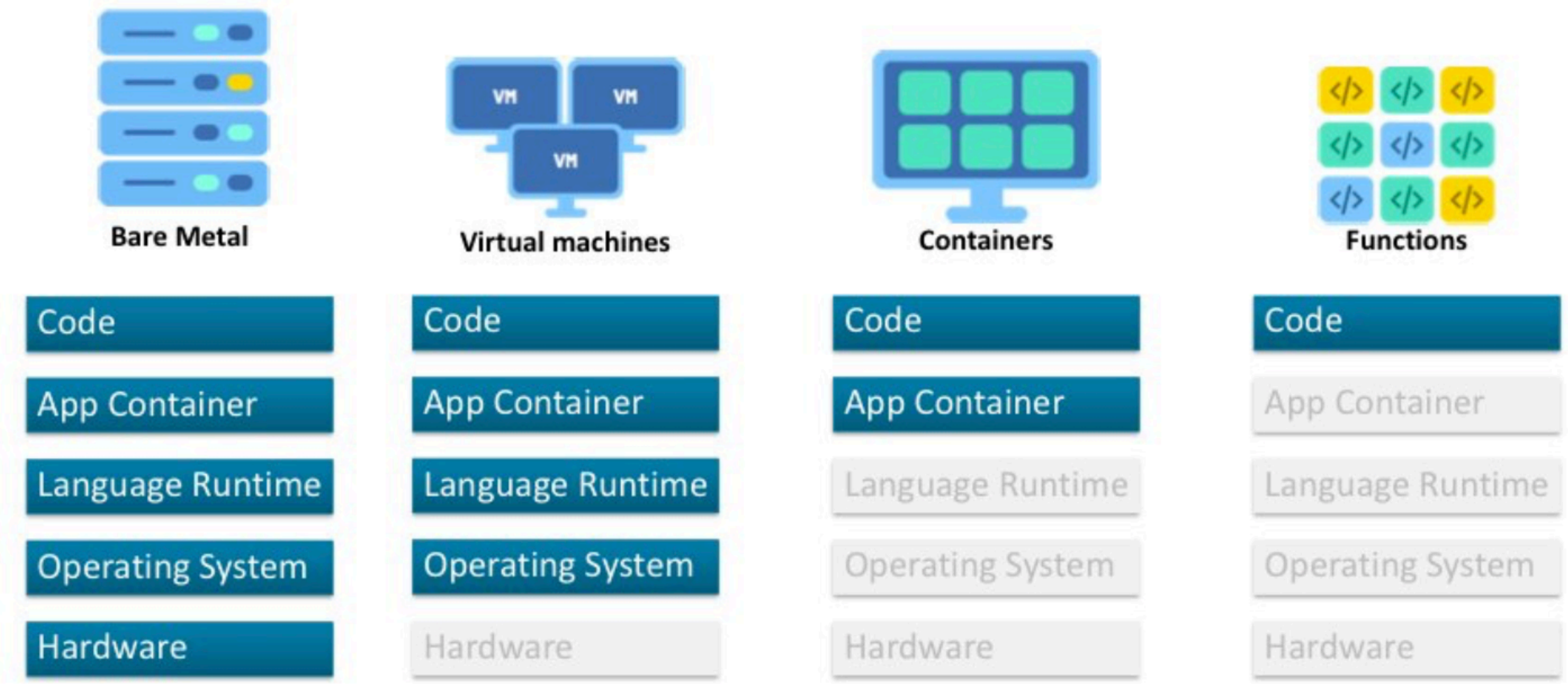
Sequoia: Enabling Quality-of-Service in Serverless Computing

SoCC 2020

(CS 591 Presentation)

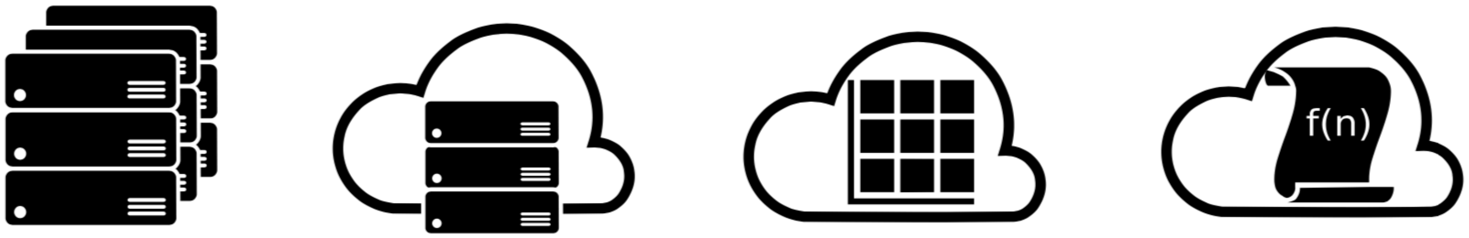
Serverless – Motivation, Pitfalls and Challenges

Four ways of cloud resource provisionings



https://blogs.oracle.com/developers/functions-as-a-service:-evolution,-use-cases,-and-getting-started

What is Serverless?

