

Resource-Centric Serverless Computing

Yiying Zhang

Outline

- Managing states: Pocket
- Resource efficiency: Zenix
- Quiz 2 next Mon (covers lectures from DNN training until today's)
- Project progress report due tomorrow



Elastic Ephemeral Storage for Serverless Analytics

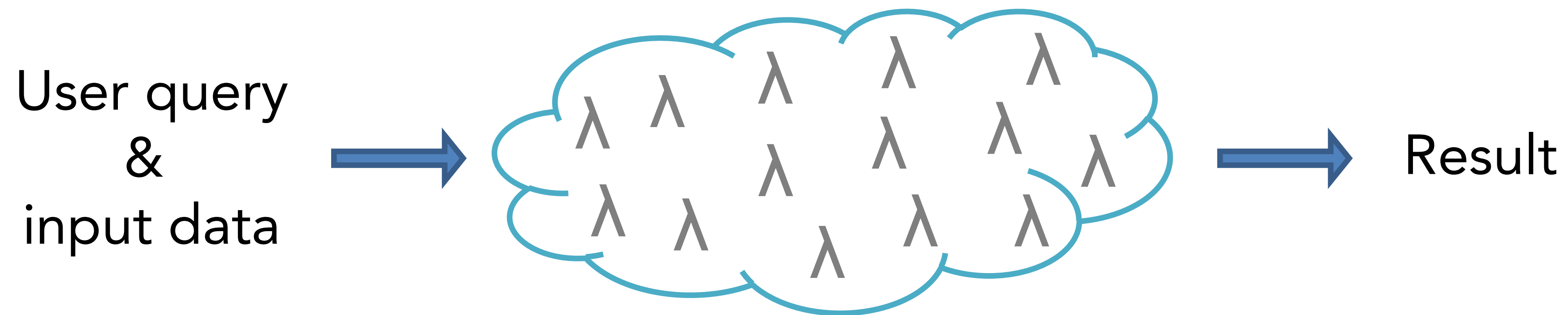
Ana Klimovic*, Yawen Wang*, Patrick Stuedi+,
Animesh Trivedi+, Jonas Pfefferle+, Christos Kozyrakis*

*Stanford University, +IBM Research

OSDI 2018

Serverless Computing

- Serverless computing enables users to launch short-lived tasks with **high elasticity** and **fine-grain resource billing**
- Serverless computing is increasingly used for **interactive analytics**
 - Exploit massive parallelism with large number of serverless tasks

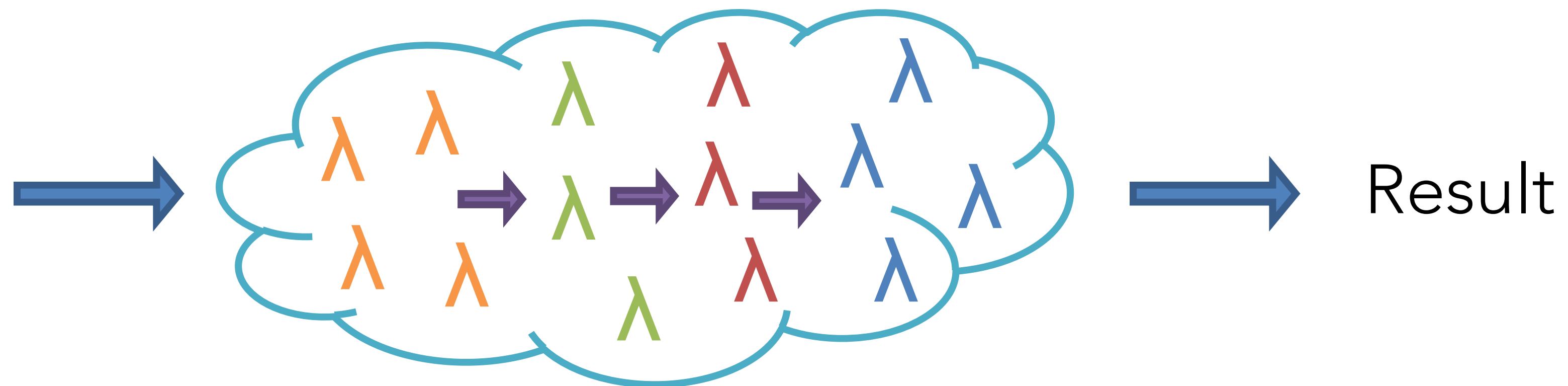


The Challenge: Data Sharing

- Analytics jobs involve multiple stages of execution
- Serverless tasks need an efficient way to communicate **intermediate data** between different stages of execution

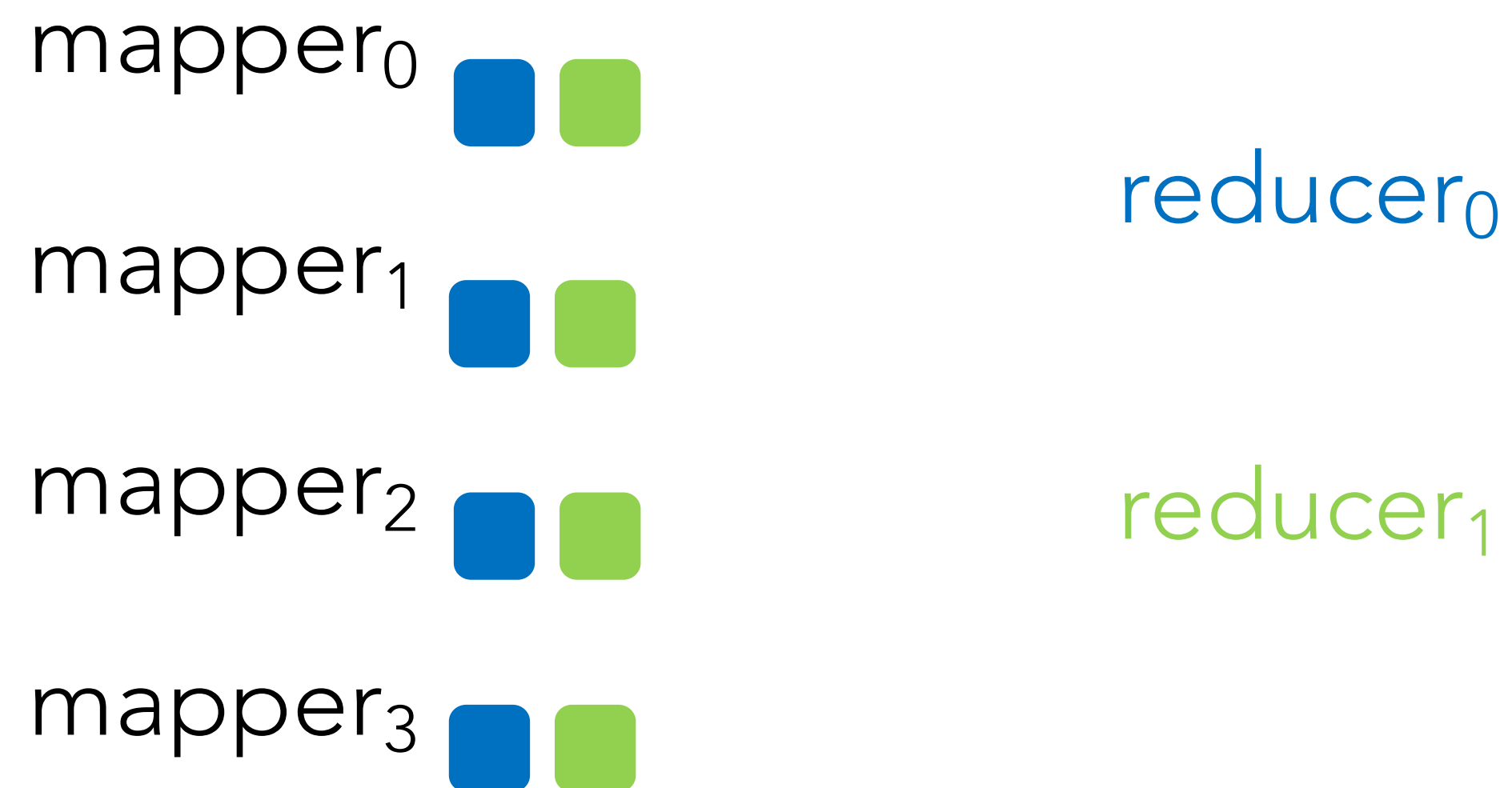
ephemeral data

User query
&
input data



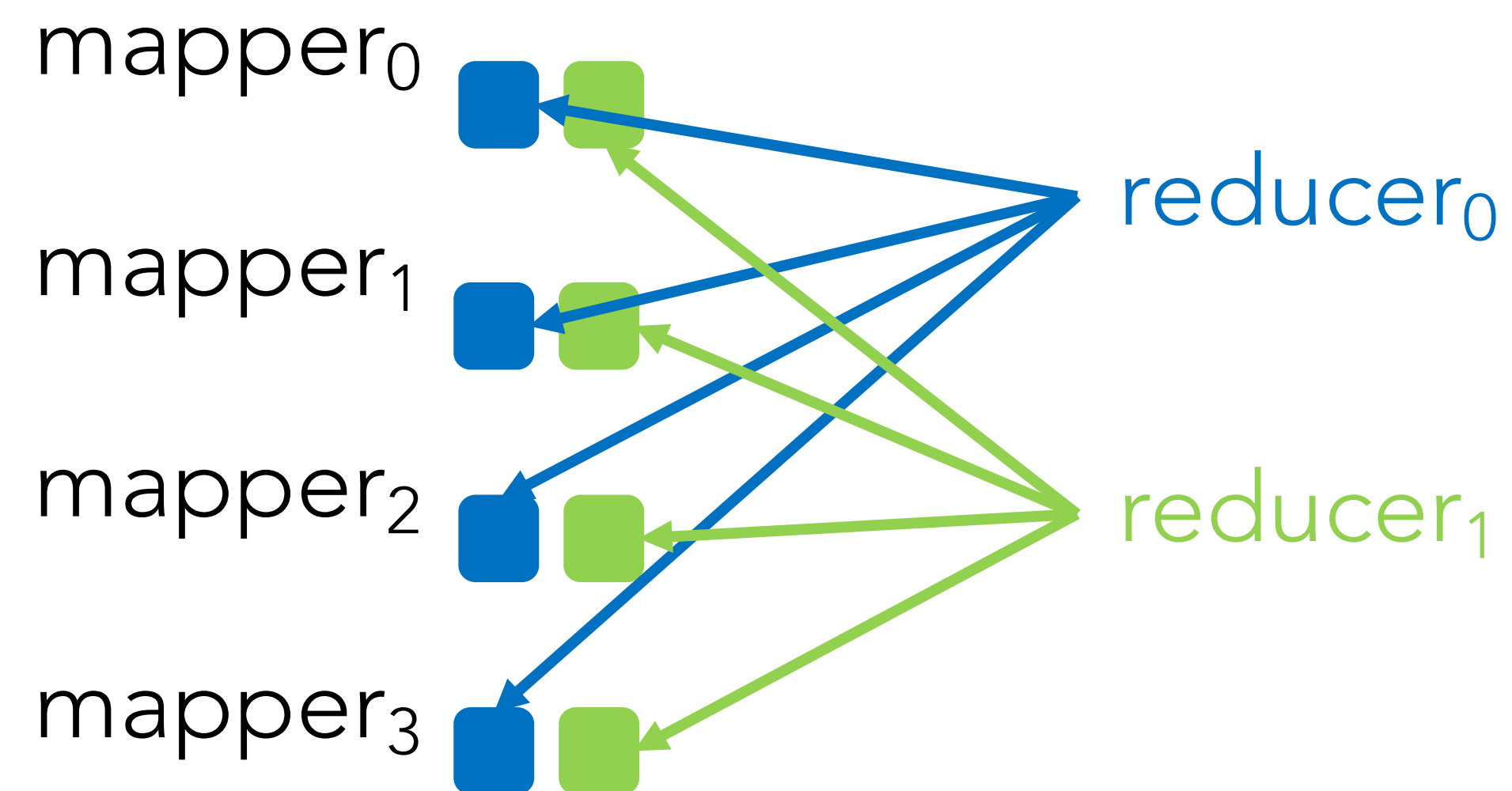
In traditional analytics...

