

# SoCC'22 serverless session notes

Wei Yue

Futurewei Cloud Lab

11/2022

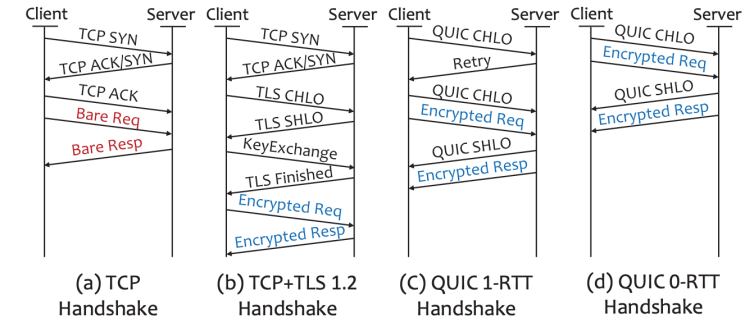
## Serverless session in SoCC'22

1. [QFaaS](#): Accelerating and Securing Serverless Cloud Networks with QUIC( *network acceleration*)
2. [Cypress](#) : Input size–Sensitive Container Provisioning and Request Scheduling for Serverless Platforms(*resource management*)
3. [Method Overloading the Circuit](#) (*service reliability study*)
4. [Hermod](#): Principled and Practical Scheduling for Serverless Functions(*resource management*)
5. [SIMPPO](#): A Scalable and Incremental Online Learning Framework for Serverless Resource Management(*ML for resource management*)
6. [SimLess](#): Simulate Serverless Workflows and Their Twins and Siblings in Federated FaaS(*Simulation*)

Note: there is also a session specifically for *resource management* in SoCC'22

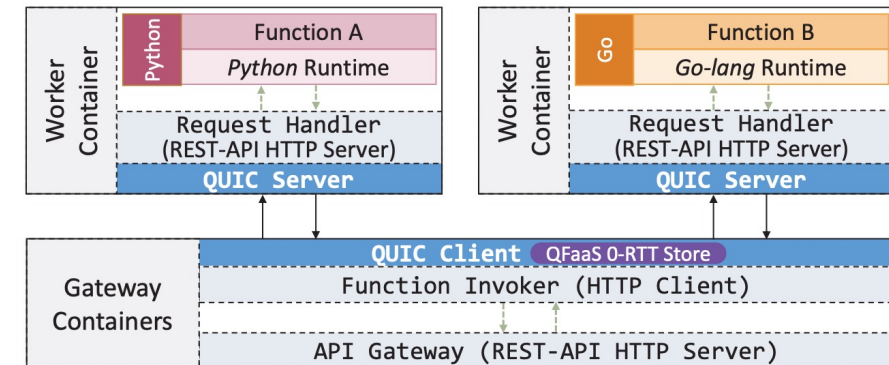
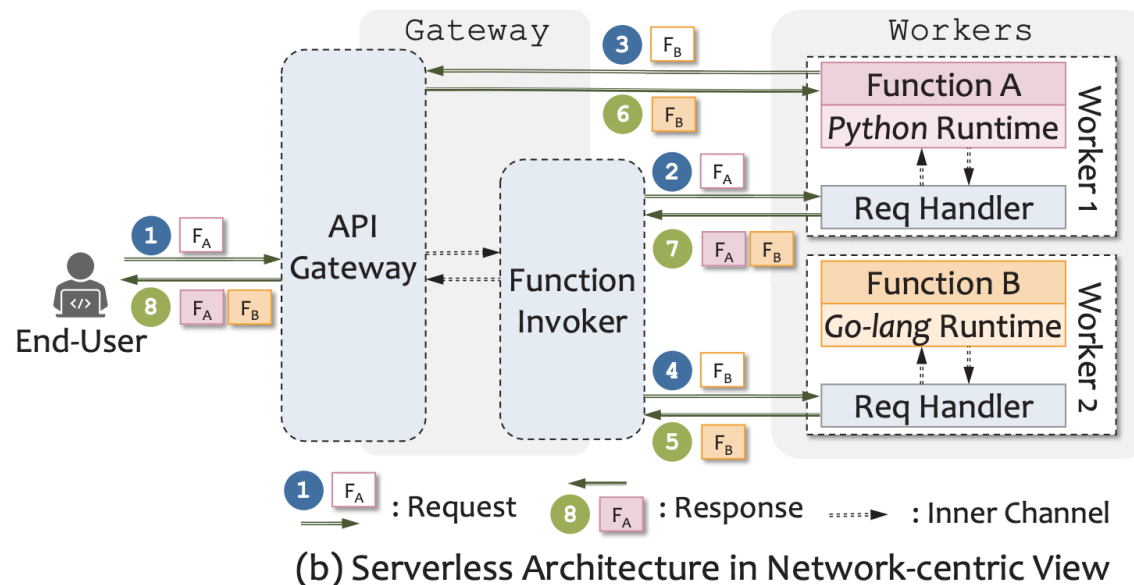
## QFaaS: Accelerating and Securing Serverless Cloud Networks with QUIC