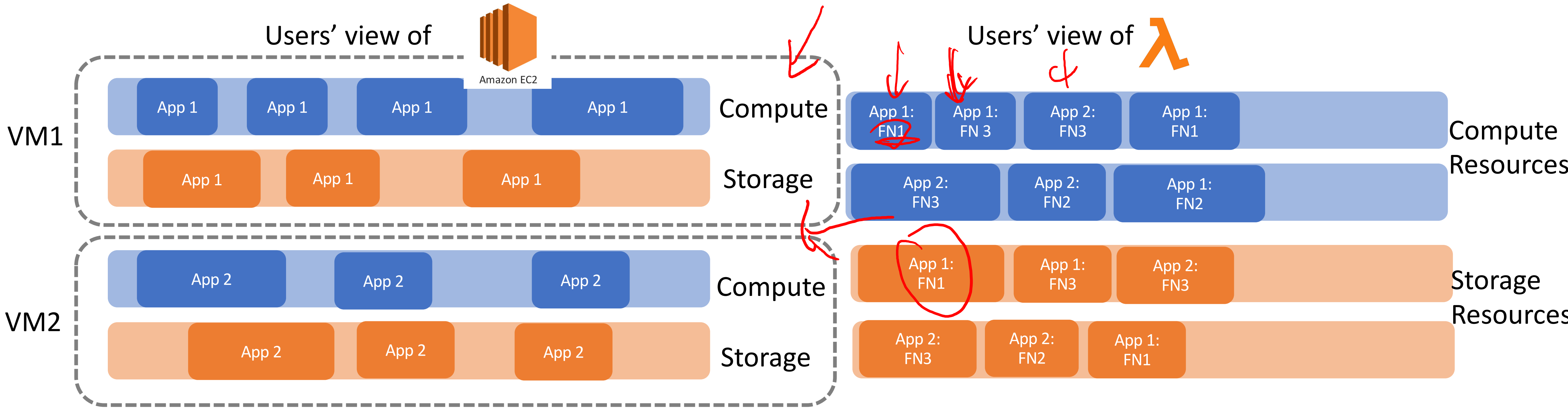


	Bare Metal	VMs (IaaS)	Containers	Functions (FaaS)
Unit of Scale	Server	VM	Application/Pod	Function
Provisioning	Ops	DevOps	DevOps	Cloud Provider
Init Time	Days	~1 min	Few seconds	Few seconds
Scaling	Buy new hardware	Allocate new VMs	1 to many, auto	0 to many, auto
Typical Lifetime	Years	Hours	Minutes	O(100ms)
Payment	Per allocation	Per allocation	Per allocation	Per use
State	Anywhere	Anywhere	Anywhere	Elsewhere

Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

Serverful vs. Serverless

- Serverful
  - User manages resources by VMs
    - Responsibilities include…
  - Charge by the time of VMs being spawned
- Serverless
  - User spawn applications in the form of chained functions
  - Charge by the resource time used by each function invocation



Cloud Programming Simplified: A Berkeley View on Serverless Computing

- Three major differences vs. serverful computing
  - Decoupled computation and storage. The storage and computation scale separately and are provisioned and priced independently.
  - Executing code without managing resource allocation.
  - Paying in proportion to resources used instead of for resources allocated.
- Limitations of today's serverless platforms
  - High cost accessing cloud storage at fine granularity
    - States maintained remotely at cloud storage and accessed frequently
  - Lack of fine-grained coordination
    - Execution dependency between tasks requires handling in a scalable fashion
  - Inefficient use of network resources
    - VMs provide opportunities to optimize network usage
  - Unpredictable performance
    - Deployment environment is beyond users' control

Cloud Programming Simplified: A Berkeley View on Serverless Computing

- Three major differences vs. serverful computing
  - Decoupled computation and storage. The storage and computation scale separately and are provisioned and priced independently.
  - Executing code without managing resource allocation.
  - Paying in proportion to resources used instead of for resources allocated.
- Limitations of today's serverless platforms
  - High cost accessing cloud storage at fine granularity
    - States maintained remotely at cloud storage and accessed frequently
  - Lack of fine-grained coordination
    - Execution dependency between tasks requires handling in a scalable fashion
  - Inefficient use of network resources
    - VMs provide opportunities to optimize network usage
  - Unpredictable performance
    - Deployment environment is beyond users' control

Sequoia: Enabling Quality-of-Service in Serverless Computing

- Motivation:
  - Function scheduling is largely done in the FIFO fashion
  - Doesn’t support function performance with QoS
- Need a ecosystem with QoS control
  - Enforce policy on functions across chains, or within chains
  - Drop-in front-end with policy enforcer

Sequoia: Enabling Quality-of-Service in Serverless Computing

Thee types of workload chains

Single: the simplest workload consisting of individual independent requests.