- Ephemeral data is exchanged directly between tasks



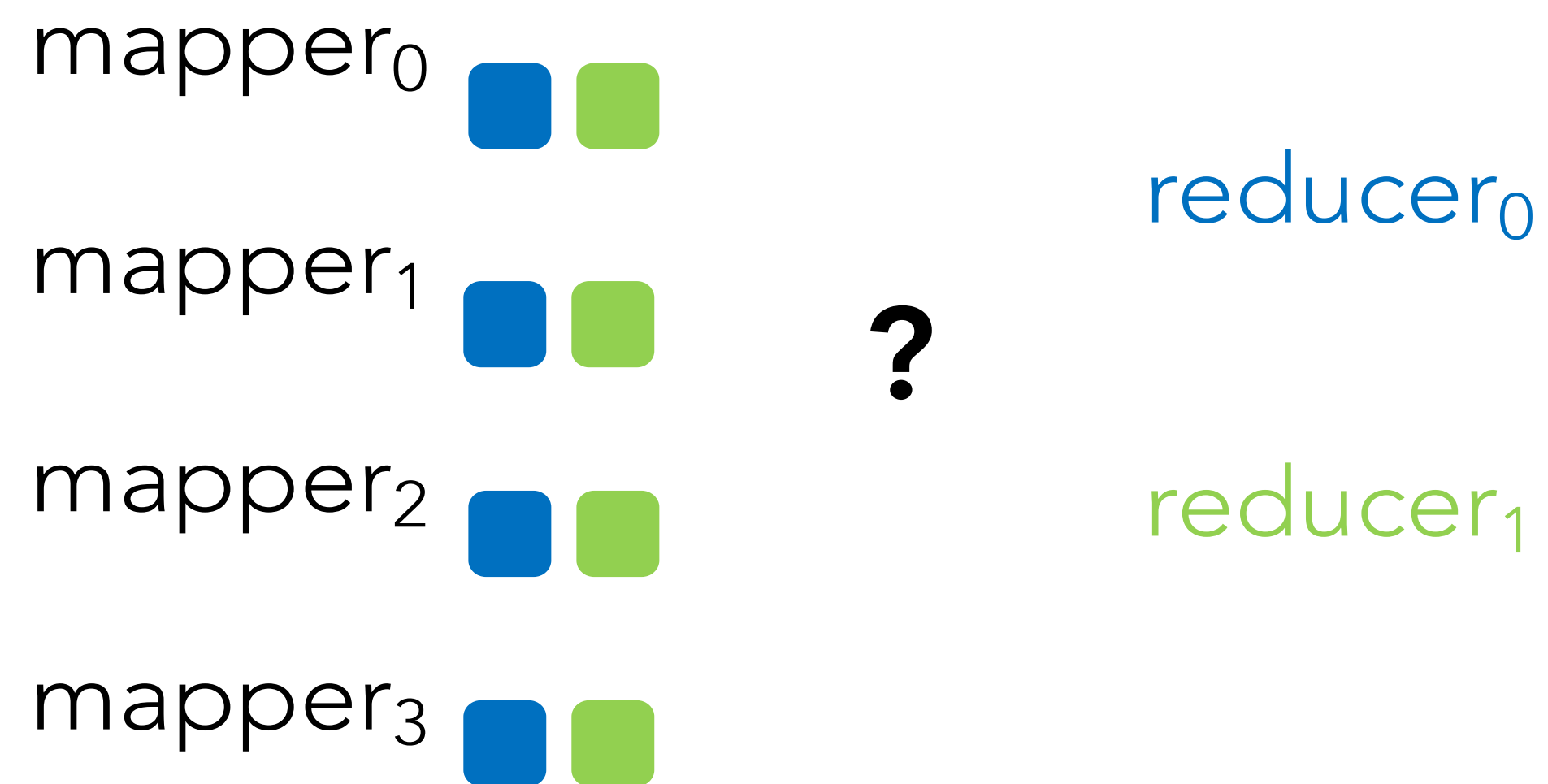
In traditional analytics...

- Ephemeral data is exchanged directly between tasks



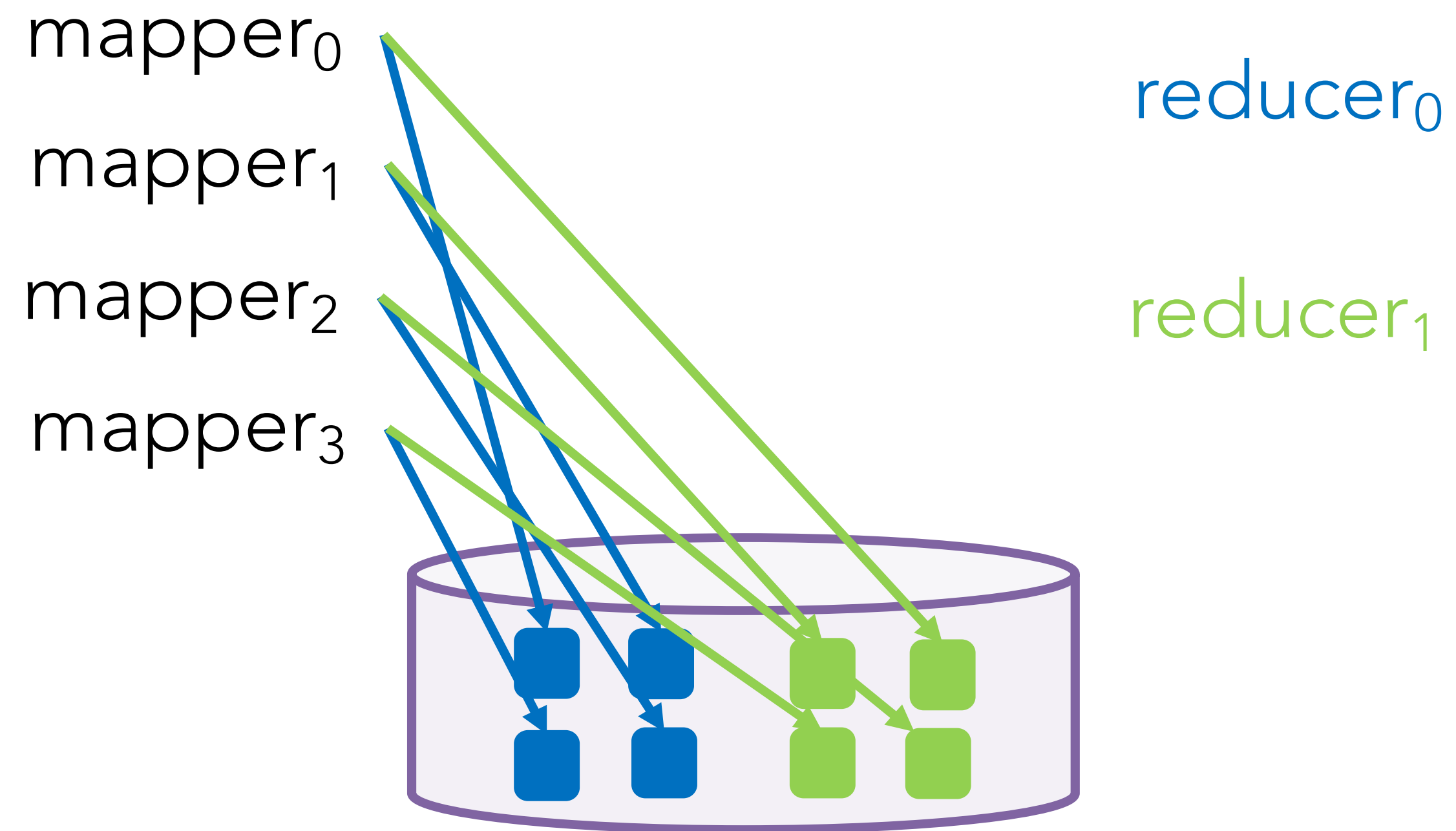
In serverless analytics...

- Direct communication between serverless tasks is difficult:
 - Tasks are short-lived and stateless



In serverless analytics...

- The natural approach for sharing ephemeral data is through a **common data store**



In serverless analytics...

- The natural approach for sharing ephemeral data is through a **common data store**

