fold device increase with 35% higher throughput during peak loads.
- **Why**: The hierarchical resource management dynamically offloaded tasks to cloud resources during spikes, preventing edge node overload.
- **How**: The orchestrator's real-time resource view enabled proactive scaling, unlike static edge-cloud setups.

Cost

- **Results**: Achieved 43% average cost reduction, with **61% savings in the smart city case.**
- **Why**: Local processing reduced cloud compute and data transfer costs, particularly for high-volume sensor data.
- **How**: The allocation algorithm optimized resource use, minimizing active cloud instances.

Energy Efficiency



- **Results**: 28% energy reduction, notably in IoT scenarios.
- **Why**: Offloading to energy-efficient edge devices and reducing data transmission lowered power consumption.
- **How**: Dynamic monitoring powered down idle nodes, and ML placement favored low-power devices for lightweight tasks.

Security/Privacy

- **Results**: Zero breaches, authentication latency <50ms, ε=0.1 differential privacy.
- **Why**: Secure enclaves isolated functions, and differential privacy protected aggregated data.
- **How**: Distributed authentication ensured rapid, secure access, while encryption safeguarded data in transit.

Metric	Cloud-Only Serverless	Static Edge-Cloud	EdgeServe
Average Latency Reduction	Baseline	35%	68%
Cost Reduction	Baseline	20%	43%
Energy Efficiency Gain	Baseline	15%	28%
Scalability (Max Devices)	1000	5000	10000
Security Breaches	3	1	0

Contributions

- **Novel Framework**: EdgeServe integrates edge-cloud serverless computing with adaptive, ML-driven placement, unlike platform evaluations in prior works.
- **Performance Gains**: Achieves 68–82% latency reduction, 43% cost savings, and 28% energy efficiency, validated across IoT use cases.
- **Real-World Validation**: Case studies provide practical insights, directly supporting internship goals.
- **Developer Tools**: Unified API and CI/CD integration enhance usability for IoT deployments.

Critical Assessment

Strengths

- **Comprehensive Framework**: EdgeServe addresses a wide range of edge-cloud challenges, surpassing the scope of prior works.
- **Robust Validation**: Combines simulations and real testbeds, enhancing credibility for internship validation tasks.
- **Practical Relevance**: IoT-focused case studies and detailed metrics directly support internship goals.

Weaknesses

- **Complexity**: The framework's sophistication may complicate implementation for resource-constrained teams

Faas-sim implementation

Problem : For trace driven energy simulation, only a factor is available so we can't really benefit to the strong point of the simulator, but we can calculate is separately

Current Status

I have installed the simulator and solved the dependencies problems, executed the examples and i am creating my own example and implemented additional tooling for data collection and visualization: also forked the repository and added all the changes i'm doing faas-sim

Created a data collection script that exports simulation metrics to CSV files in the analysis_data directory

Developed a visualization script that generates:
- Function execution time distributions
- Timeline views of function executions by node
- Cumulative invocation graphs
- Wait time vs execution time analysis

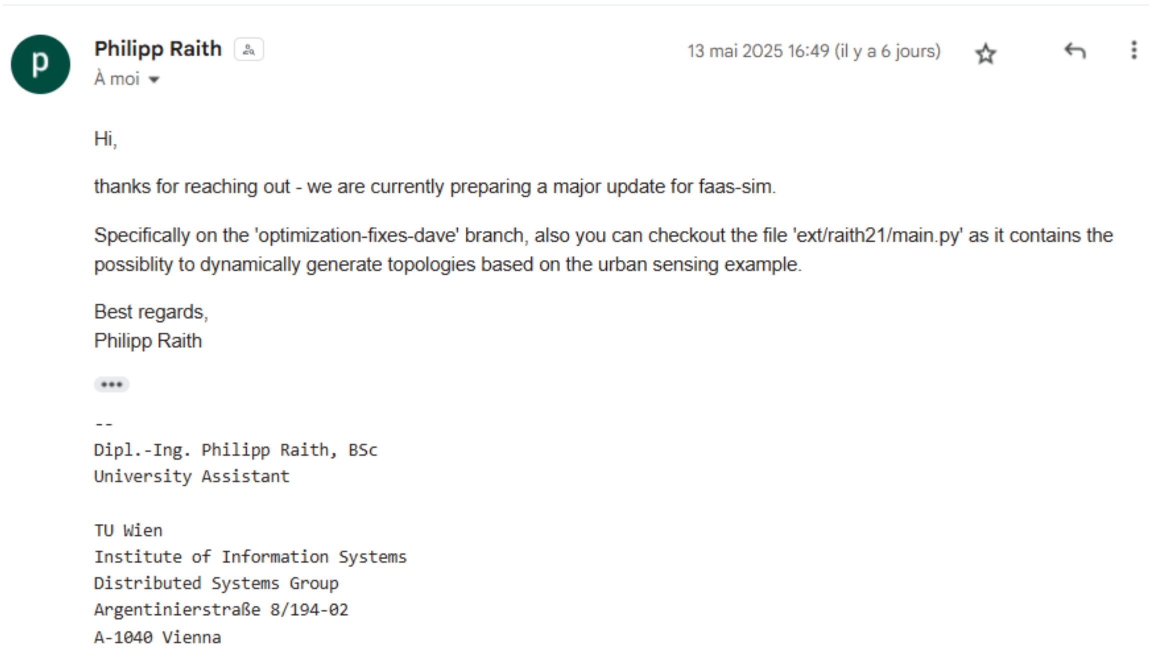
Current Issues & Next Steps

Issues

1. Time unit inconsistencies:
   - SimPy expects seconds
   - Metrics system records in milliseconds
   - Leading to unrealistic execution times (~0.17ms average)

2. Unrealistically Resource Utilization:
- low CPU usage (0.25%)
- Single node execution (only server_0 active)
- Unrealistic execution times:
    - Getting 0.05-0.58ms when should be 50-500ms
    - ResNet50: 0.06ms avg (should be 30-500ms)
    - Python Pi: 0.30ms avg (should be 50-200ms)
- Very low CPU utilization (0.25% avg, 1% peak)
- Limited network activity
- No proper scaling behavior observed

After getting stuck on trying to change around the topology and the images being executed, i got into contact with the authors and they guided me:


Philipp Raith
À moi

13 mai 2025 16:49 (il y a 6 jours)

Hi,

thanks for reaching out - we are currently preparing a major update for faas-sim.

Specifically on the 'optimization-fixes-dave' branch, also you can checkout the file 'ext/raith21/main.py' as it contains the possiblity to dynamically generate topologies based on the urban sensing example.

Best regards,
Philipp Raith

```
--
Dipl.-Ing. Philipp Raith, BSc
University Assistant

TU Wien
Institute of Information Systems
Distributed Systems Group
Argentinierstraße 8/194-02
A-1040 Vienna
```

now i will be working with the “optimization-fixes-dave” branch and the “raith21” example which is much more sophisticated and more realistic than the examples in the main branch i was using, so understanding how it works would make it easier to work with in the long run.