Linear-N: A serverless chain where every serverless function invokes up to one new serverless function.

Fan-N: Another chain where multiple tasks depend on a previous function's completion.

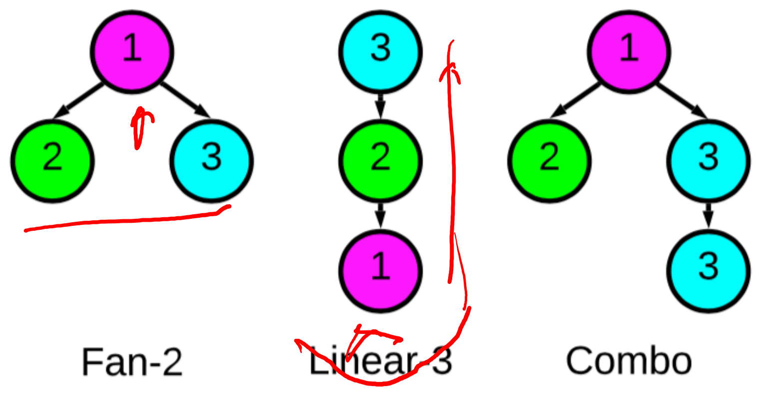
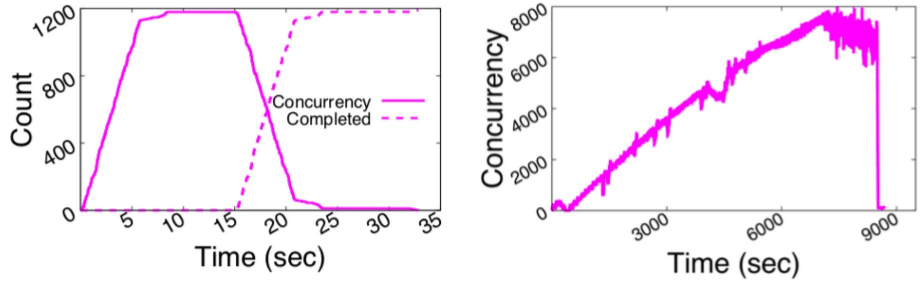


Figure 1: Example function chains in study

Conclusion:
(i) scheduling across frameworks follows a simple FIFO queuing model and
(ii) scheduling is performed on a per-function basis (instead of other policies like per-chain).

Limitations: Inconsistent and incorrect concurrency limits



(a) IBM Cloud Functions (b) Azure Functions

Figure 2: Incorrect concurrency limits

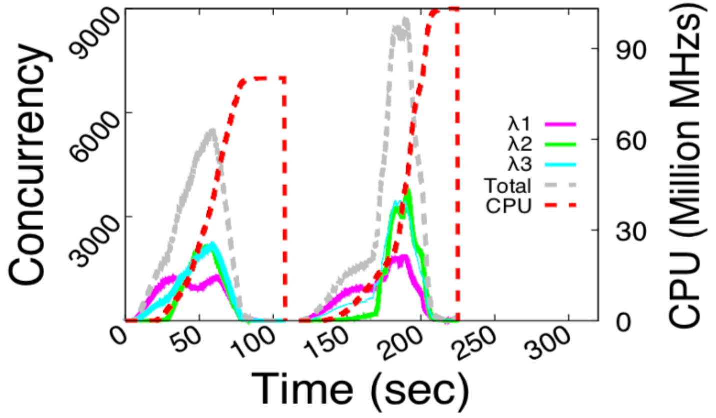


Figure 3: GCF: MixedChain workload CPU usage

IBM: **Configured**: Concurrency up to 1000. **Actual**: 1200

Azure: **Configured**: Max functions up to 1000, max instances: 200. **Actual**: 8000 and 440

GCF: **Configured**: CPU usage is configured to reach 40M MHz/s . **Actual**: 90M MHz/s

When limits are **under** intended values, workloads may unexpectedly encounter **poor performance** or **increased drops**.
When limits are **over** intended values, developers may incur **higher** costs than budgeted for.

Limitations: Mid-chain Drops

Mid-chain drops: Functions issued beyond hard-concurrency limit will be dropped or queued

- Developers may rely on function chain completion, and when function chains drop mid-chain, **incorrectness** may arise.
- Solving problem from application levels **increase developer's effort**, defeating purpose of Serverless architecture
- **Resource waste** for incomplete function chains