mapper₀

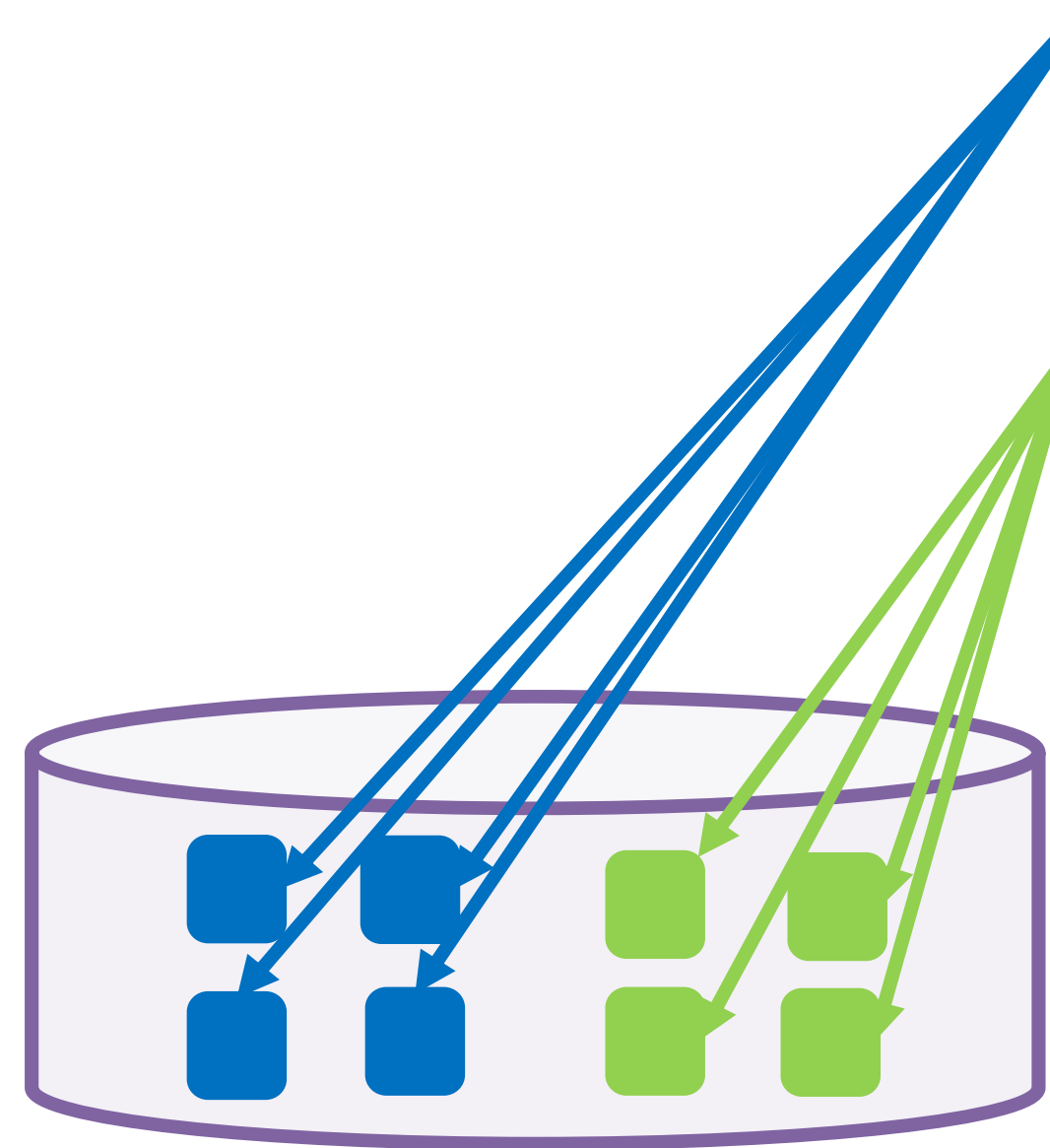
mapper₁

mapper₂

mapper₃

reducer₀

reducer₁

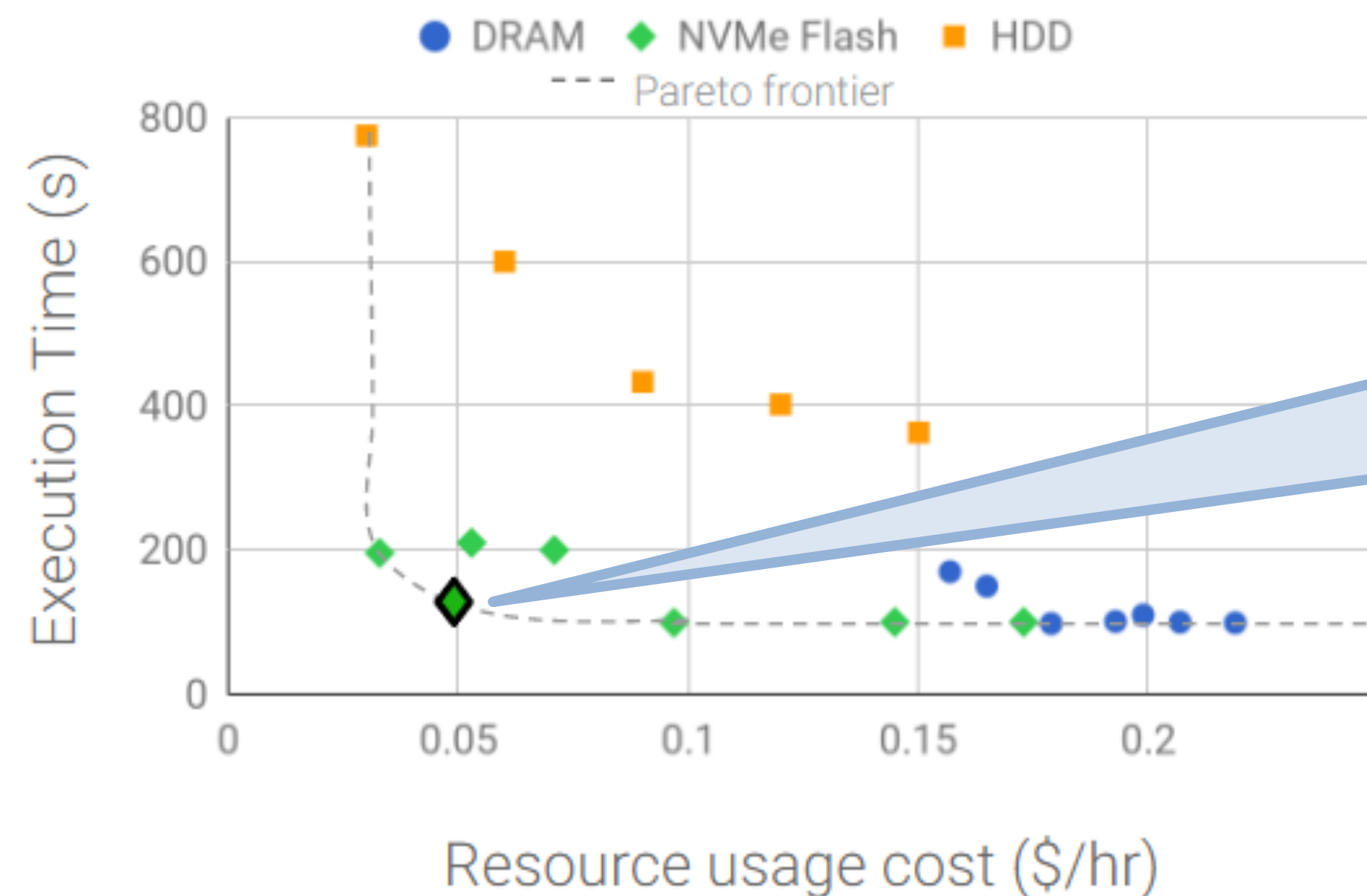


Requirements for Ephemeral Storage

1. High performance for a wide range of object sizes
2. Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing

Requirements for Ephemeral Storage

1. High performance for a wide range of object sizes
2. Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing
 - Example of performance-cost tradeoff for a serverless video analytics job with different ephemeral data store configurations



Finding the Pareto optimal resource allocation is non-trivial...and gets harder with multiple jobs.

Requirements for Ephemeral Storage

1. High performance for a wide range of object sizes
2. Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing
3. ~~Fault-tolerance~~

Existing cloud storage systems do not meet the elasticity, performance and cost demands of serverless analytics jobs.

Pocket



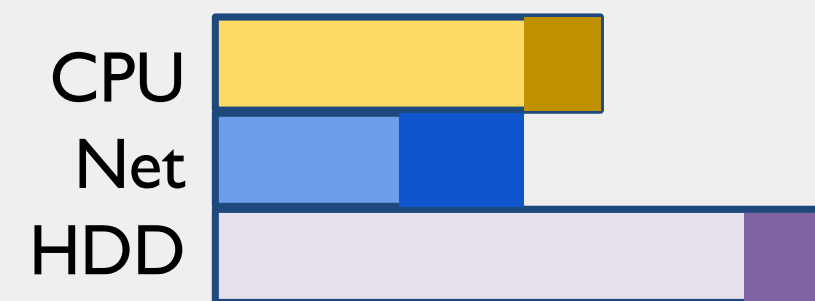
- An elastic, distributed data store for ephemeral data sharing in serverless analytics
- Pocket achieves high performance and cost efficiency by:
 - Leveraging multiple storage technologies
 - Rightsizing resource allocations for applications
 - Autoscaling storage resources in the cluster based on usage
- Pocket achieves similar performance to Redis, an in-memory key value store, while saving ~60% in cost for various serverless analytics jobs

Pocket Design

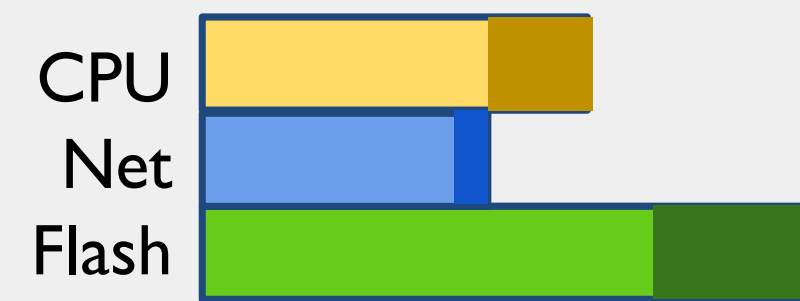
Controller
*app-driven resource
allocation & scaling*

Metadata server(s)
request routing

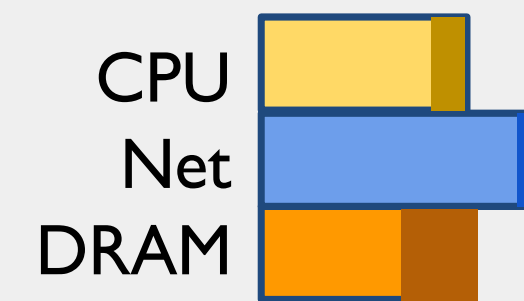
Storage server



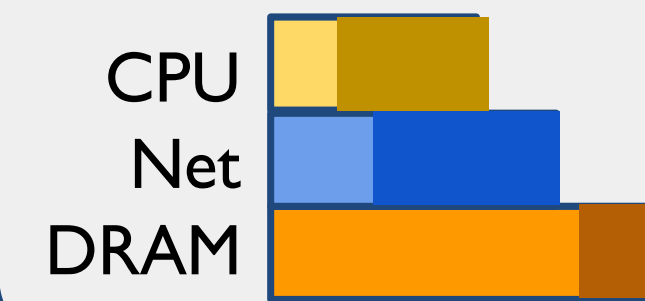
Storage server



Storage server



Storage server

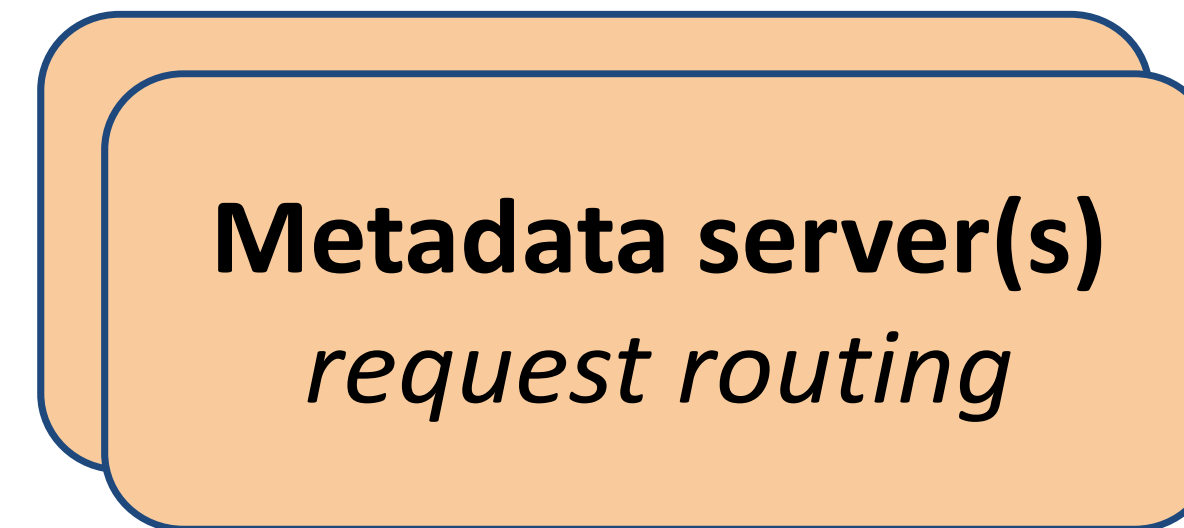
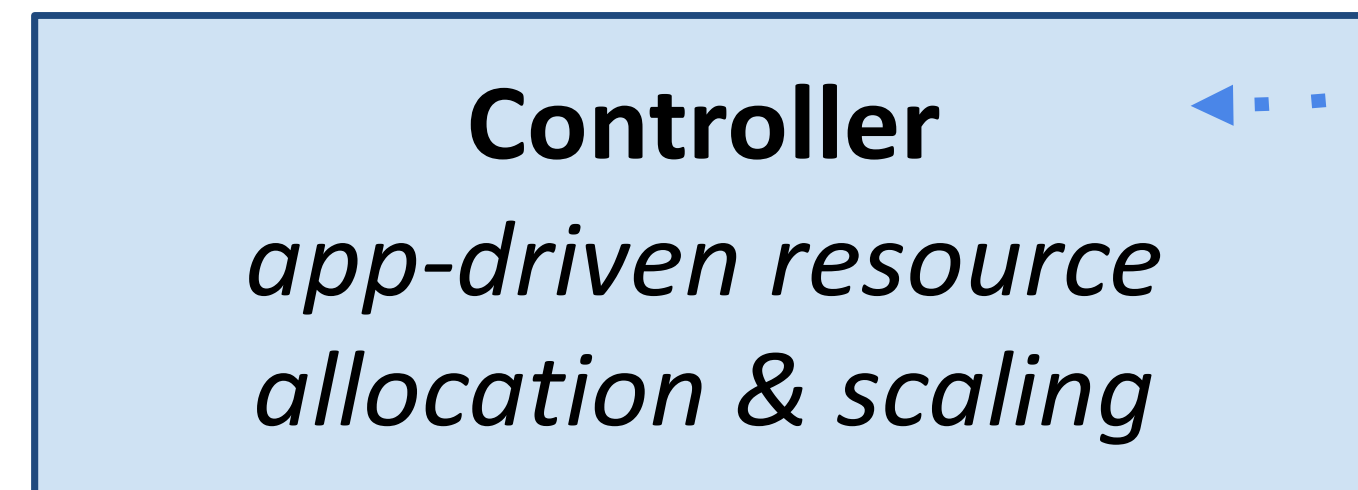