Work in Progress

1. Implementing the topology correctly:
   - Execution on edge nodes & server nodes with GPU capability
   - Proper network routing between nodes
   - Debugging node distribution issues
2. Fixing time unit handling to get more realistic execution times:
   - Python Pi: Expected 50-200ms
   - ResNet50: Expected 30-500ms (GPU/CPU dependent)

3. Adding proper network transfer simulation between nodes

Framework Understanding

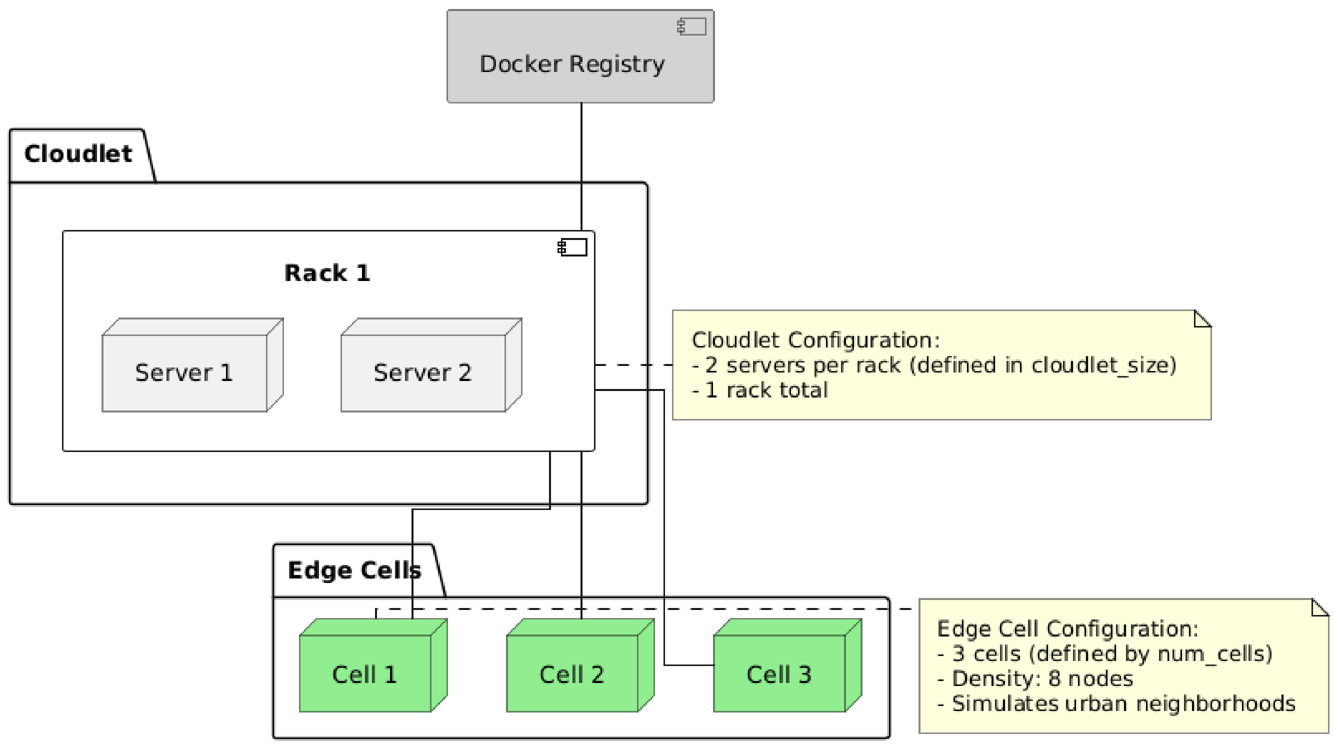
While I don't fully understand all internal mechanics, I have a good grasp of:
- Basic simulation capabilities
- Metric collection system
- Visualization requirements

Initial example modified by me : Code available here

Urban Sensing Topology : ether/scenarios/urbansensing.py

Overview

The Urban Sensing topology is a simulated edge-cloud environment that models a smart city deployment with multiple neighborhoods, IoT devices, edge computing resources, and a central cloudlet. This topology is used to simulate function deployments and executions in a geographically distributed environment.



Urban Sensing Scenario

Key Parameters

- Number of Cells: 3 neighborhoods
- Cell Density:
  - 8 edge nodes
  - This creates a realistic population density distribution
- Cloudlet Size: (2 , 1) configuration
  - 2 servers per rack
  - 1 rack

Networking

- Uses default Docker registry initialization (t.init_docker_registry())
- Registry is connected to internet nodes automatically

Architecture Layers

The topology creates a hierarchical structure:

1. Cloud Layer
   - Contains the primary compute servers
   - Organized in racks with servers each
2. Edge Layer
   - 3 neighborhood cells
   - Density can be fixed or variable

    - Connected via shared network links
3. Network Layer
    - Docker registry for container distribution
    - Internet connectivity for external access

Implementation Details

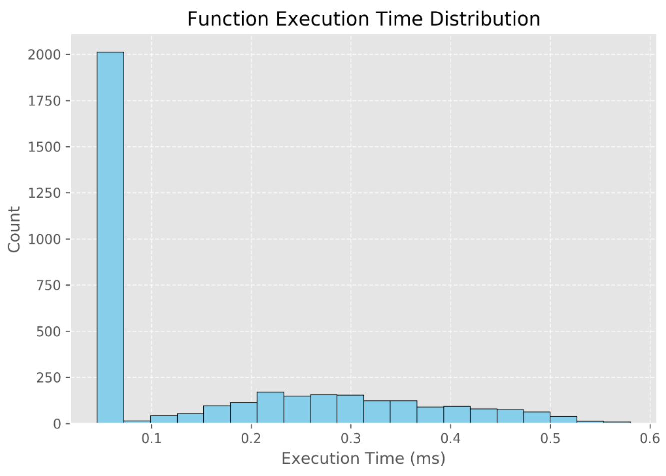
This topology is instantiated through the UrbanSensingScenario class from the ether.scenarios.urbansensing module. The scenario is materialized into a Topology object via:

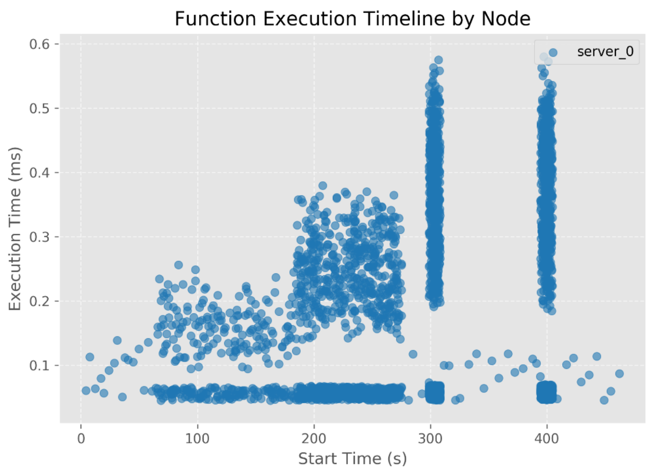
```
urban_scenario = scenario.UrbanSensingScenario(

  num_cells=3,                         # 5 neighborhoods

  cell_density=ParameterizedDistribution.fixed(8), # 8 nodes per neighborhood

  cloudlet_size=(2, 1)                   # 2 servers per rack, 1rack

)

urban_scenario.materialize(t)  # Apply to topology object t

t.init_docker_registry()      # Initialize container registry
```

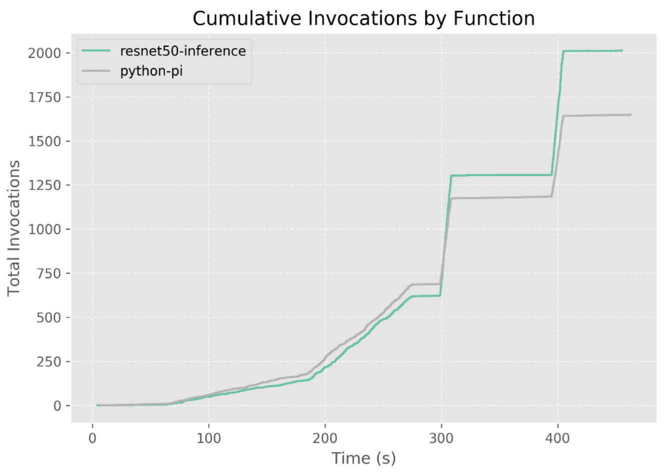
The materialize() method adds both the city and cloudlet components to the topology, creating all the necessary nodes and connections. This provides a realistic environment for simulating edge-cloud function deployments and executions.

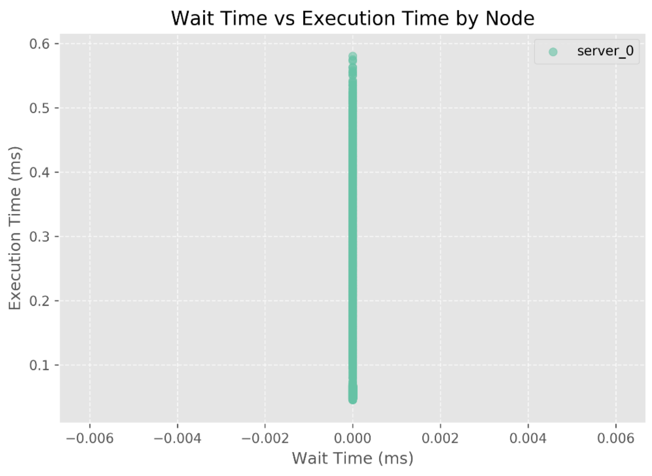
– Not close to accurate simulation but we will get there :

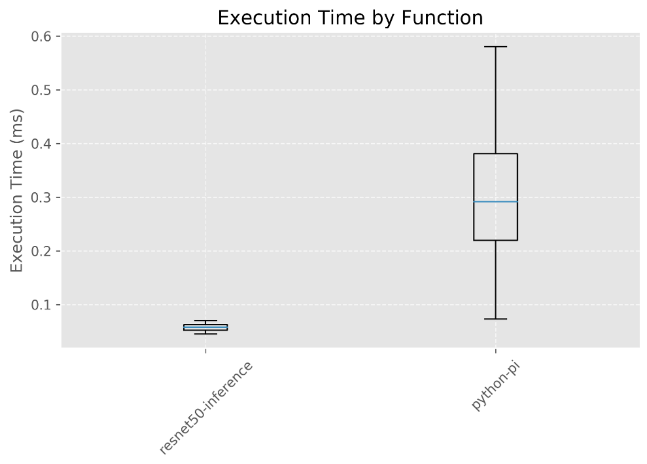
```
FAAS SIMULATION SUMMARY
======================

Report generated: 2025-05-12 16:00:14

Topology Overview:
  Number of nodes: 1
  Number of functions: 2
  Number of images: 2

Function Distribution:
  resnet50-inference: 2012 invocations (55.0%)
  python-pi: 1648 invocations (45.0%)

Overall Invocation Statistics:
  Total invocations: 3660
  Avg execution time: 0.17 ms
  Max execution time: 0.58 ms
  Avg wait time: 0.00 ms
  Max wait time: 0.00 ms
  Simulation duration: 458.02 s

Resource Utilization:
  Overall avg CPU utilization: 0.25%
  Overall peak CPU utilization: 1.00%

Network Activity:
  Total transfers: 2
  Total bytes transferred: 108.72 MB
  Transfer duration: 1.87 s

Network Flows:
  registry → server_0: 108.72 MB in 1.87 s
```

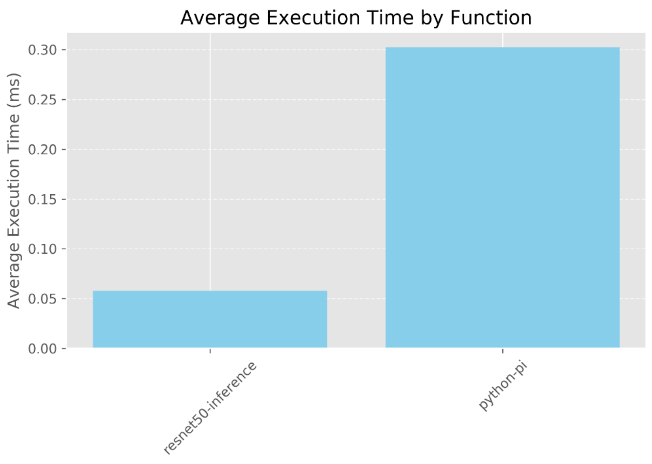

Function Execution Time Distribution
Count
Execution Time (ms)

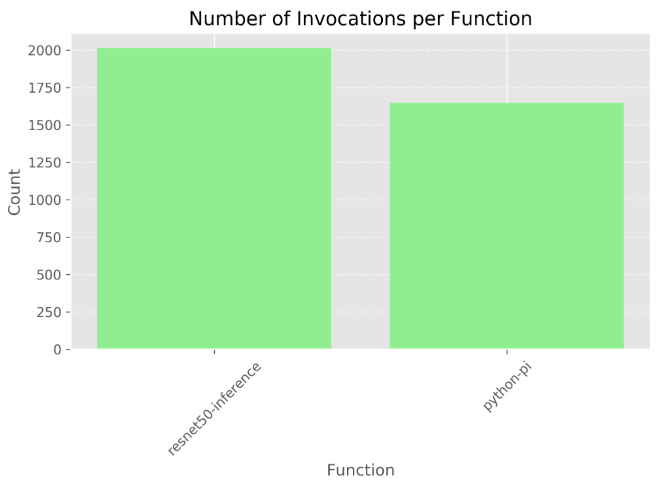

Function Execution Timeline by Node
server_0
Execution Time (ms)
Start Time (s)