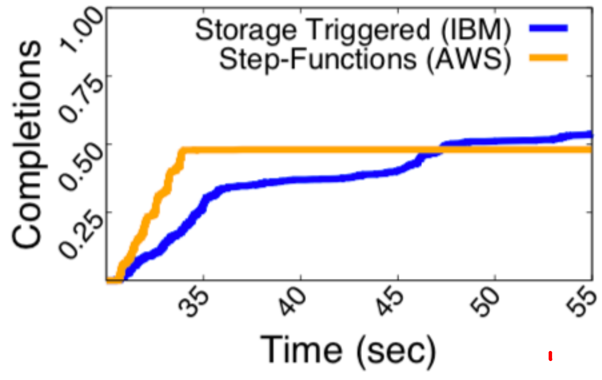
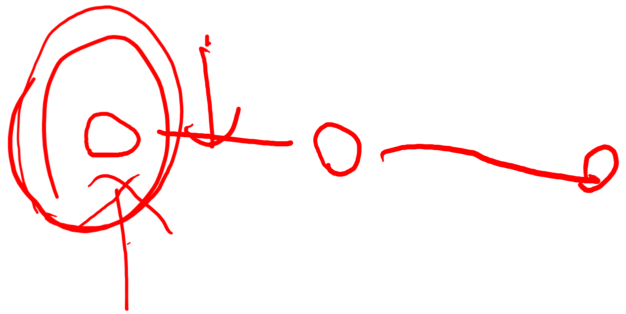


Figure 4: Mid-chain drops

Fan-2 Burst workload: Burst set to concurrency limit (1000), result in total concurrency of 2000: 48 – 54% complete successfully

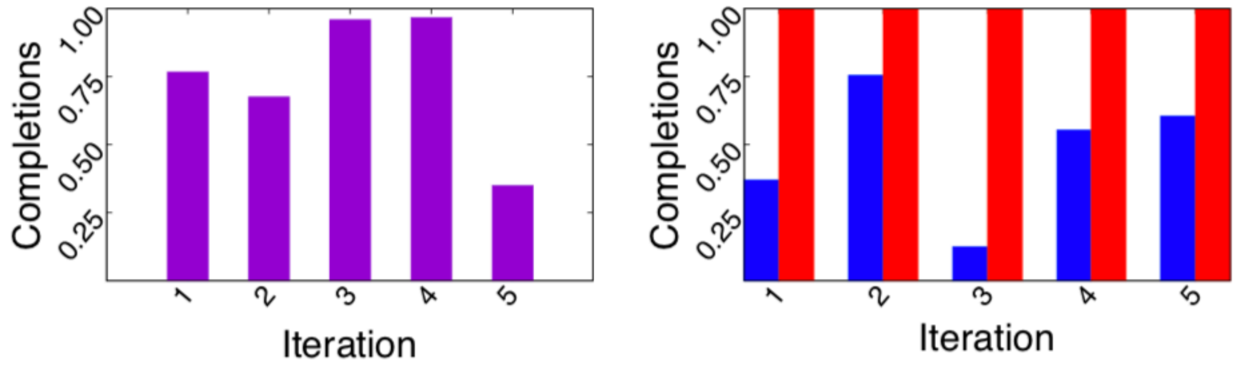


Limitations: Burst Intolerance

Inconsistent achievable concurrency:

- Concurrency ranges from approximately 1,000-2,000 when a large burst of 6,000 HTTP requests is invoked.
- Bursts repeated over 5 iterations have inconsistent achievable concurrency (5a)
- Significant loss of consistency in concurrency for functions with cold starts (introduced by bursts)

Burst intolerance limits achievable concurrency, which in turn creates loss or queuing when demands spike.



(a) GCF (b) Azure Functions

Figure 5: Workload burst intolerance; in Figure 5b blue bars are cold starts and red bars are warm starts

Limitations: HTTP Prioritization

HTTP requests are being prioritized:

- Fan-2 workload ($\lambda 1$: HTTP, $\lambda 2$, $\lambda 3$: background work) that saturate concurrency limit
- Single workload ($\lambda 4$: HTTP) from time 500-1000
- $\lambda 1$ and $\lambda 4$ have concurrencies **increased** over time (prioritized over $\lambda 2$, $\lambda 3$)
- HTTP functions prioritized over regular workload, for unknown reasons