- QUIC-based FaaS framework on **OpenFaaS** platform
- Problem they claim**
  - huge internal network communications between latency sensitive serverless features such as agile autoscaling and function chains;
  - Currently serverless providers sacrifice security for performance by keeping those communications unencrypted
- Proposed solution**
  - Using QUIC instead of TCP for internal communication
  - Define Serverless Architecture in Network-centric View
  - Optimize Internal communication on top of QUIC(QUIC Server, QUIC Client)
- Their tests show QFaaS can reduce communication latency for single functions and function chains by **28%** and **40%**, respectively, and save up to **50 ms** in end-user response time.



Scheme	TCP	TCP + TLS 1.2	QUIC 1-RTT	QUIC 0-RTT
New Session	1	3	1	-
Recover Session	1	2	1	0

**Figure 1: Round-trips in different transport protocols:** (a) insecure TCP incurs 1 extra RTT; (b) in TCP+TLS 1.2 scheme, the encrypted request is sent after 3 RTTs; (c) in QUIC 1-RTT mode (new session establishment), the encrypted request is sent after 1 RTT; (d) in QUIC 0-RTT (session resumption), the encrypted request is sent immediately.



**Figure 3: System Design of QFaaS.** QUIC client and QUIC servers are integrated into Function Invoker and worker request handlers to replace the TCP/TLS client and servers. This modification is transparent to cloud tenants and ensures the activation of the QUIC 0-RTT feature.

## Cypress : Input size–Sensitive Container Provisioning and Request Scheduling for Serverless Platforms

- **Problem**  
Existing serverless **resource management** frameworks are agnostic to the input size–sensitive nature of these apps
- Cypress propose an *input size-sensitive resource management framework* (OpenFaaS + K8s)
  - Minimizes containers provisioned for apps
  - Ensuring high degree of SLO compliance
- Their experimental results show up to **66% less** container spawns, improving container utilization and saving cluster-wide energy by **up to 2.95X and 23%**