Using Pocket

Job A
λλλλλλλλ
λλλλλλλλ

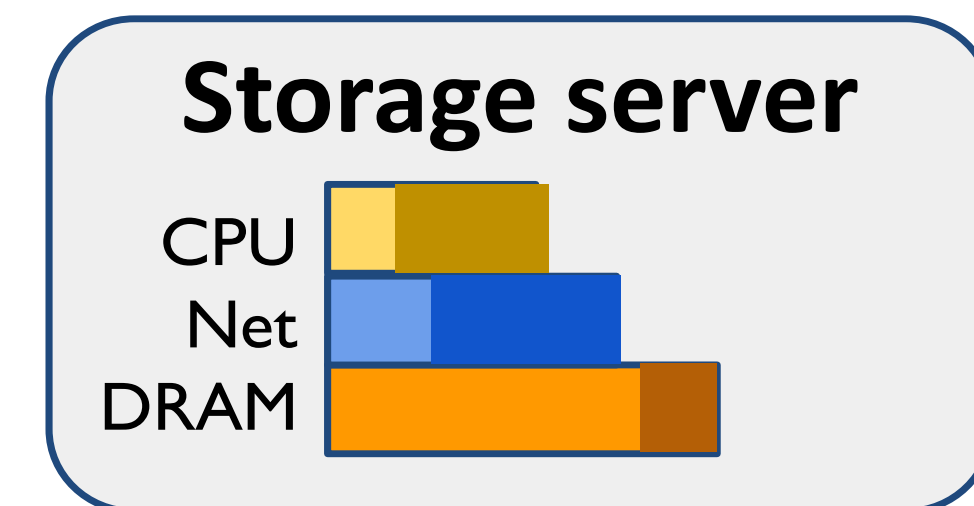
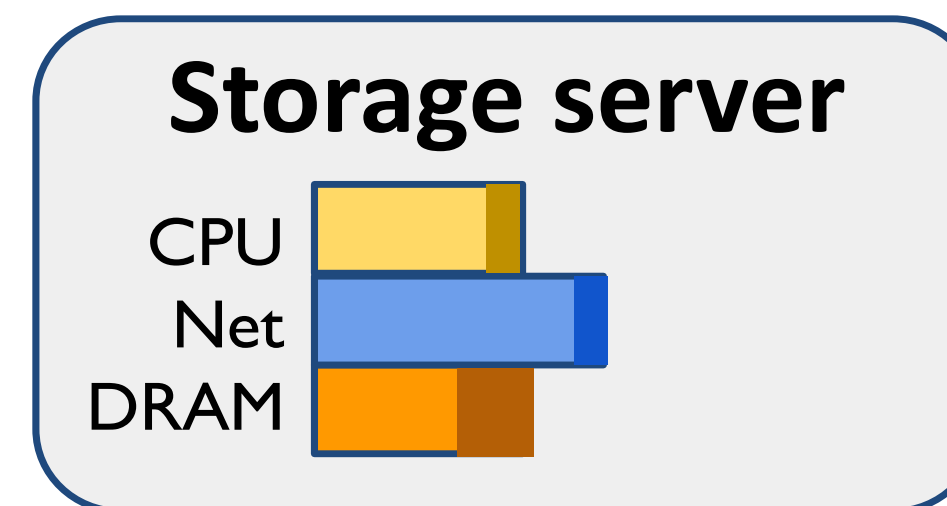
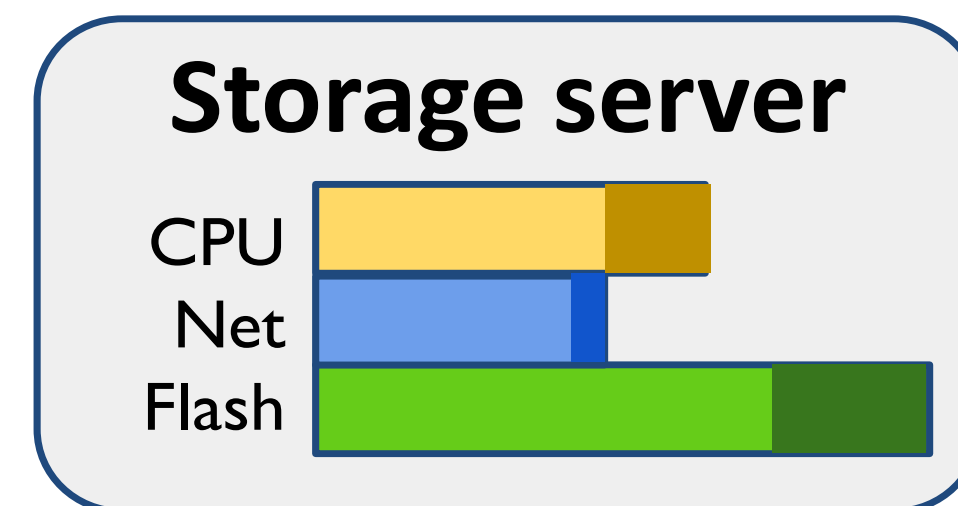
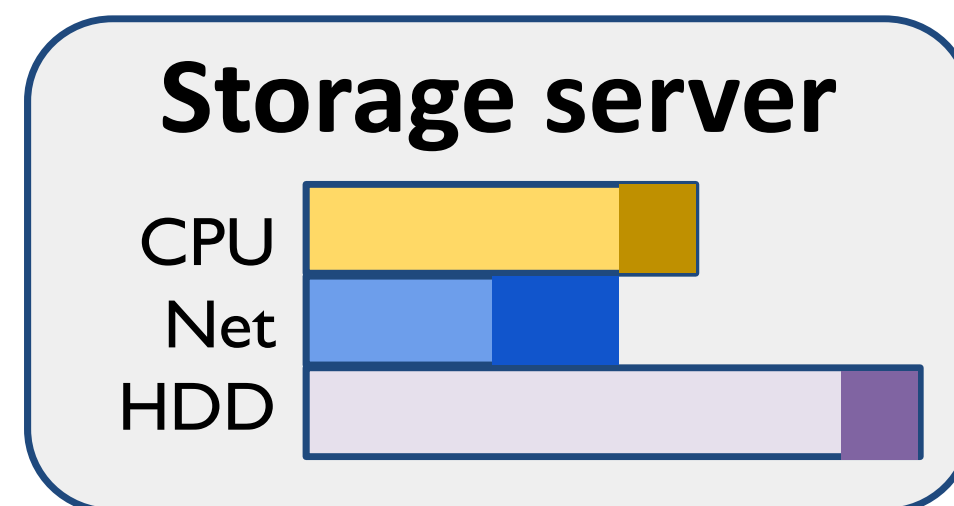
Job B
λλλλλλ
λλλλ

Job C

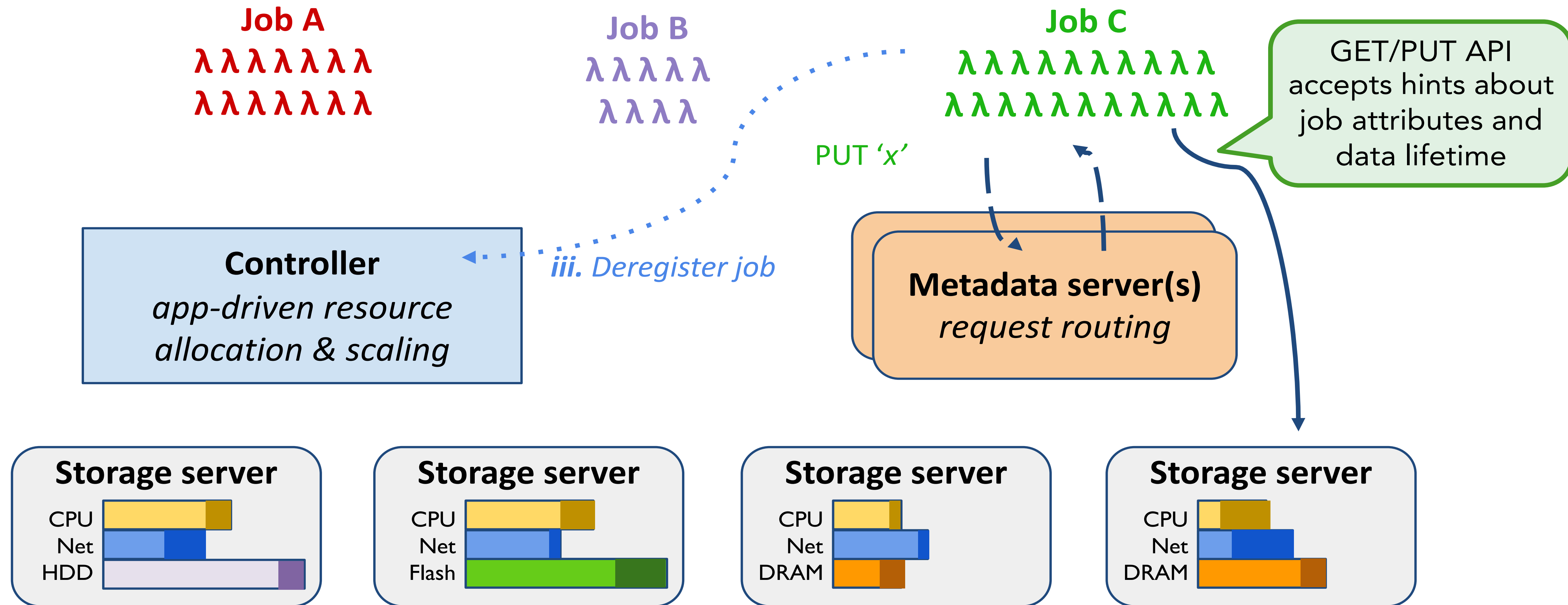


i. Register job

*ii. Allocate & assign
resources for job*

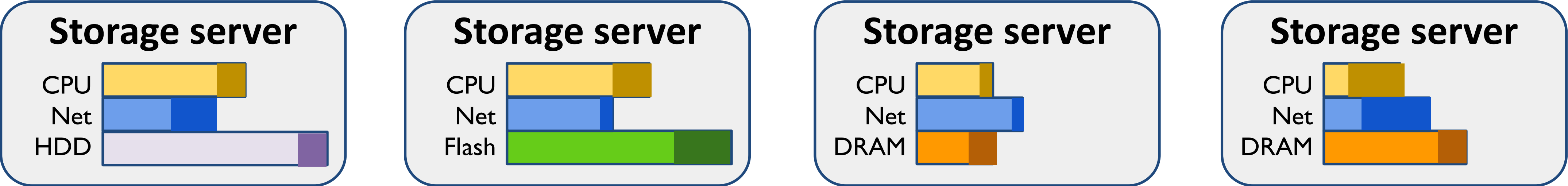
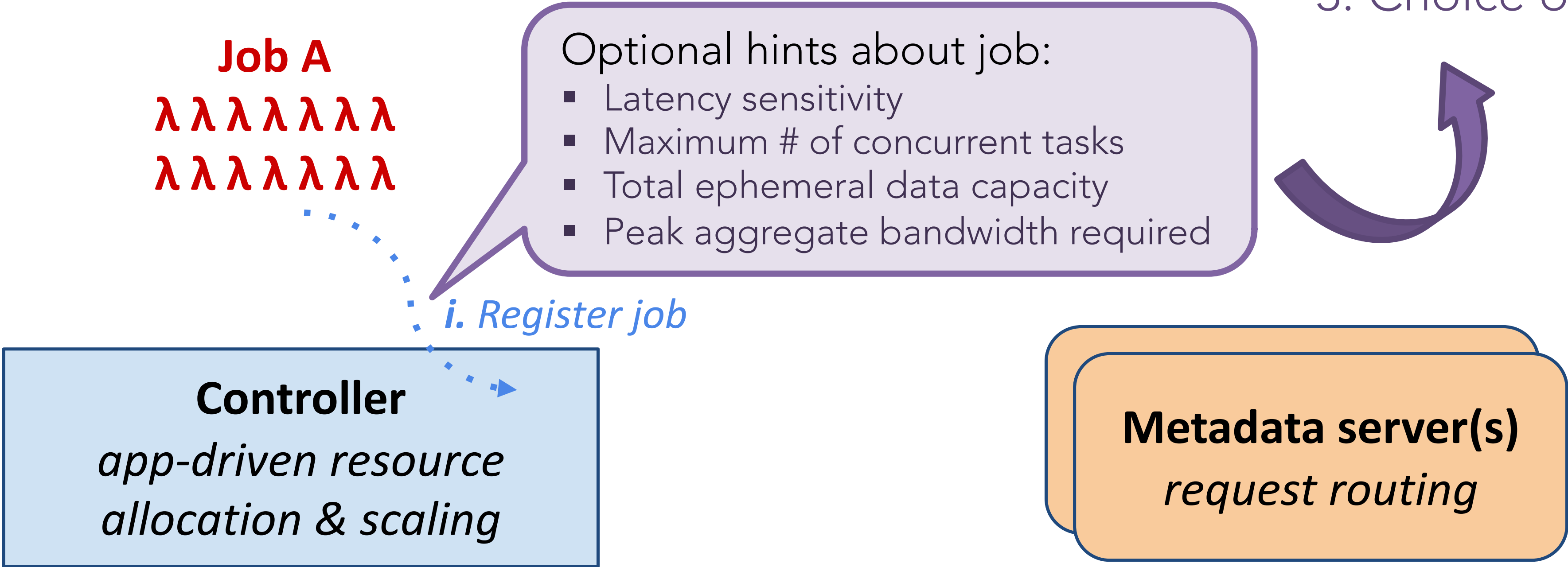