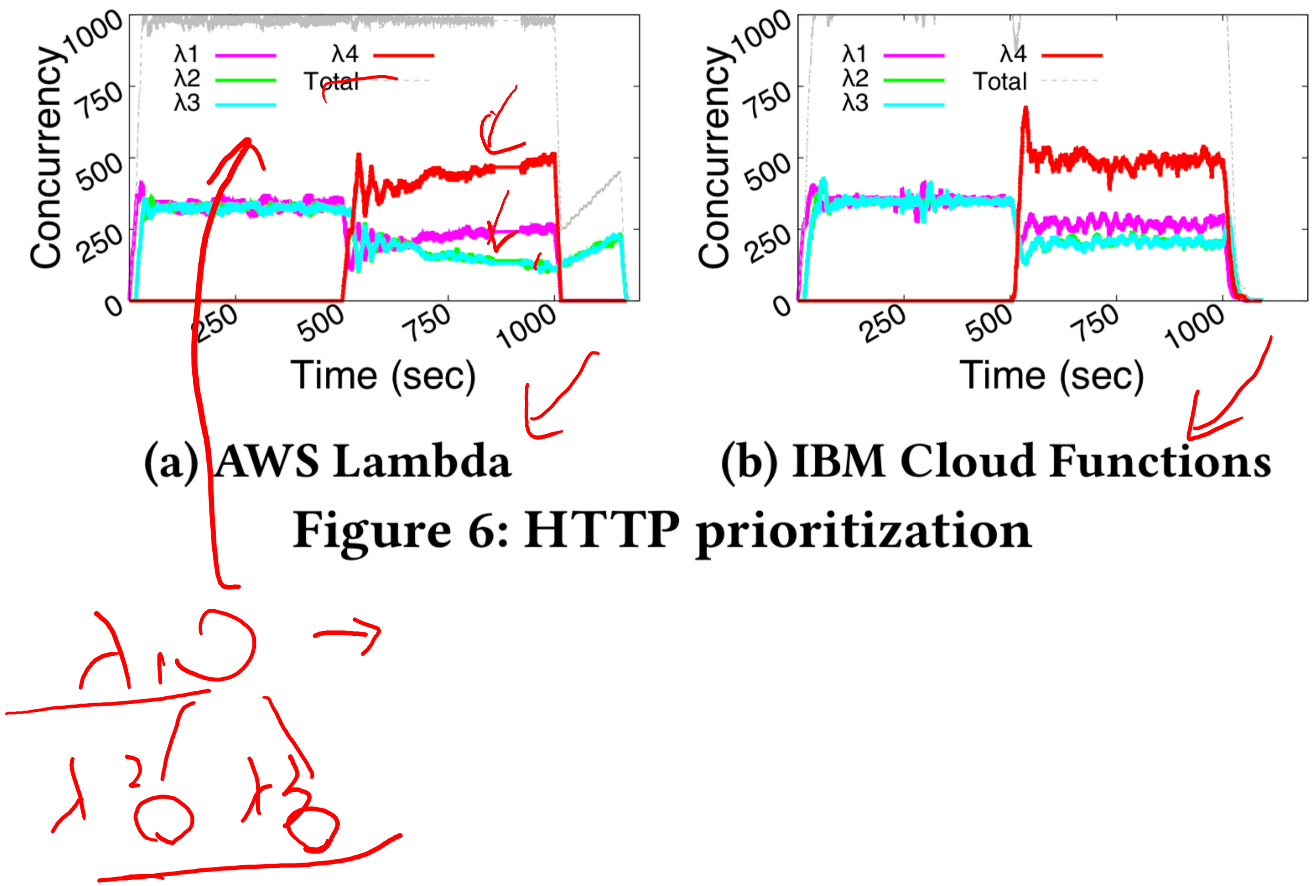
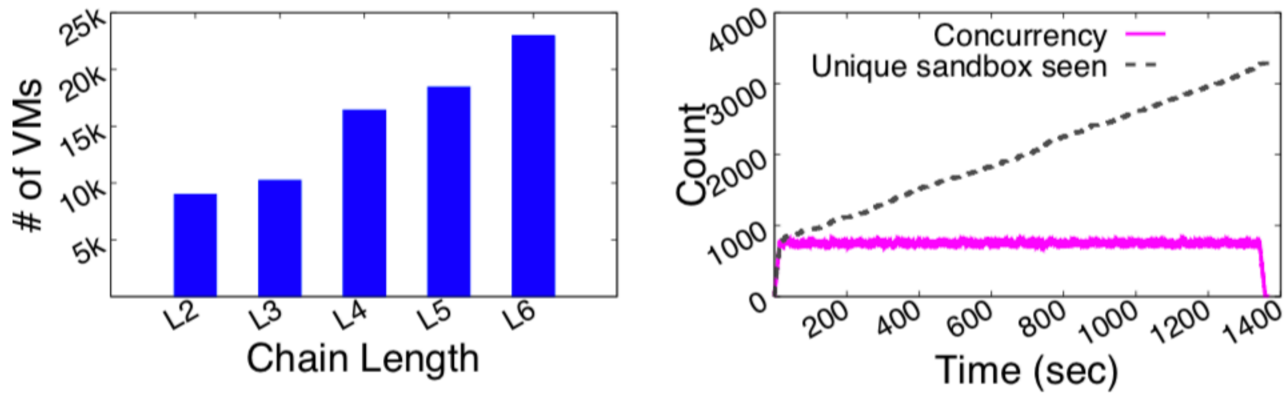


Figure 6: HTTP prioritization

Limitations: Insufficient Resource Allocation

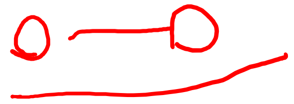
Insufficient resource allocation: Using a much higher VM/container pool than the concurrency limit ensures faster scale-up and less cold-starts. This leads to inefficient resource allocation