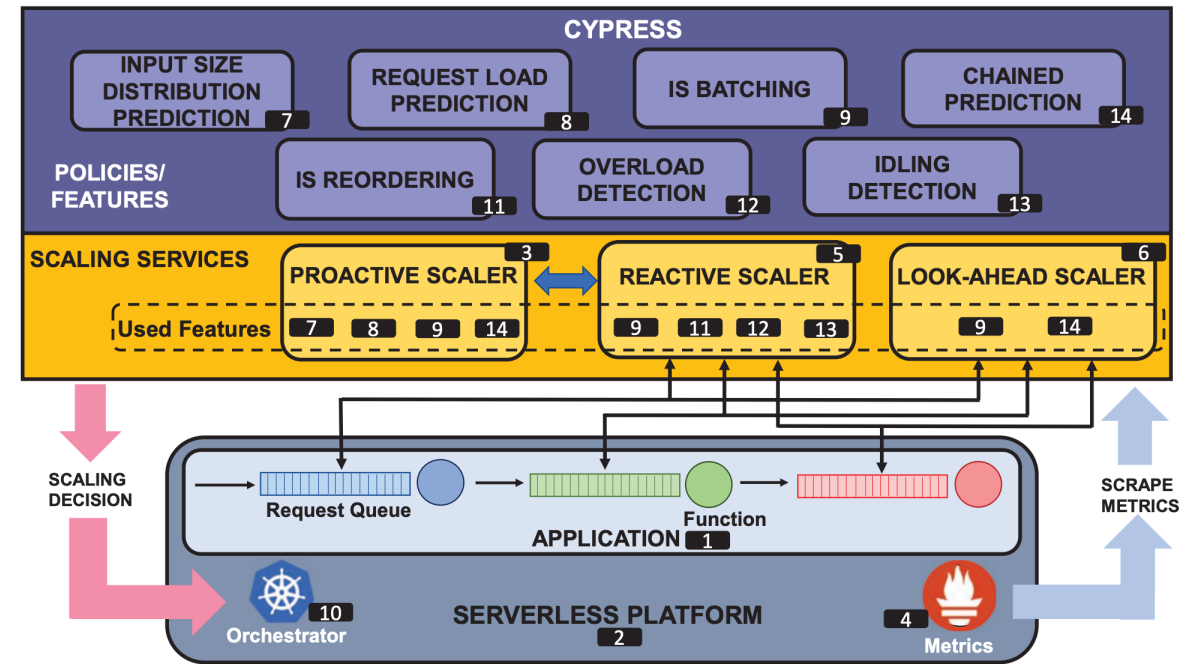
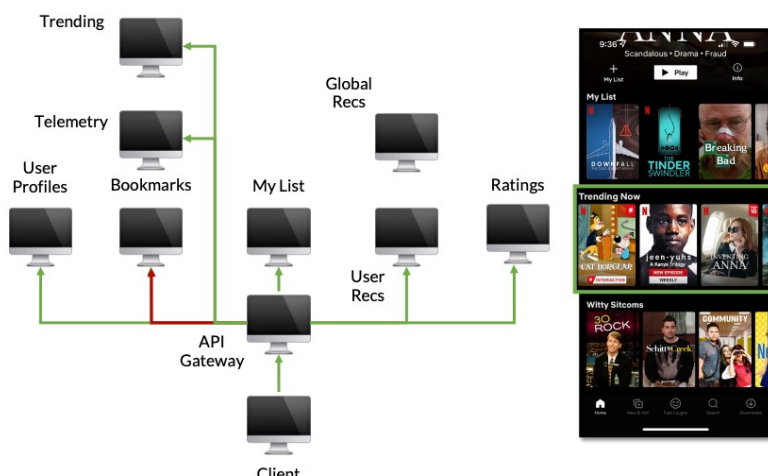


Figure 7: High-level view of *Cypress's* design.

# Method Overloading the Circuit (Service Reliability study)

Case study of DoorDash's **Circuit breaker** behavior in its microservice architecture, which composes of 500+ services.

## What should, and what does, happen?



**Fallbacks**  
Developers specify **alternative application logic** in the event of dependency failure.

**Other resilience techniques:**

- 1 Retries
- 2 Timeout
- 3 Load shedding
- 4 Circuit breakers

Fault injection and chaos engineering used to **verify what should happen does happen**.  
[Meiklejohn et al. 2021, SoCC '21]

## Reliability at DoorDash

### 1. Fallbacks

When dependencies are unavailable, load alternative content from different services or use default responses to allow application to degrade gracefully.  
(e.g., personalized recommendations become generic recommendations.)

### 2. Cluster Orchestration

Support for rolling deploys with replicas of services supported by load balancing. Combined with single retries (not timeout), lets nodes to hit non-failed replica on retry. Automatic readiness and liveness checks with auto-scaling and restart policies.