Using Pocket



Assigning Resources to Jobs

- 1. Throughput allocation
- 2. Capacity allocation
- 3. Choice of storage tier(s)



Assigning Resources to Jobs

- 1. Throughput allocation
- 2. Capacity allocation
- 3. Choice of storage tier(s)



Job Weight Map

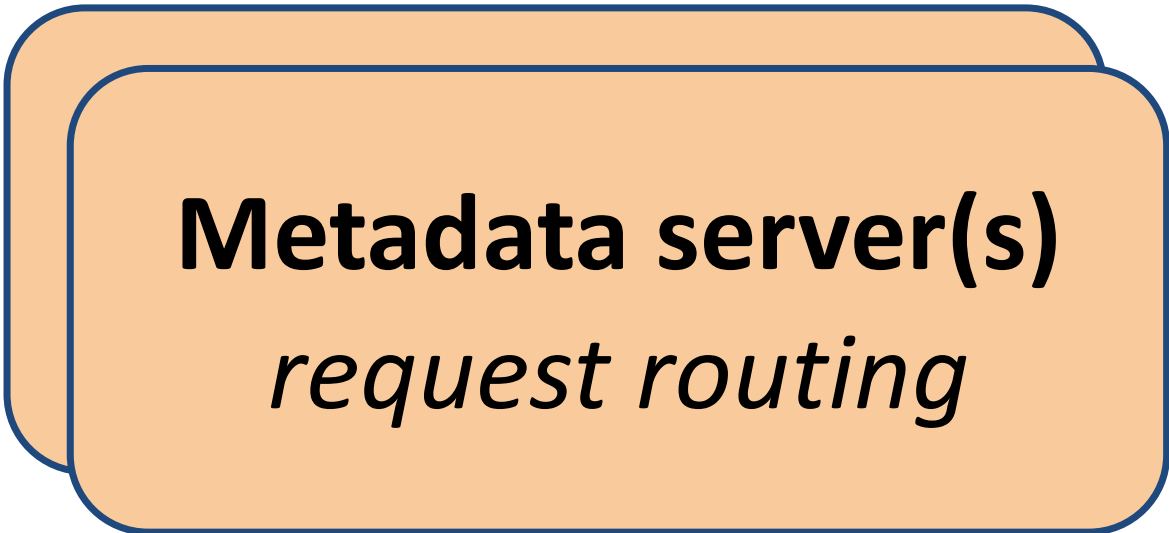
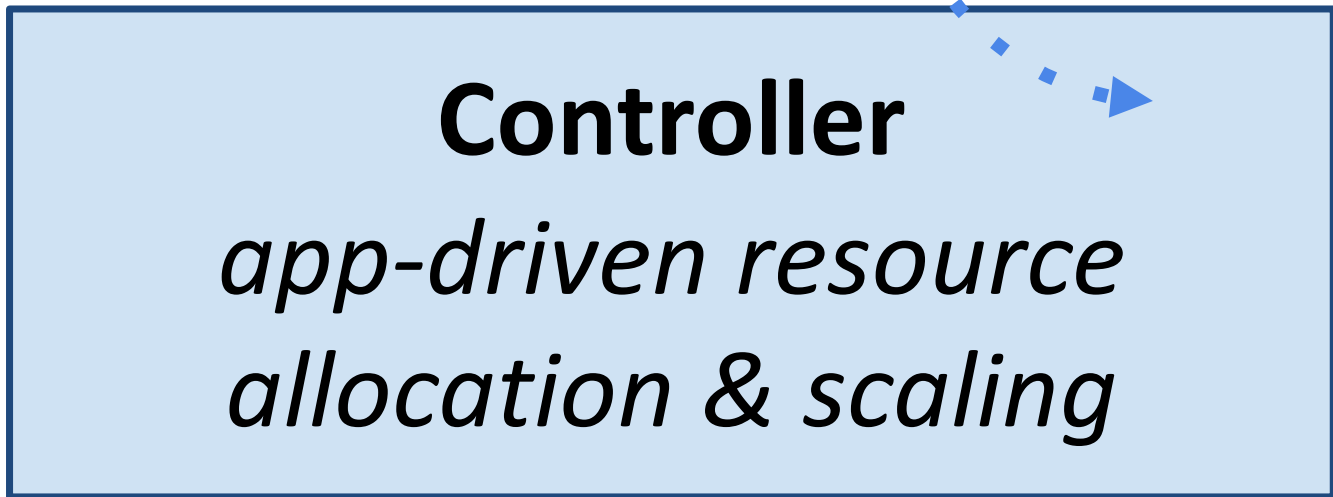
Job A:

Server C	→	0.4
Server D	→	0.6

Job B:

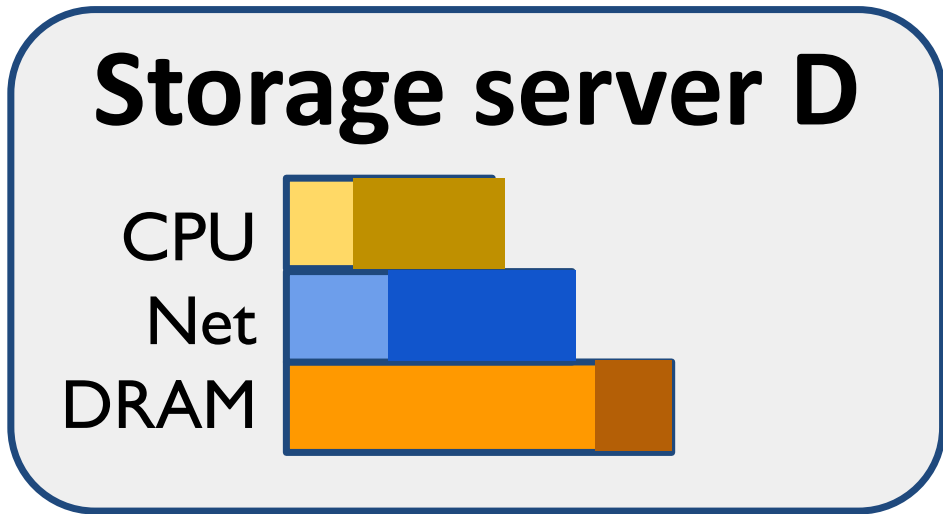
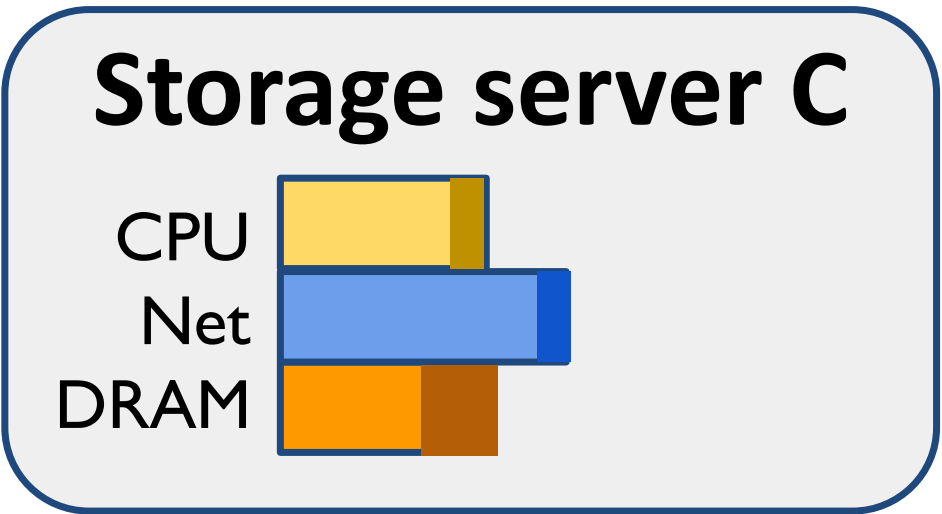
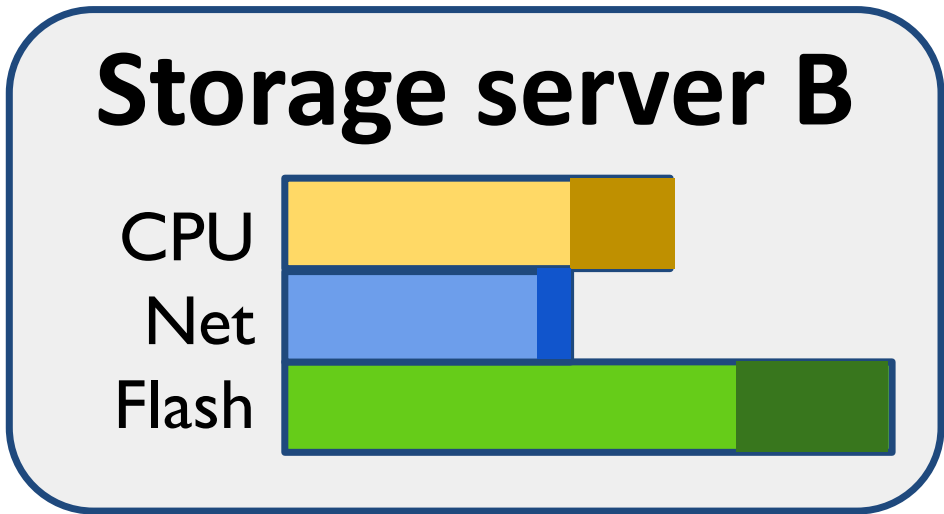
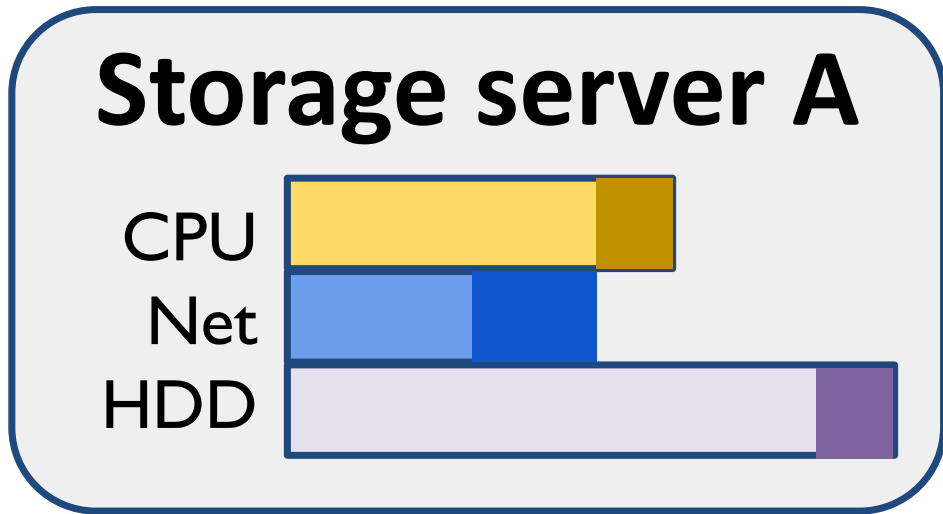
Server A	→	0.2
Server B	→	0.3
Server C	→	0.5