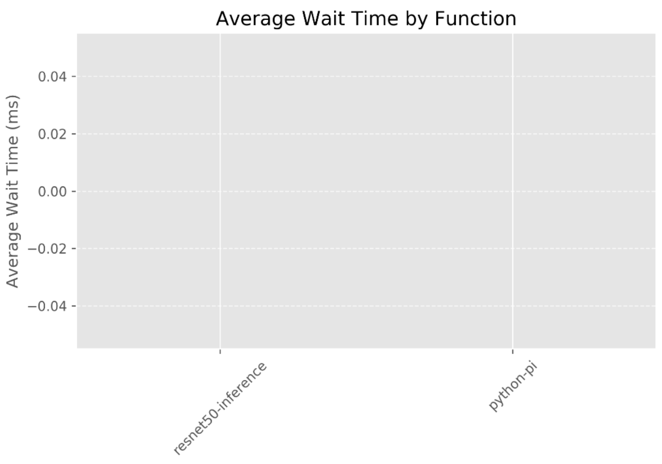

Cumulative Invocations by Function
resnet50-inference
python-pi
Total Invocations
Time (s)

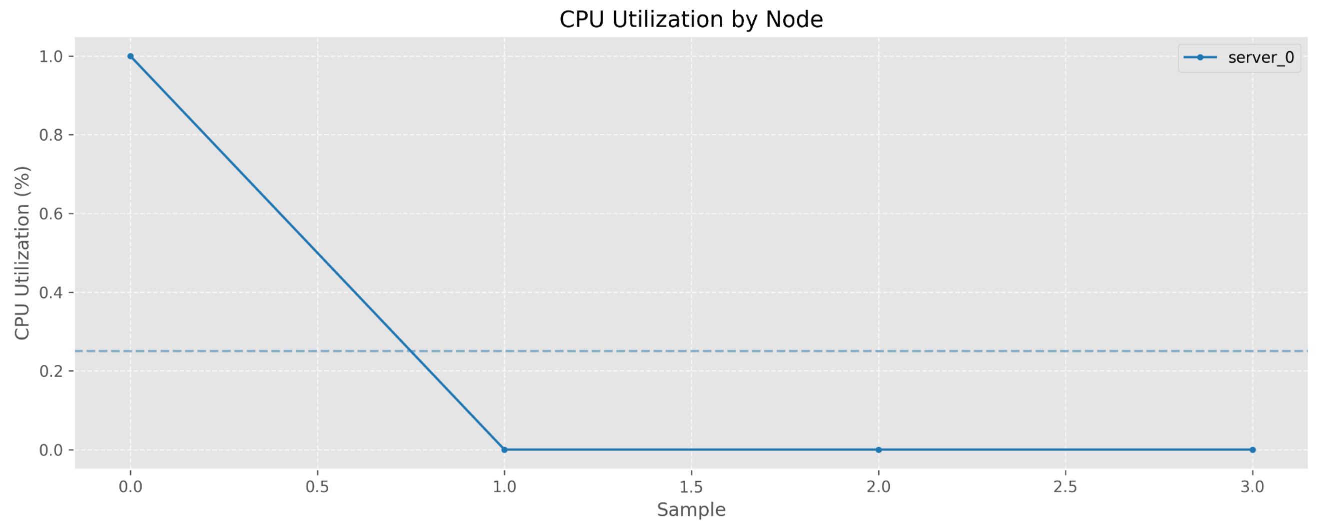

Wait Time vs Execution Time by Node
server_0
Execution Time (ms)
Wait Time (ms)