### 3. Load Shedding

Short-circuit request at the **callee** using a predefined error indicating overload. Typically performed based on the number of outstanding concurrent requests.

### 4. Circuit Breakers

Short-circuit request at the **caller** using a predefined error indicating failure condition. Typically performed based on the number of observed errors within a specific period.

## Circuit Breakers: Overview

### Circuit Breakers

**Interpose on RPCs** between services and record successes/errors to determine if RPC should be allowed. With on a min threshold of requests and a sliding window, determine if the num of errors have exceeded a threshold.

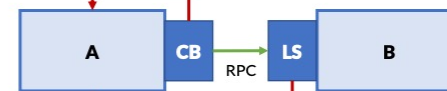
### Load Shedding

Special case of circuit breakers that use number of outstanding requests at a given service.

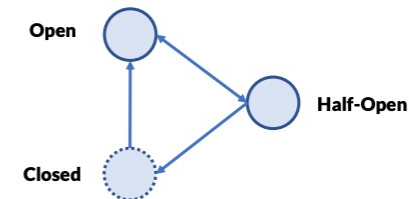
### Transitions

1. Circuits begin in the **closed** state. When the threshold is exceeded move to the **open** state where all RPCs are refused.
2. Circuits move to the **half-open** state to determine if they should move to **open** if a subset of RPCs succeed.

Too many errors, short-circuit RPC



Too many outstanding requests, short-circuit RPC



- Their claim
  - Existing circuit breaker designs are insufficient for fault tolerance;
  - Even a small common abstraction changes in application code can drastically alter how circuit breakers work.
- Propose two new designs for circuit breaker: **path-sensitivity** and **context-sensitivity**.
- They **prefer path-sensitivity** approach for minimal developer overhead with proper sensitivity.
- Introduce **DEI** (distributed execution Indexing), **an algorithm used in fault injection testing in DoorDash**, to be used in circuit break implementation to provide path-sensitivity.

# Hermod: Principled and Practical Scheduling for Serverless Functions

A **Scheduler for Serverless Functions** with two key characteristics

- a combination of early binding and processor sharing for scheduling at individual worker machines to avoid head-of-line blocking due to high function execution time variability.
- Cost, load, and locality-aware. It improves consolidation at low load, it employs least-loaded balancing at high load to retain high performance, and it reduces the number of **cold starts** compared to pure load-based policies.
- Build on open-source serverless platforms, Apache **OpenWhisk**( it powers IBM’s commercial serverless offering).

Claims to achieve up to **85% lower function slowdown** and **60% higher throughput** compared to existing production and state-of-the-art research schedulers for the case of the function patterns observed in real-world traces.

## Existing Approaches

### Serverless Scheduling Taxonomy

#### T / LB / S

T: Type of binding used (early E vs. late L)

LB:        LOC – locality-based  
             LL – least-loaded  
             R – random

S: intra-Worker policy  
     FCFS – First-Come-First-Serve  
     PS – Processor Sharing

System	Policy	Load-aware	Cost-aware	Locality-aware
OpenWhisk	E/LOC/PS	✗	✗	✓
kNative	E/R/PS	✗	✗	✓
Sparrow	Late Binding	✓	✗	✗
Hermod	E/Hybrid/PS	✓	✓	✓



# SIMPPO: A Scalable and Incremental Online Learning Framework for Serverless Resource Management

## What do they do?

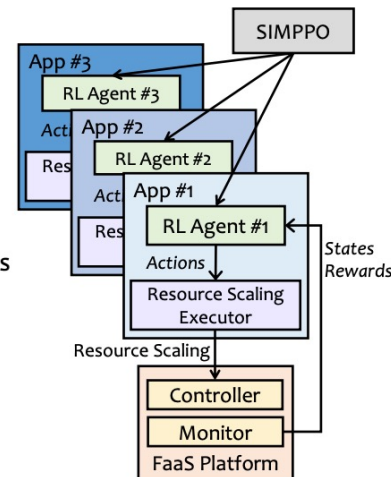
- Apply ML to cloud computing/cloud network
- Automate the management for diverse workloads with reinforcement learning (RL)
- Trying to achieve *per-function performance-wise SLOs*, which are **not supported** yet in Serverless (FaaS).