Job A
λλλλλλλλ
λλλλλλλλ



i. Register job

ii. Allocate & assign resources for job

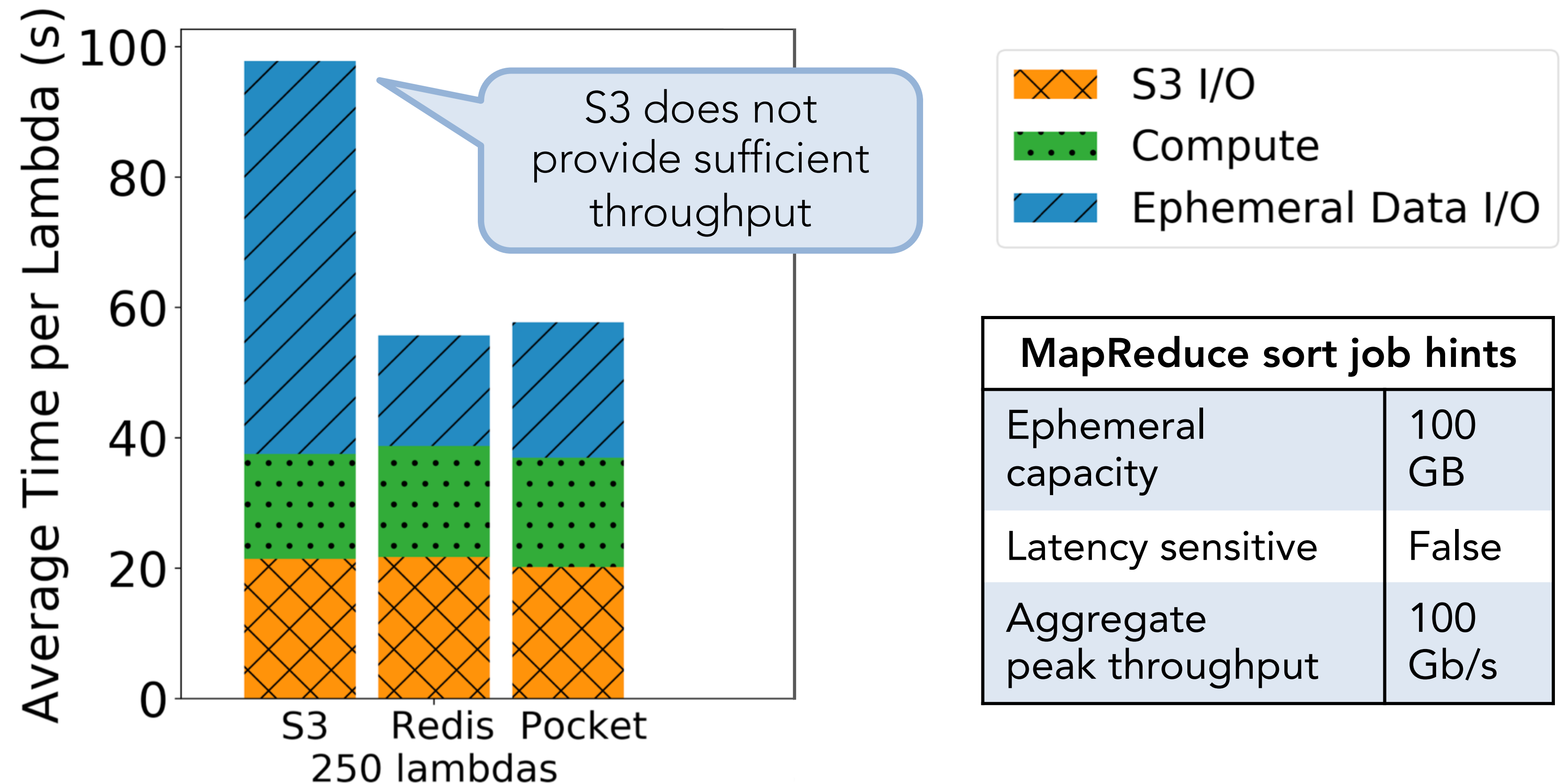


Autoscaling the Pocket Cluster

- Goal: scale cluster resources dynamically based on resource usage
- Mechanisms:
 - Monitor CPU, network bandwidth, and storage capacity utilization
 - Add/remove storage & metadata nodes to keep utilization within range
 - Steer data for incoming jobs to active nodes
 - Drain inactive nodes as jobs terminate
- Avoid migrating data

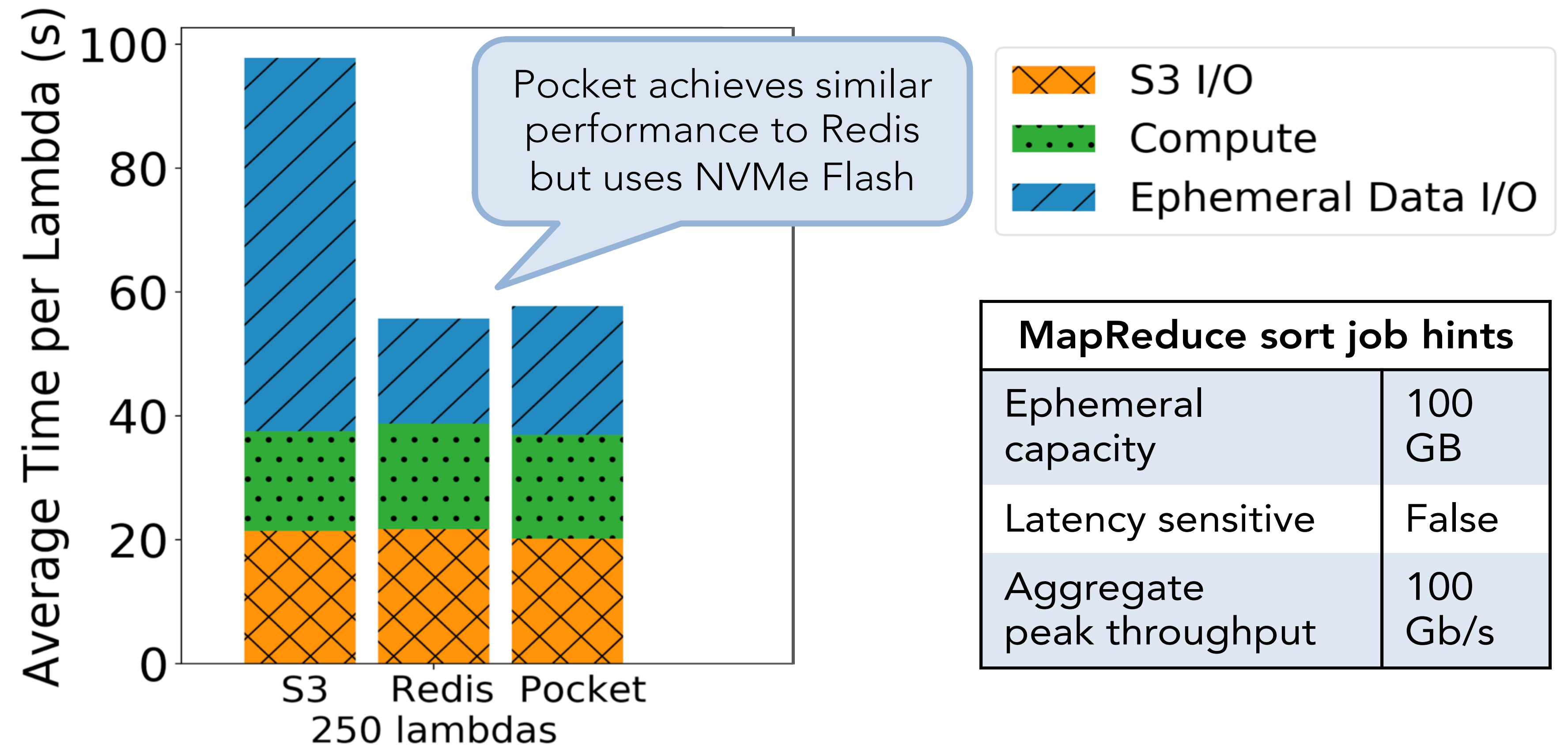
Application Performance with Pocket

- Compare Pocket to S3 and Redis, which are commonly used today



Application Performance with Pocket

- Compare Pocket to S3 and Redis, which are commonly used today



Conclusion

- Pocket is a distributed ephemeral storage system that:
 - Leverages multiple storage technologies
 - Rightsizes resource allocations for applications
 - Autoscales storage cluster resources based on usage
- We designed Pocket for ephemeral data sharing in serverless analytics. More generally, Pocket is an elastic, distributed /tmp.

www.github.com/stanford-mast/pocket



Outline

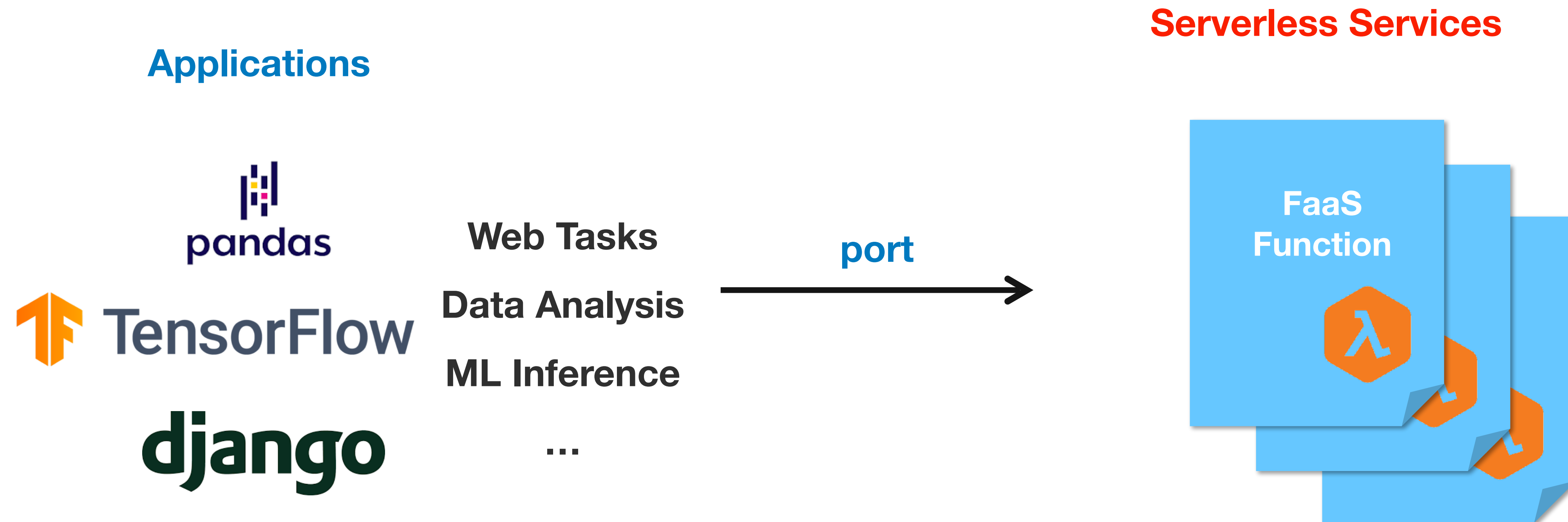
- Managing states: Pocket
- Resource efficiency: Zenix

Serverless Computing

- **No Management:** Users do not manage server or provision resources
- **Auto-Scale:** User computation is event triggered and auto scaled
- **Cost Efficient:** Users only pay for what they use



Serverless Computing

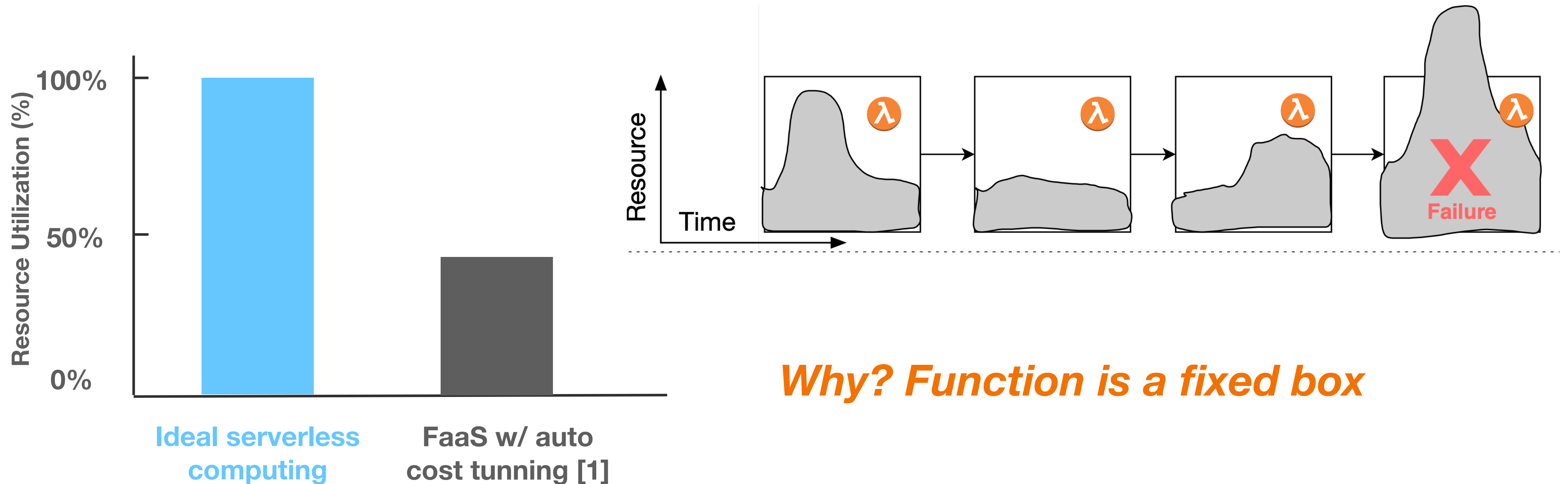


a **Function-as-a-Service (FaaS)** model:
Adapting user applications to **Fix-Sized Functions**

Does FaaS Meet All Serverless Goals?