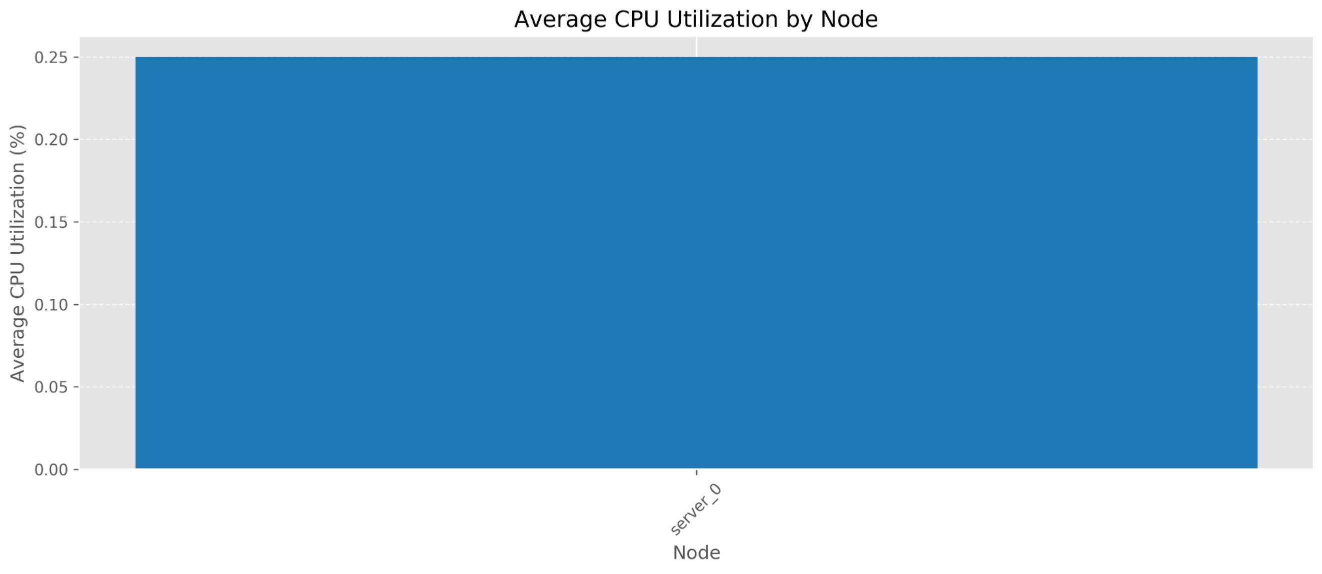

Execution Time by Function
Execution Time (ms)
resnet50-inference
python-pi

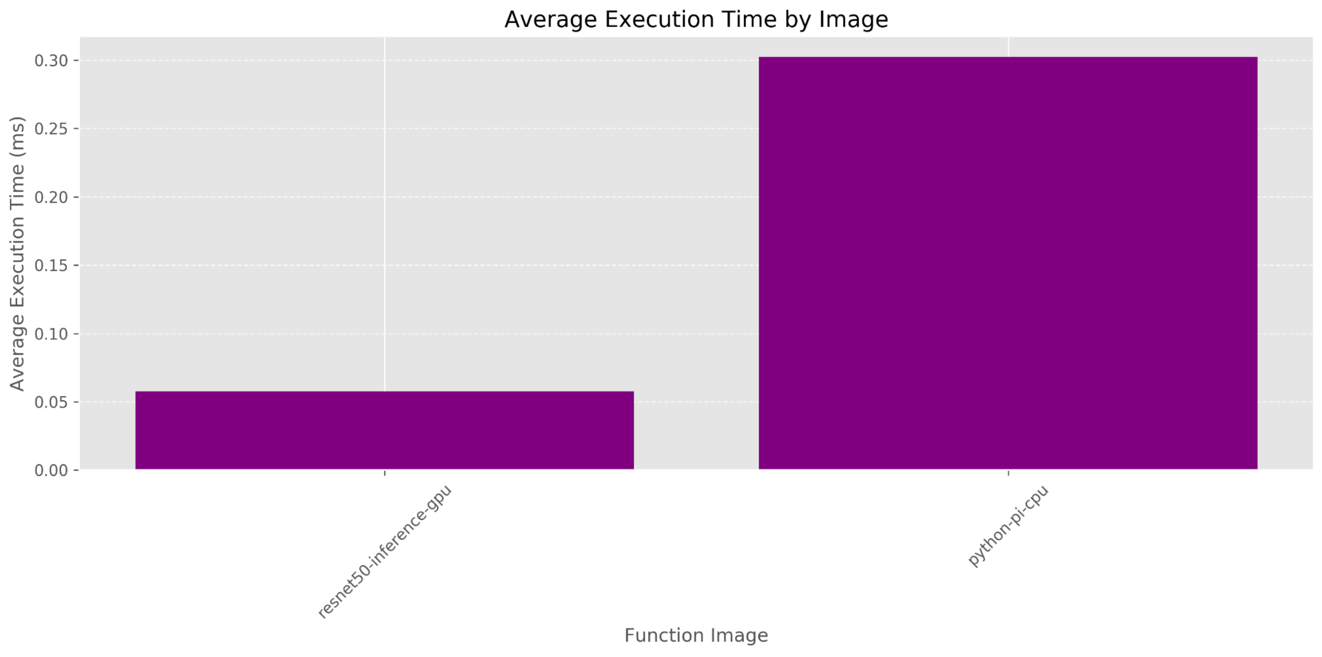

Average Execution Time by Function
Average Execution Time (ms)
resnet50-inference
python-pi


Number of Invocations per Function
Count
resnet50-inference
python-pi
Function


Average Wait Time by Function
Average Wait Time (ms)
resnet50-inference
python-pi







CSVs generated : https://github.com/M4hf0d/faas-sim/tree/master/analysis_data
raith21 example :

Urban Sensing Topology : ether/scenarios/urbansensing.py

- 400 Heterogeneous Devices: Generated using the generate_devices() function with cloudcpu_settings
    - Edge Computing Devices:
        - NVIDIA Jetson family: tx2, nx, nano
        - Intel: nuc mini PCs
        - Raspberry Pi: rpi3, rpi4
        - Other ARM devices: rockpi, coral (with TPU)
- Multi-level Hierarchy:
    - Cloudlet: High-performance Xeon servers (both CPU and GPU variants)
    - Neighborhoods: Connected via shared 10 Gbps links
    - Edge Devices: Various edge computing devices

Network Organization

The devices are organized in a hierarchical urban sensing scenario:
class HeterogeneousUrbanSensingScenario(UrbanSensingScenario):
Connection Types:

- FasterMobileConnection: 250 Mbps for edge devices
- FiberToExchange: Higher bandwidth for cloud connections
- SharedLinkCell: 10 Gbps shared links within neighborhoods

Energy Profiles

Each device type has specific energy consumption characteristics defined in the resources.py file:

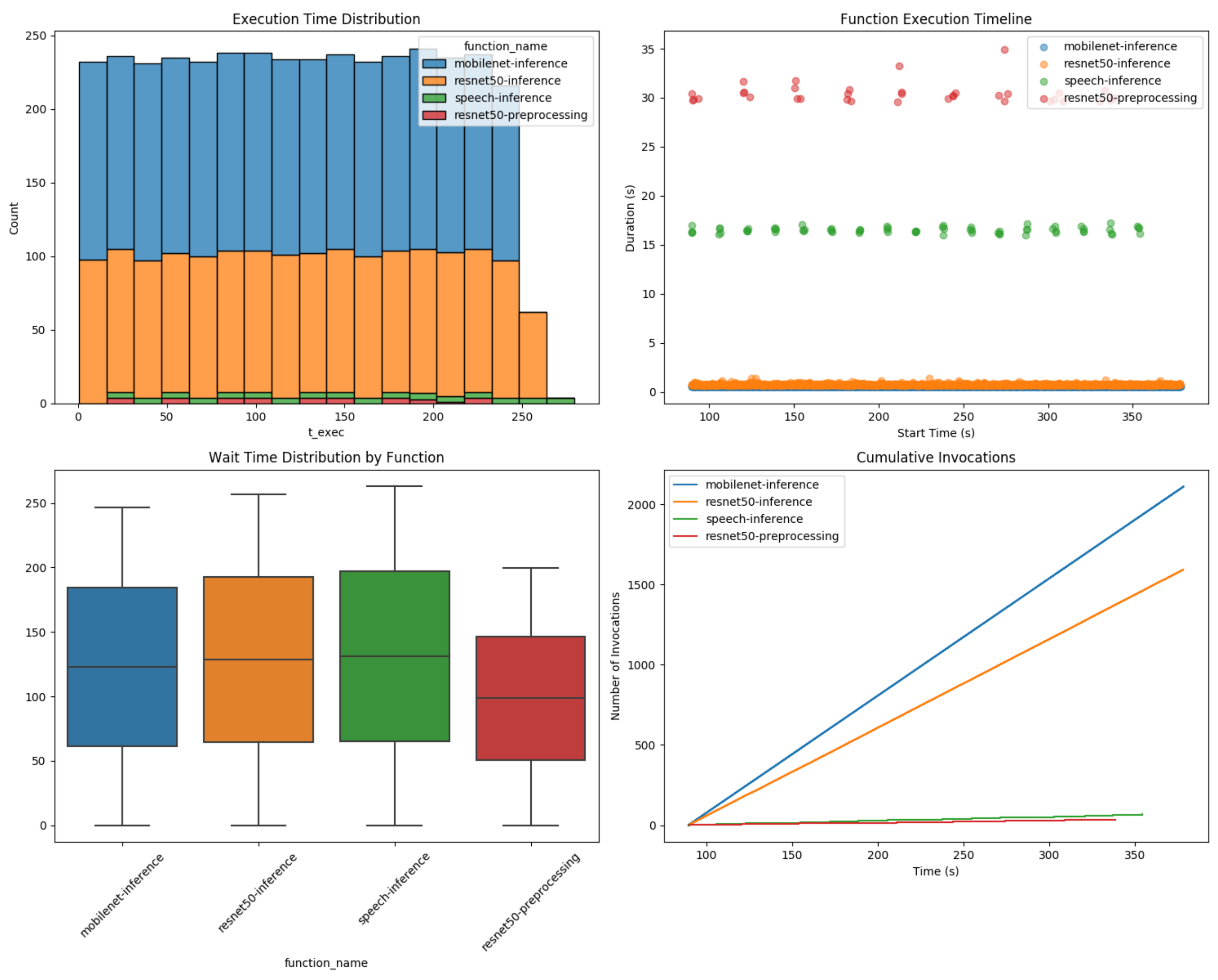
```
# Example energy profile
('rpi3', 'faas-workloads/resnet-inference-cpu'): FunctionResourceCharacterization(
    0.5448381974407112,  # CPU utilization
    2615.734632324097,   # Network in KB/s
    0,                   # GPU utilization
    2062539.5723288062,  # Network out bytes/s
    0.36519032690251113
)
```