## Benchmark Serverless Function

- SIMPPO provides online policy-serving performance **comparable to single-agent RL** in isolation (the baseline), with the performance degradation <9.2%
- In **multi-tenant/agent** environments:
  - SIMPPO achieves **2x-4.4x** improvement compared to single-agent RL
  - SIMPPO has **21.4x** less performance degradation compared to a threshold-based approach ENSURE (ACSOS 2020)

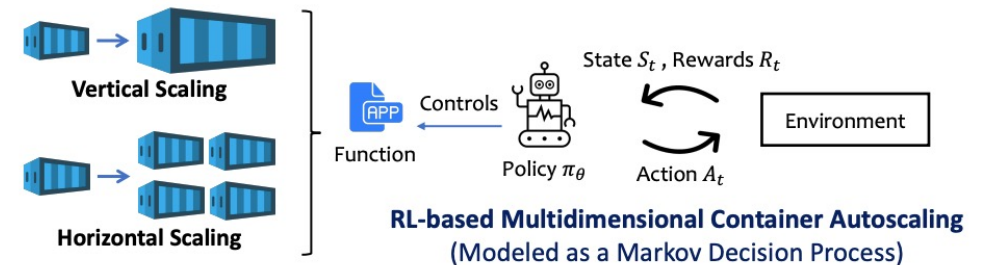
## SIMPPO: Scalable and Incremental MARL

- Two building blocks of SIMPPO
  - **Virtual agent**
  - **Auxiliary global system states**
- Applied SIMPPO to **multi-dimensional autoscaling** of serverless platforms
  - Based on the state-of-the-art RL algorithm – **PPO** (Proximal Policy Optimization)
  - Serverless platform: **OpenWhisk**
- Evaluated SIMPPO on 12 open-source serverless benchmarks
  - Function invocation patterns from Azure Functions traces
- **RQ1: Incremental training?**
- **RQ2: Online policy-serving performance?**

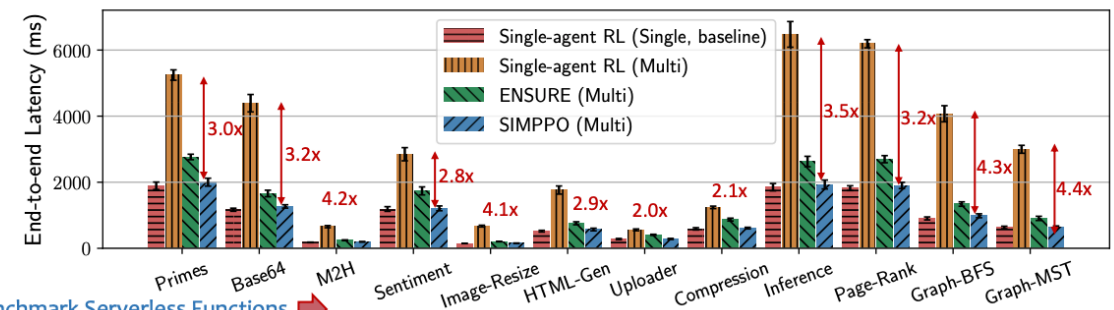


## ML-managed SLO-driven Cloud Services

- **Why ML?:** **Heuristics-based** resource management are inefficient and not tenable
  - Providers dynamically manage orchestration platforms to achieve efficiency as cloud evolves
- **Contributions:**
  - **SIMPPO:** Automate the management for diverse workloads with **reinforcement learning (RL)**
  - **Quantitative characterization study** of existing RL approaches
  - A system that orchestrates **multiple learning-based agents** to achieve optimal resource allocation in the task of **multidimensional container autoscaling**
  - **Key Idea:** “Virtual Agent” and mean-field theory