- AWS: **Bursting Workload with Linear-N topo with variable length:** VMs are not re-used and # of unique VMs increase with chain length
  - Linear L2, **ideal:** 2K **Actual:** 10K
- IBM: single function sees unique sandboxes increase even when reused is possible



(a) AWS Lambda (b) IBM Cloud Functions

Figure 7: Inefficient resource allocation

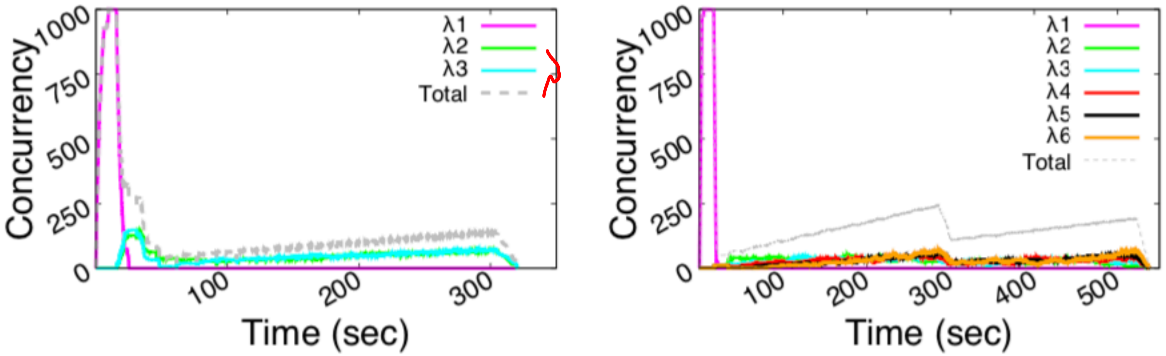


Inefficient VM/container reuse can increase overheads such as cold start and also inefficiently utilize memory.

Limitations: Concurrency Collapse

Concurrency Collapse: Concurrency reaches the limit but then drops and does not immediately recover

- The concurrency collapse significantly after $\lambda 1$ completes
- $\lambda 2$ and $\lambda 3$ does not saturate resources after $\lambda 1$ completes
- Possibly caused by no available container is readily available after $\lambda 1$ completes



(a) Fan-2 (b) Fan-5

Figure 8: AWS Lambda concurrency collapse

Sequoia Architecture

Standalone scheduling framework that can be deployed proxy to existing cloud services

- QoS Scheduler
  - Producer: Enqueuing new functions
  - Producer: Initializing ChainState (read by RM)
  - RM: pulling functions to CRQ (subsequent functions in the chain)
- Logging Framework
  - Provide historical information (function performance, error reporting, details about the underlying containers and VMs hosting the functions)
- Policy Framework
  - An entry point to add, remove, or alter policies in the system
  - Policies: *Function-level Allocation, Chain-level Allocation, Reactive Concurrency Allocation, Ongoing Chain Prioritization, Shortest Job First, Explicit Priority Assignment, Hybrid Scheduler, Resource-aware Scheduler*

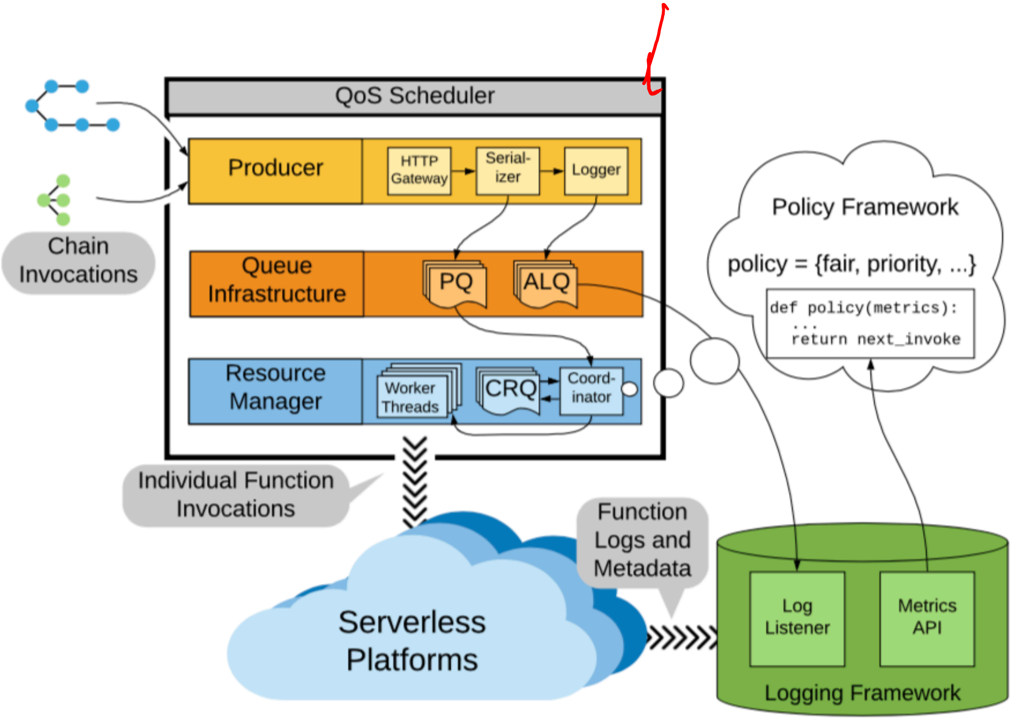
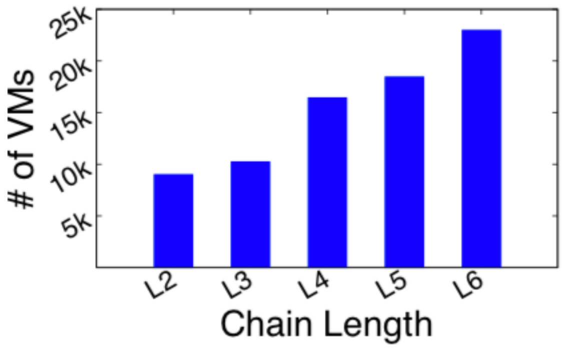