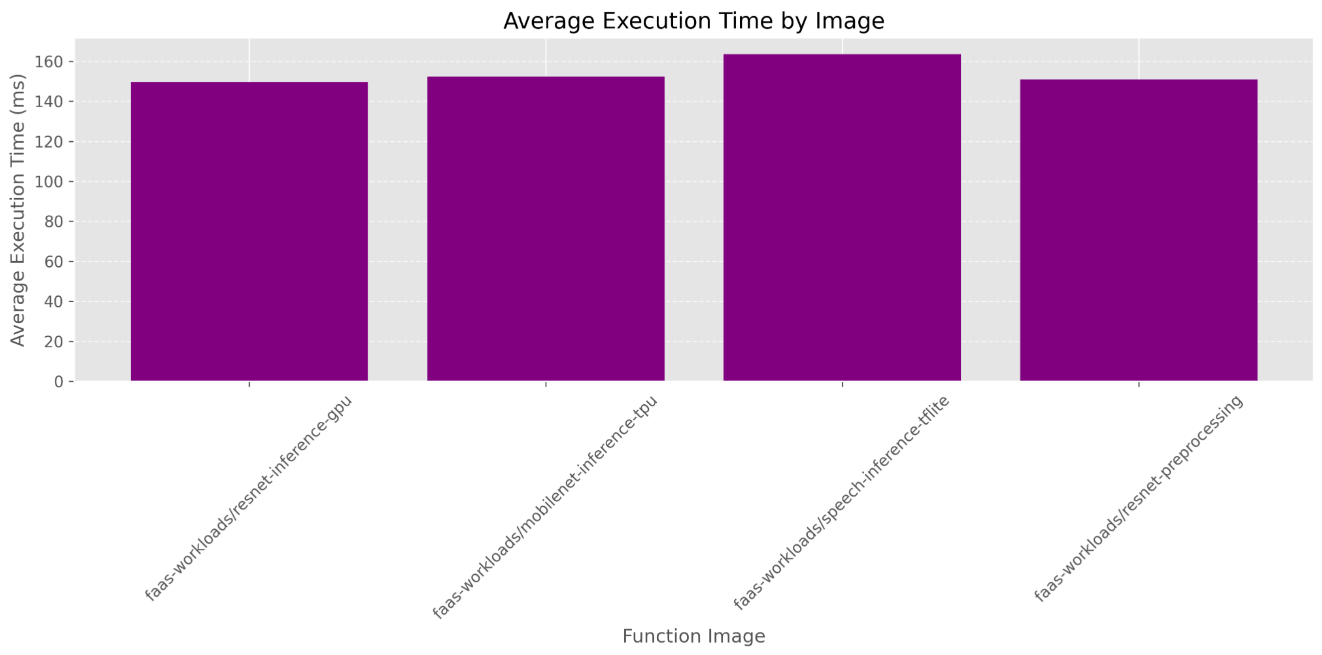
Device Generation Settings

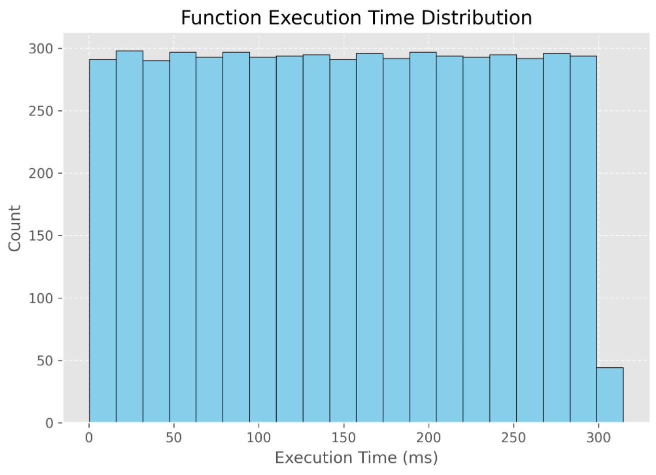
The cloudcpu_settings in generators/cloudcpu.py controls the distribution of device types with parameters for:

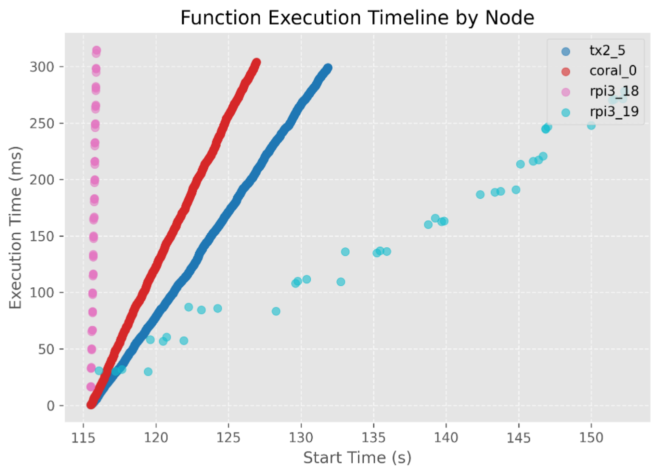
- Location (CLOUD, EDGE, MOBILE)
- Network connectivity (ETHERNET, WIFI, MOBILE)
- CPU model and power
- RAM capacity
- GPU availability and type