## SIMPPO Online Performance



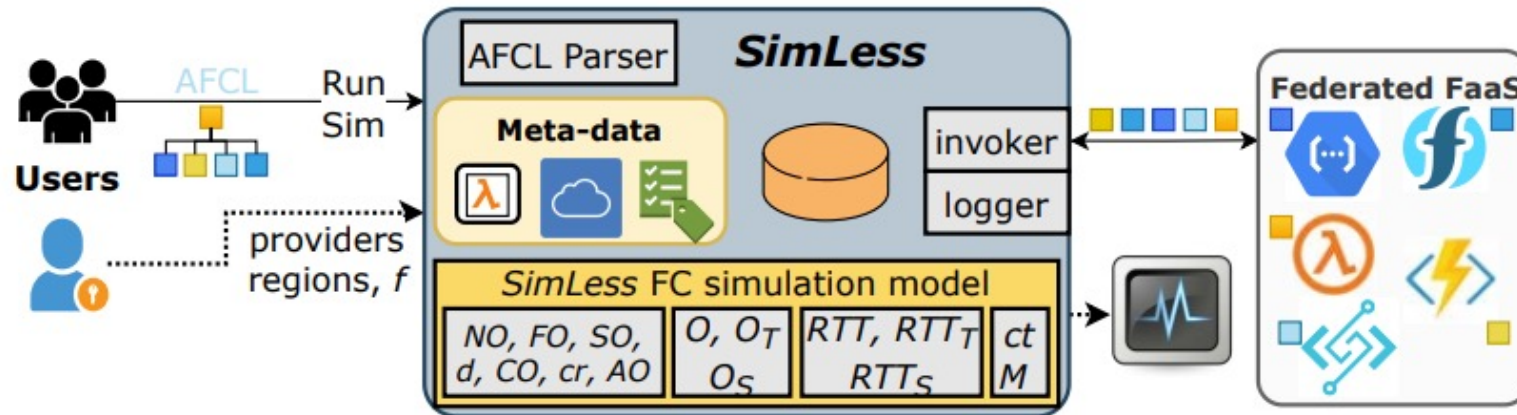
## Benchmark Serverless Functions

- SIMPPO provides online policy-serving performance **comparable to single-agent RL** in isolation (the baseline), with the performance degradation <9.2%
- In **multi-tenant/agent** environments:
  - SIMPPO achieves **2x-4.4x** improvement compared to single-agent RL
  - SIMPPO has **21.4x less** performance degradation compared to a threshold-based approach ENSURE (ACSOS 2020)

## SimLess: Simulate Serverless Workflows and Their Twins and Siblings in Federated FaaS

An FC(function choreographies) simulation framework for accurate FC simulations across multiple FaaS providers with a simple and lightweight parameter setup with two light concepts:

- **Twins**, representing the same function deployed with the same computing, communication, and storage resources, but in other regions of the same FaaS provider,
- **Siblings**, representing the same function deployed in the same region with different computing resources.



**Figure 4: *SimLess* system architecture.**



# *Current Serverless Limitation and how XDP/eBPF may help?*

- **Existing framework limitations**

- Still in the early stage, mostly suitable to simple workload of independent tasks and/or well tailored proprietary service offerings from the vendors
- Limitation in AWS Lambda:
  - Limited Lifetimes – 15 mins
  - I/O Bottlenecks -- 538Mbps network bandwidth, an order of magnitude slower than a single modern SSD
  - Communication Through Slow Storage
  - No Specialized Hardware support
- Academia think existing serverless offering is “***one step forward, two steps back***”, more to be studied

- **Use Case deep dive**

- QFaaS as an example, tries to address both security and latency issues in serverless framework

- **Which serverless frameworks to start?**

- Opensource frameworks:
  - OpenFaaS, OpenWhisk, Knative
- Cloud provider solutions:
  - AWS Fargate, AWS Lambda, Azure Functions, Google Cloud Functions, etc.

- **Potential approach**

- ***Target desirable application in a particular serverless framework, analyze it and try further optimize overall performance with XDP/eBPF and/or other technologies***