Figure 9: Sequoia architecture

Mitigating limitations:



(a) AWS Lambda

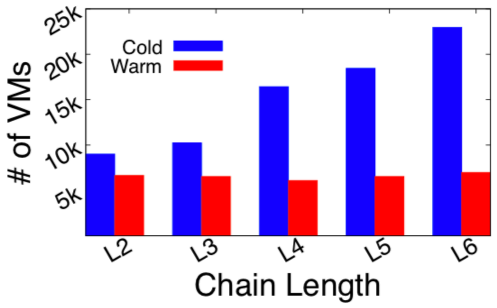
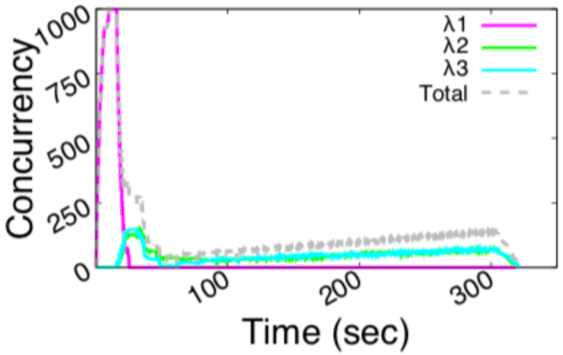


Figure 11: Resource allocation in linear chain analysis



(a) Fan-2

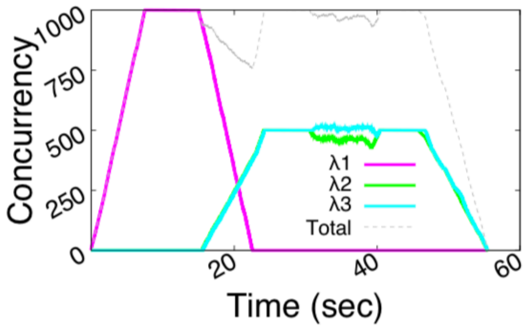


Figure 12: Resource-aware policy prevents concurrency drop

Limiting sending rate improves utilization

Limiting sending rate prevents concurrency collapse (shorten the overall completion time by 5.5X)