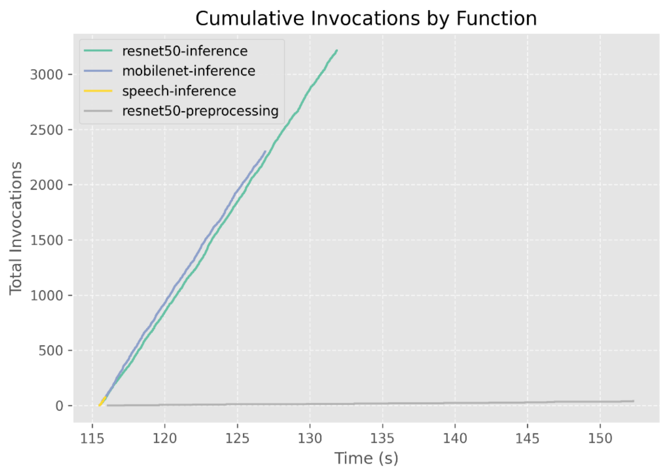
```
FAAS SIMULATION SUMMARY
======================

Report generated: 2025-05-19 05:41:45

Topology Overview:
  Number of nodes: 4
  Number of functions: 4
  Number of images: 4

Node Distribution:
  tx2_5: 3215 invocations (57.1%)
  coral_0: 2302 invocations (40.9%)
  rpi3_18: 75 invocations (1.3%)
  rpi3_19: 40 invocations (0.7%)

Function Distribution:
  resnet50-inference: 3215 invocations (57.1%)
  mobilenet-inference: 2302 invocations (40.9%)
  speech-inference: 75 invocations (1.3%)
  resnet50-preprocessing: 40 invocations (0.7%)

Overall Invocation Statistics:
  Total invocations: 5632
  Avg execution time: 150.88 ms
  Max execution time: 314.60 ms
  Avg wait time: 0.00 ms
  Max wait time: 0.00 ms
  Simulation duration: 36.79 s

Resource Utilization:
  Overall avg CPU utilization: 0.25%
  Overall peak CPU utilization: 1.00%


Network Activity:
  Total transfers: 12
  Total bytes transferred: 4060.75 MB
  Transfer duration: 403.22 s

Network Flows:
  nuc_4 → coral_0: 3.81 MB in 0.14 s
  nuc_4 → nano_10: 221.25 MB in 7.67 s
  nuc_4 → rpi3_18: 45.78 MB in 6.34 s
  nuc_4 → tx2_5: 98.23 MB in 9.09 s
  registry → coral_0: 164.99 MB in 5.71 s
  registry → nano_10: 953.67 MB in 111.36 s
  registry → rpi3_18: 312.81 MB in 43.28 s
  registry → rpi3_19: 1306.53 MB in 115.15 s
  registry → tx2_5: 953.67 MB in 104.50 s
```

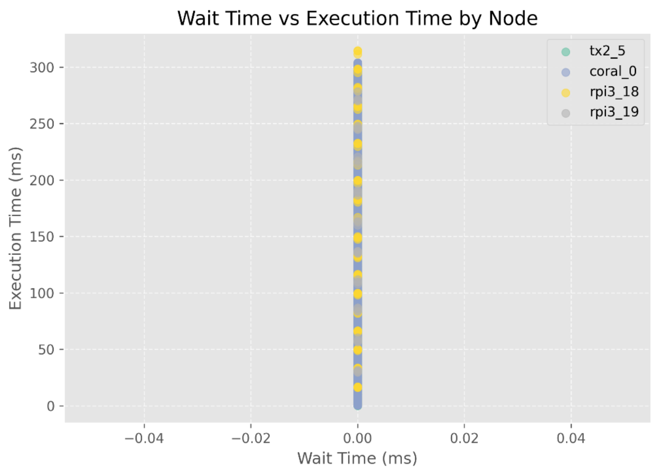

Execution Time Distribution
Count
t_exec
function_name
mobilenet-inference
resnet50-inference
speech-inference
resnet50-preprocessing
Function Execution Timeline
Duration (s)
Start Time (s)
mobilenet-inference
resnet50-inference
speech-inference
resnet50-preprocessing
Wait Time Distribution by Function
mobilenet-inference
resnet50-inference
speech-inference
resnet50-preprocessing
function_name
Cumulative Invocations
Number of Invocations
Time (s)
mobilenet-inference
resnet50-inference
speech-inference
resnet50-preprocessing


Average Execution Time by Image
Average Execution Time (ms)
faas-workloads/resnet-inference-gpu
faas-workloads/mobilenet-inference-tpu
faas-workloads/speech-inference-tflite
faas-workloads/resnet-preprocessing
Function Image









More here :