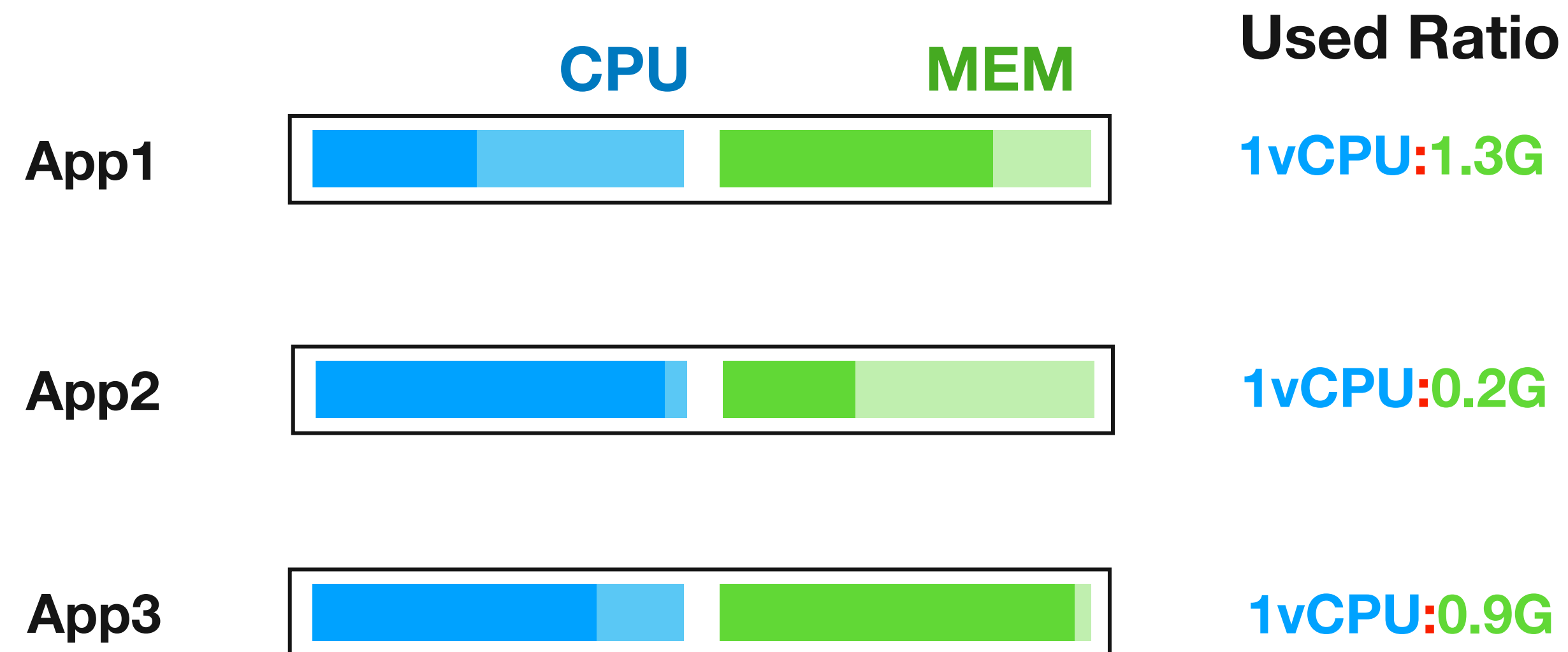
- **No Management:** Users do not manage server or provision resources
- **Auto-Scale:** User computation is event triggered and auto scaled
- **Cost Efficient:** any allocated but unused resources?

FaaS is Resource Inefficient



- Running Logistic Regression with 10 different input data samples

FaaS Issue 1: Fixed Resource Type Ratios



CPU & memory allocated in **fixed ratios**

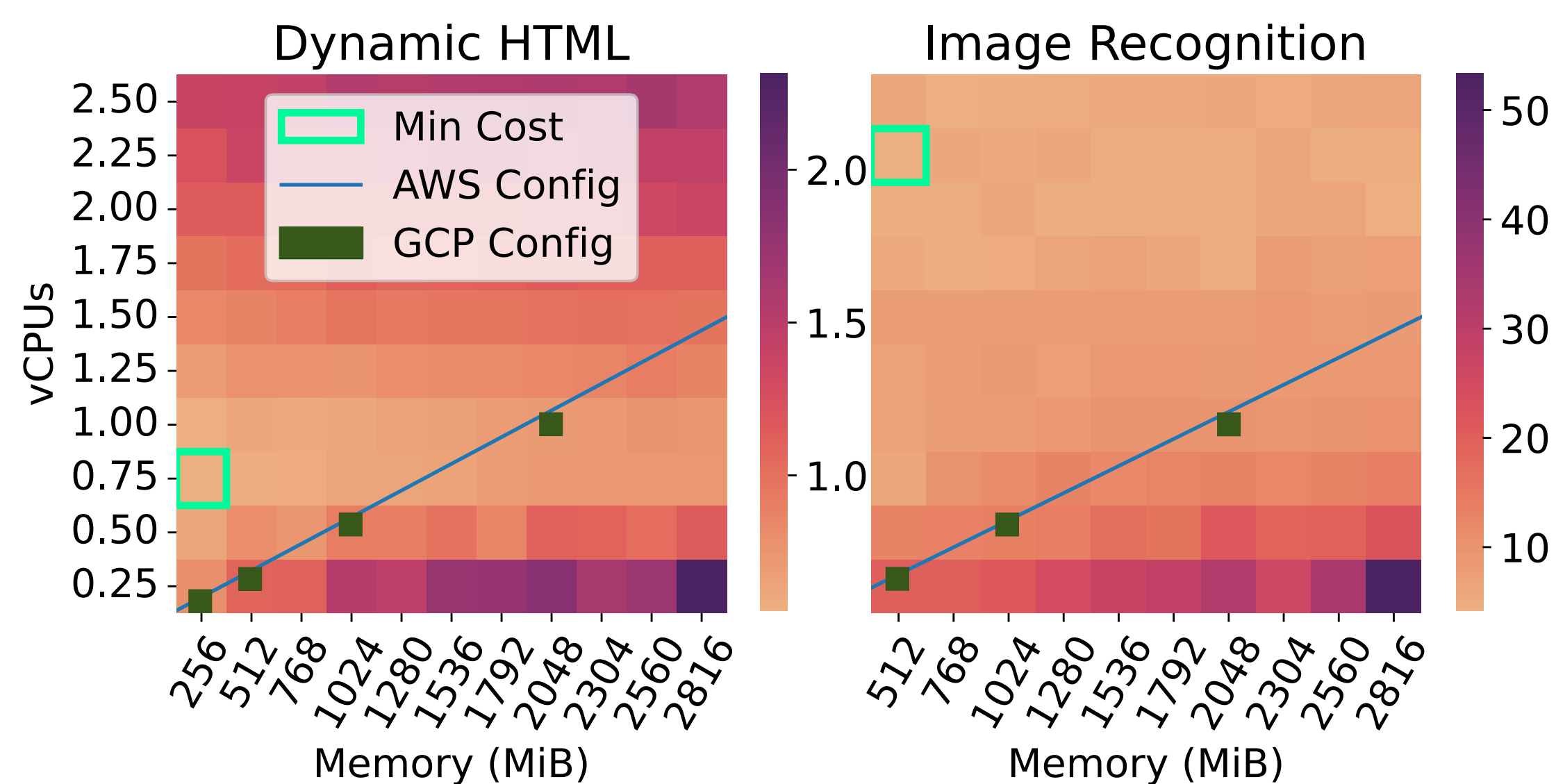
=> User applications' needs are not!

AWS Lambda Allocation:

Fixed Ratio, 1 vCPU : 1.7G Memory

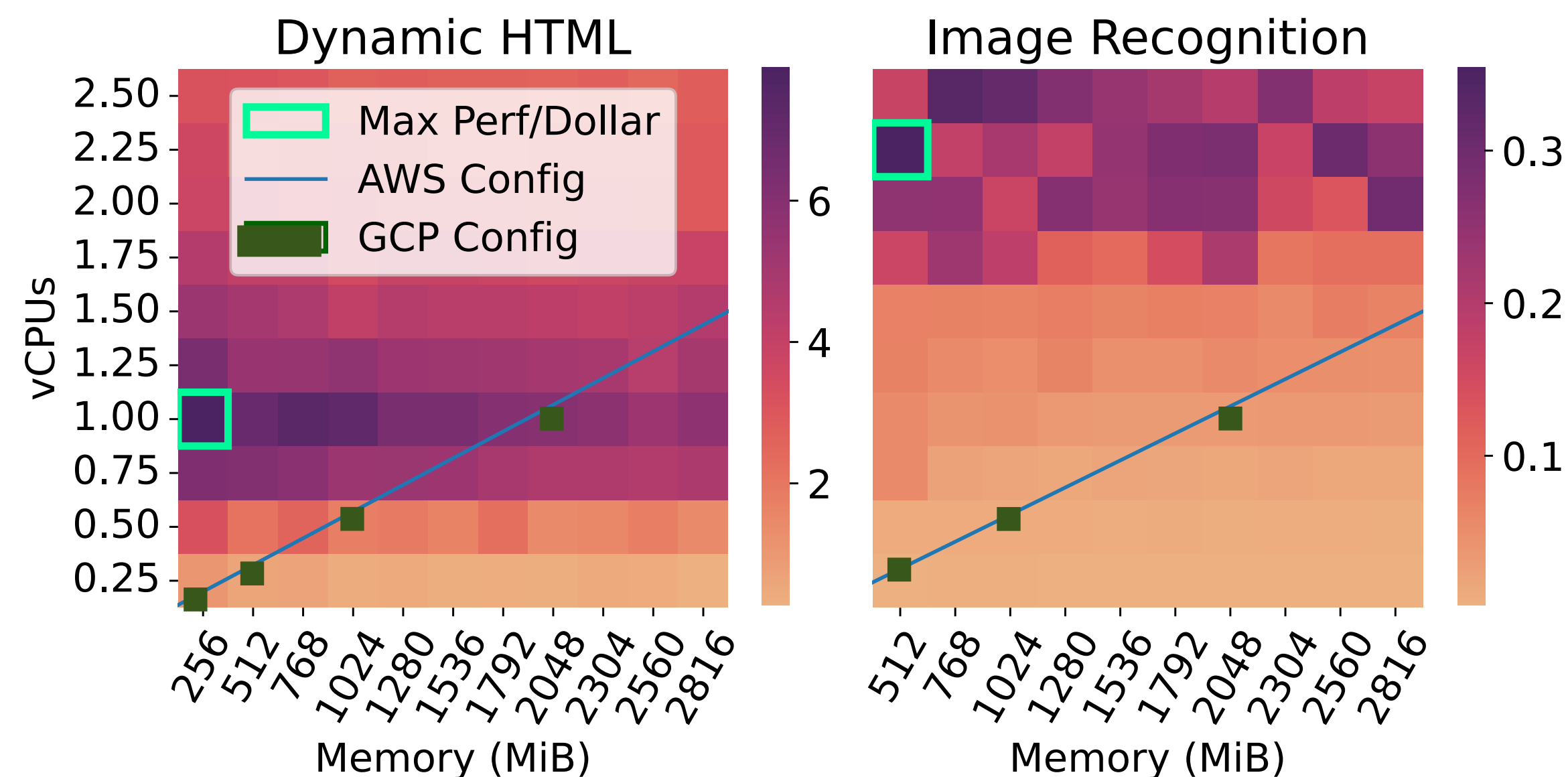
* Data Collected from Image Recognition, HTML and MapReduce examples, from SeBS benchmarks