QoS based Policy

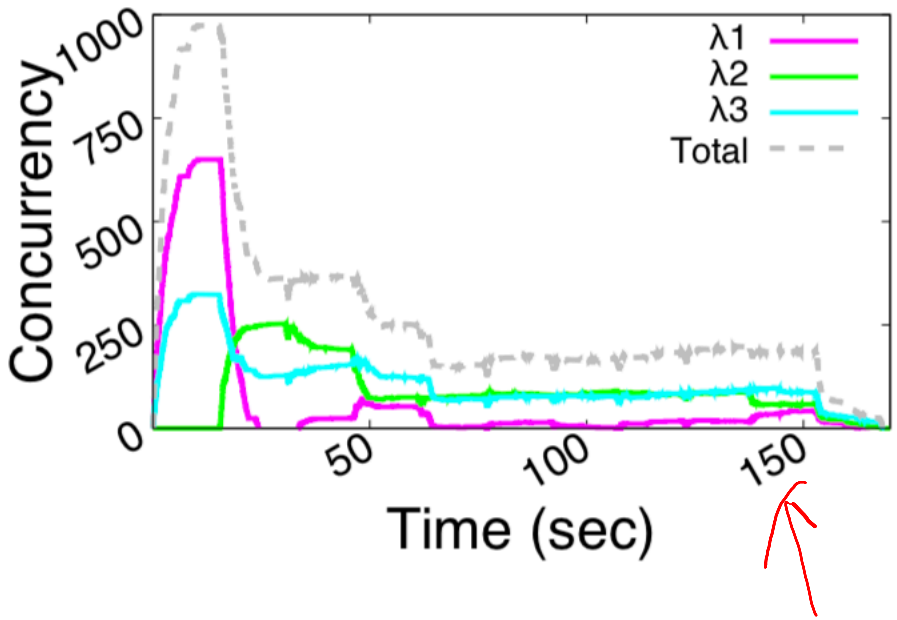
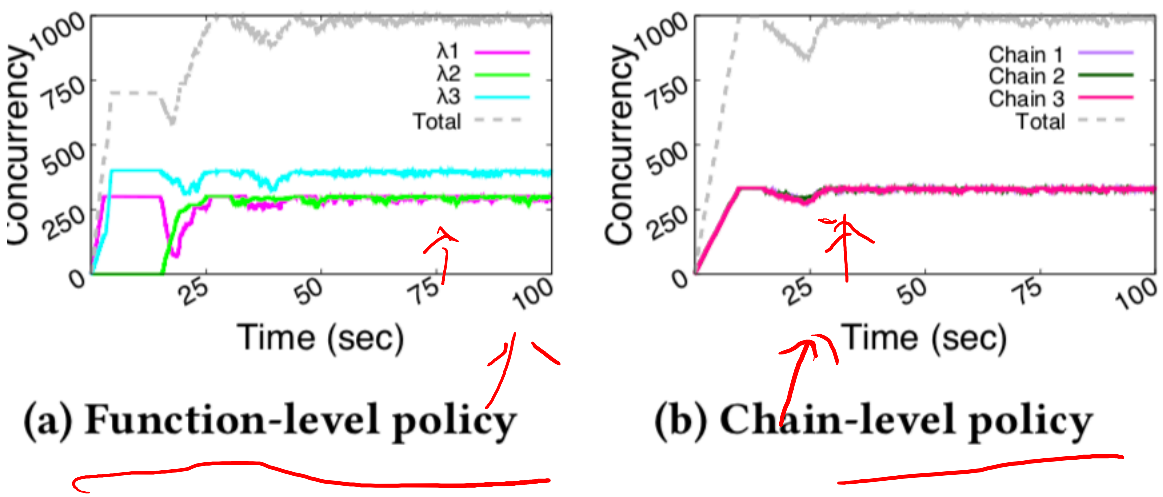


Figure 13: AWS MixedChain baseline



(a) Function-level policy (b) Chain-level policy

Figure 14: QoS policy validation

Function level: Concurrency divided fairly based on number of function invocations

Chain level: Concurrency divided fairly based on number of Chains

QoS based Policy

Reactive Concurrency Scheduling:

- Fan-2 and $\lambda 4$ starts equal
- t = 200: Fan-2 becomes 20 IPS and λ4 becomes 8 IPS (a 2.5:1)
- t = 400: IPS ratios again become equal (17 IPS)
- t = 600: the λ4 IPS becomes 3× the Fan-2 IPS.
- t = 800: ratios are again equal.

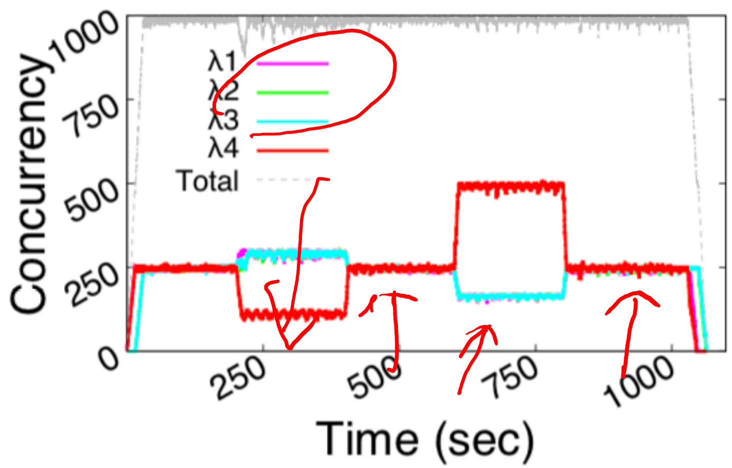


Figure 15: Reactive Concurrency Sharing with adaptive workload

List of Papers

- Serverless Computing: Vision, Pitfalls, Challenges:
  - Cloud Programming Simplified: A Berkeley View on Serverless Computing
  - Serverless Computing: One Step Forward, Two Steps Back
- Serverless computing today, observations:
  - Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider
- Serverless computing from every aspects:
  - Narrowing the Gap Between Serverless and its State with Storage Functions
  - Cirrus: a Serverless Framework for End-to-end ML Workflows
  - Kappa: A Programming Framework for Serverless Computing
  - Serverless Boom or Bust? An Analysis of Economic Incentives