https://github.com/M4hf0d/faas-sim/tree/master/visualization_results32

λContSim implementation

Problem : Custom functions are written in prolog

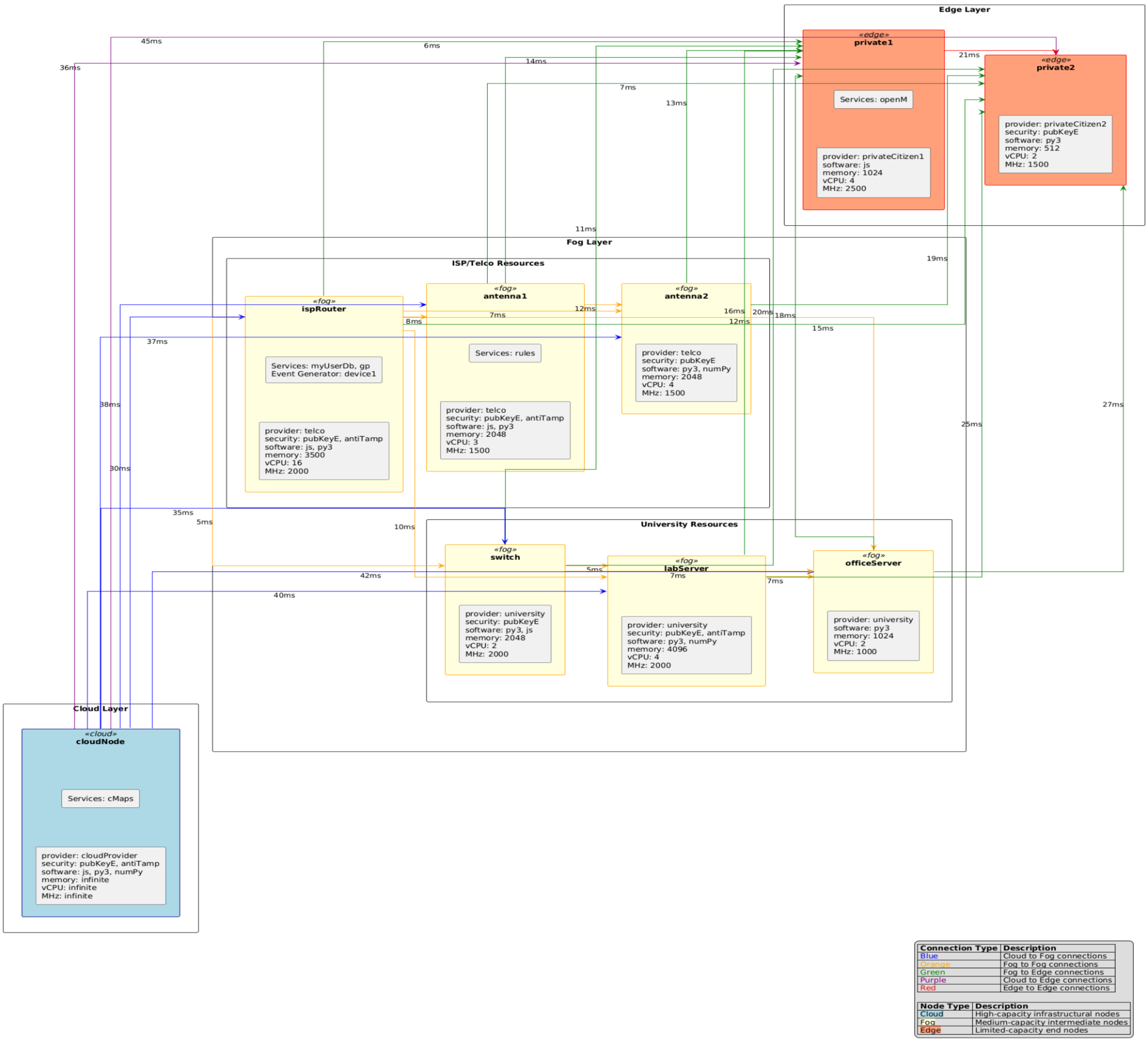
Current Status

installation and execution was a lot simpler than faas-sim, i’m implementing my own topology and implemented additional tooling for data collection and visualization: also forked the repository and added all the changes i’m doing **LambaContSim Repository**

same process, understanding fundamental components & how they work and contribute to the bigger picture also ran initial example and collected into CSV files then used that to plot and visualize what happened in the simulation,
you can find it here :

- Exports : https://github.com/M4hf0d/LambdaContSim/tree/main/csv_exports
- app success rate, node load overtime …..and more https://github.com/M4hf0d/LambdaContSim/tree/main/analysis_results
- Network and infrastructure https://lfs-vis.vercel.app/

High resolution version



Connection Type	Description
Blue	Cloud to Fog connections
Orange	Fog to Fog connections
Green	Fog to Edge connections
Purple	Cloud to Edge connections
Red	Edge to Edge connections

Node Type	Description
Cloud	High-capacity infrastructural nodes
Fog	Medium-capacity intermediate nodes
Edge	Limited-capacity end nodes

Work in Progress

1. Diving Deeper into the components

2. Playing around with different topologies

Guide

Overview

LambdaFogSim is a simulator designed to model and analyze serverless (lambda) function execution across fog computing environments. It enables you to simulate how applications perform in distributed edge/fog infrastructures with various constraints and conditions.

Core Components

1. Infrastructure Layer

The simulator supports two types of infrastructure:

- **Physical Infrastructure**: Defined in YAML format (infrastructure_config.yaml), representing a fog computing environment with nodes at different tiers
- **Logical Infrastructure**: Defined in Prolog format (.pl files), providing a more abstract representation

Each infrastructure consists of:

- **Nodes**: Computing devices with properties like CPU, memory, and security capabilities
- **Links**: Network connections between nodes with properties like latency and bandwidth
- **Event Generators**: Sources that trigger application execution

2. Application Model

Applications are represented as collections of lambda functions with dependencies:

- **Functions**: Individual serverless functions with resource requirements
- **Dependencies**: Data flows between functions
- **QoS Requirements**: Performance and security constraints

3. Placement Engine

The placement engine (in placer.pl) determines where to deploy functions:

```
# placement.py interfaces with the Prolog engine

raw_placement, execution_time = get_raw_placement(

    placement_type=placement_type,

    orchestration_id=config.applications[application_name]["orchestration_id"],

    generator_id=generator_id,

)
```

The engine uses constraint logic programming to find optimal placements considering:

- Resource constraints
- Security requirements
- QoS needs

4. Simulation Engine

Based on SimPy, the simulation engine (in lambdafogsim.py) orchestrates the simulation:

```
# Start simulation with SimPy

env = simpy.Environment()

env.process(simulation(env, config.sim_num_of_epochs, infrastructure))
env.run(until=config.sim_num_of_epochs)
```

Key features:

- Discrete event simulation
- Function execution modeling
- Network transfer simulation
- Resource monitoring
- Fault injection (node/link failures)

5. Configuration System

The configuration system (config.py) allows customization of:

```
# SIMULATOR parameters

sim_num_of_epochs          # Duration of simulation

sim_function_duration      # Execution time for functions

sim_seed                   # Random seed for reproducibility

sim_use_padding            # Whether to use padding in placement

# EVENTS parameters

event_generator_trigger_probability  # Likelihood of events being triggered

event_min_probability               # Min probability for events

event_max_probability               # Max probability for events

# INFRASTRUCTURE parameters

infr_type                          # "physical" or "logical"

infr_is_dynamic                    # Whether infrastructure can change

infr_node_crash_probability        # Likelihood of node failures

infr_node_resurrection_probability  # Likelihood of node recoveries
```

6. Reporting and Visualization

The simulator generates detailed reports:

```


print_stats()                # Output simulation statistics

export_csv_data()            # Export data to CSV format

visualize_results()          # Generate visualizations
```

Output includes:

- Placement success/failure rates
- Node and link utilization
- Application performance metrics
- Failure event logs

Workflow

Configuration: Define infrastructure and applications

```
python src/lambdafogsim.py -c custom_config.yaml
```

Infrastructure Setup: Either generate physical infrastructure or use logical

```
# For physical infrastructure

python src/lambdafogsim.py -p infrastructure_config.yaml

# For logical infrastructure

python src/lambdafogsim.py -l infrastructure_definition.pl
```

Simulation: The simulator:

- Places applications using the Prolog engine
- Executes function invocations
- Simulates network transfers
- Handles dynamic events (failures, replacements)

Analysis: After simulation:

- Review reports in reports directory
- Analyze CSV exports for detailed data
- Visualize results with the visualization module

Important Files

- lambdafogsim.py: Main entry point and simulation controller
- placement.py: Interface to the Prolog placement engine
- config.py: Configuration parameters parser
- infrastructure_config.yaml: Physical infrastructure definition
- placer.pl: Prolog placement engine
- vis.py: Visualization utilities
- export_data.py: Data export utilities

Running Experiments

Basic execution:

```
python src/lambdafogsim.py
```

With verbose output:

```
python src/lambdafogsim.py -v
```

With custom configuration:

```
python src/lambdafogsim.py -c my_config.yaml -p my_infrastructure.yaml
```

Extending the Simulator

You can customize:

1. Infrastructure definitions
2. Application topologies
3. Placement policies (by modifying the Prolog rules)
4. Failure models
5. Visualization and reporting

Use Cases

1. Testing application resilience to failures
2. Comparing placement strategies
3. Analyzing resource utilization patterns
4. Evaluating QoS guarantees in fog environments
5. Studying security constraints in serverless systems