FaaS Issue 1: Fixed Resource Type Ratios



Resource Consumption Heatmap

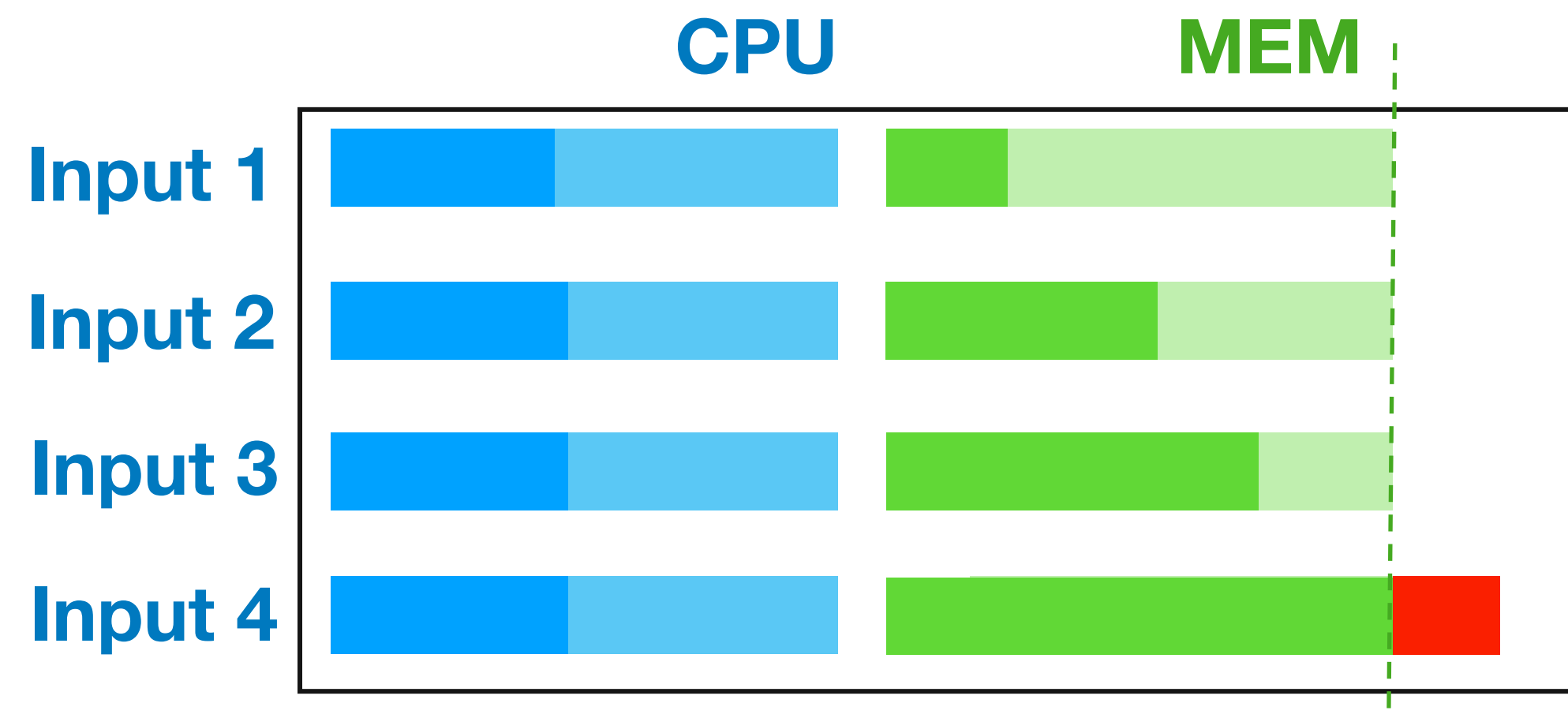
lighter color: lower cost



Perf-per-Dollar Heatmap

darker color: better perf/\$

FaaS Issue 2: Fixed Amount across Invocations



Uses set **one function size across invocations**

=> Different inputs cause different usages

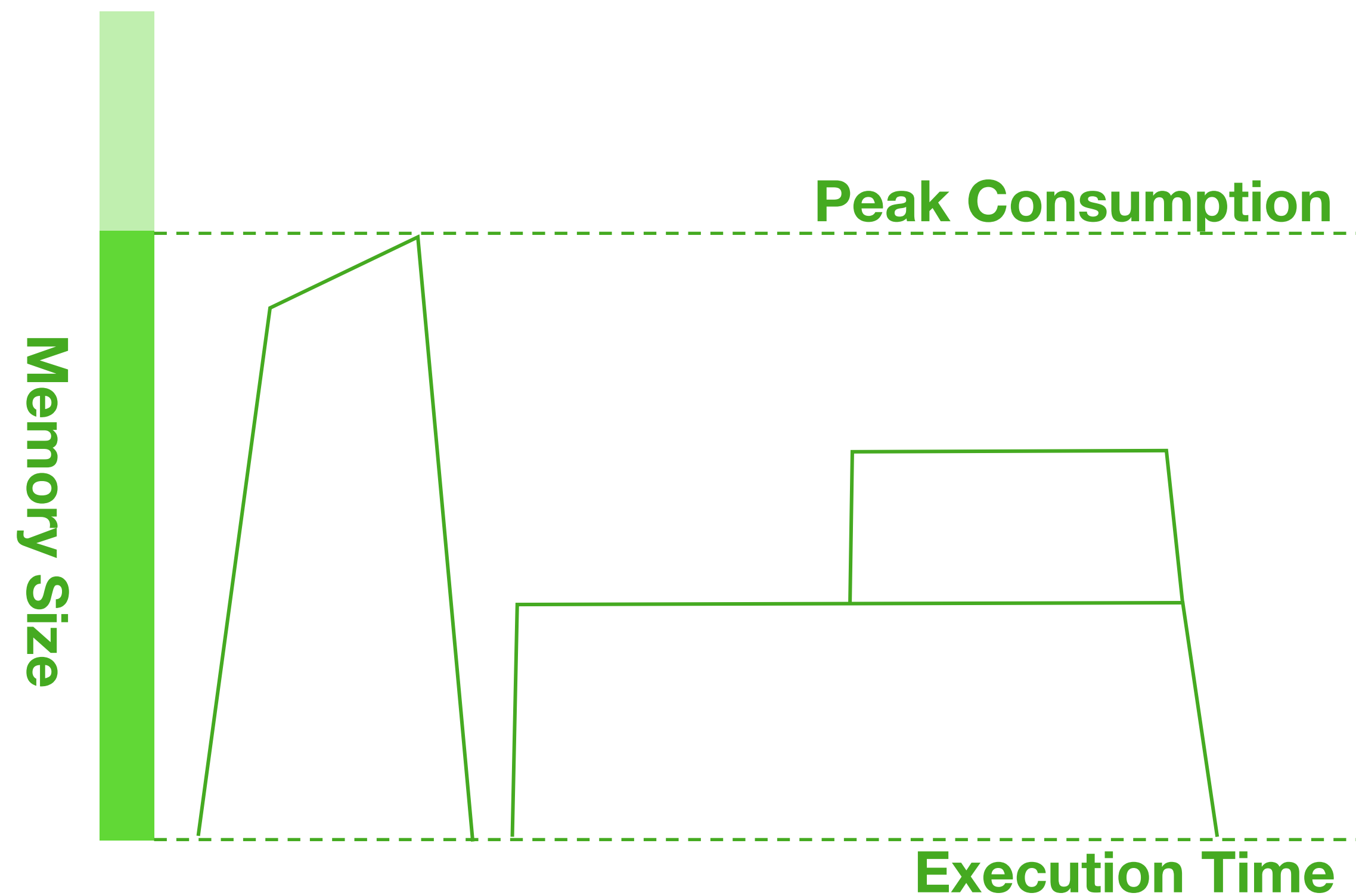
Provision for the peak?

Resource wastage for non peaks

Provision for avg?

Execution fails when larger than avg

FaaS Issue 3: Fixed Amount throughout Execution



One function size **throughout execution**
=> Resource usage changes in real apps

*Image Processing Example. On a large size image, each line showing one object.

Running “Bulky” Applications on FaaS

- Bulky applications like data analytics, ML, video processing, scientific computing
 - Exhibit resource usage variations across different phases of computation
 - Require different amounts of resources with different inputs
 - Run longer or consume more memory than function size limits
- Running bulky applications on today’s FaaS
 - Manually split or rewrite a program as a set of functions
 - Provider increasing function sizes (AWS: 3GB to 10GB and 5 min to 15 min in 2020)

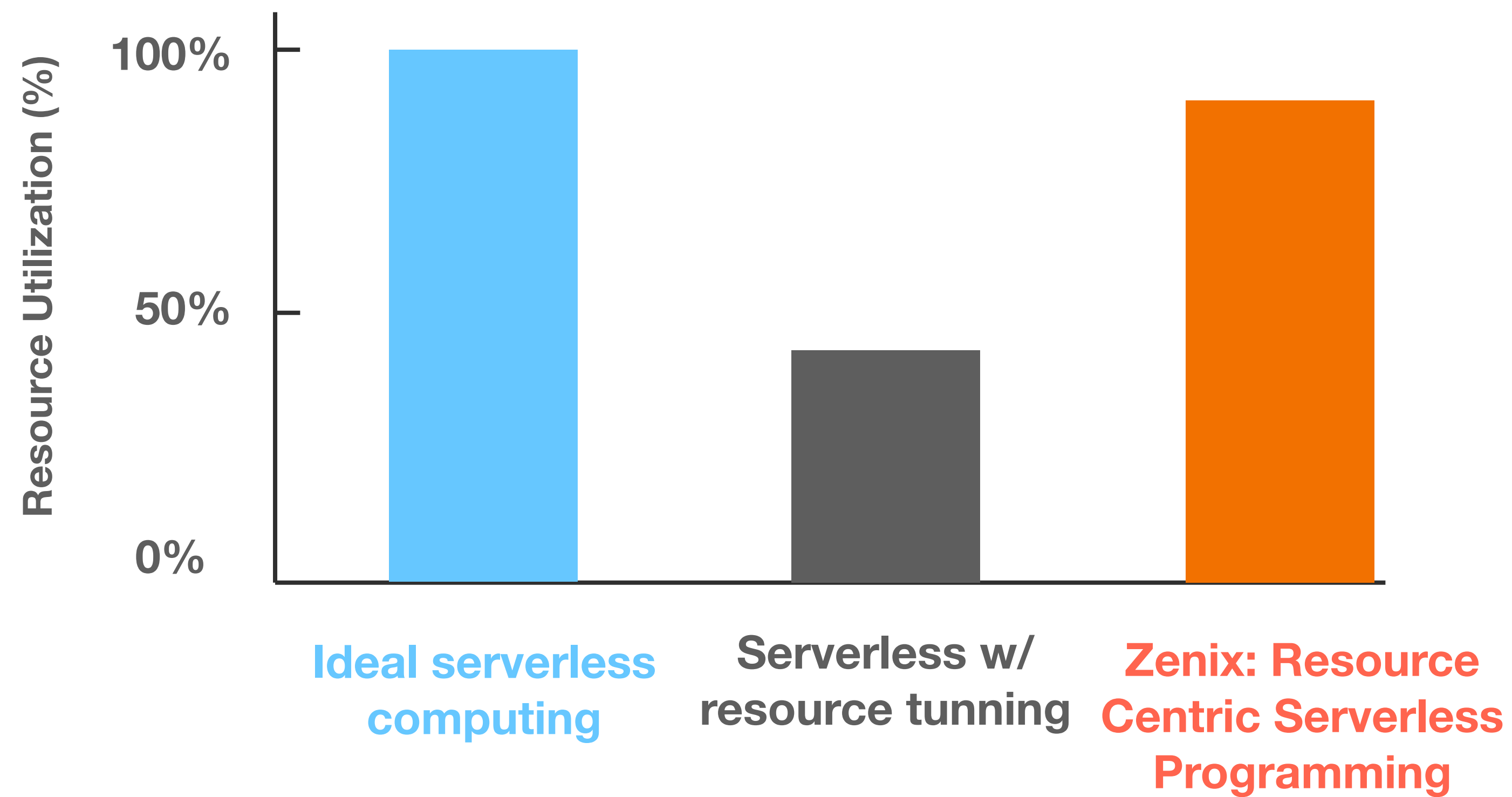
Root cause for FaaS issues:

***User Applications Adapt to
Provider-Defined Functions***

Resource-Centric Serverless Computing:

Adapt Execution Towards User Applications

Zenix: a Resource-Centric Serverless Computing Platform



*Logistic Regression Example with 10 different input data samples



Zenix: Resource-Centric Serverless Computing

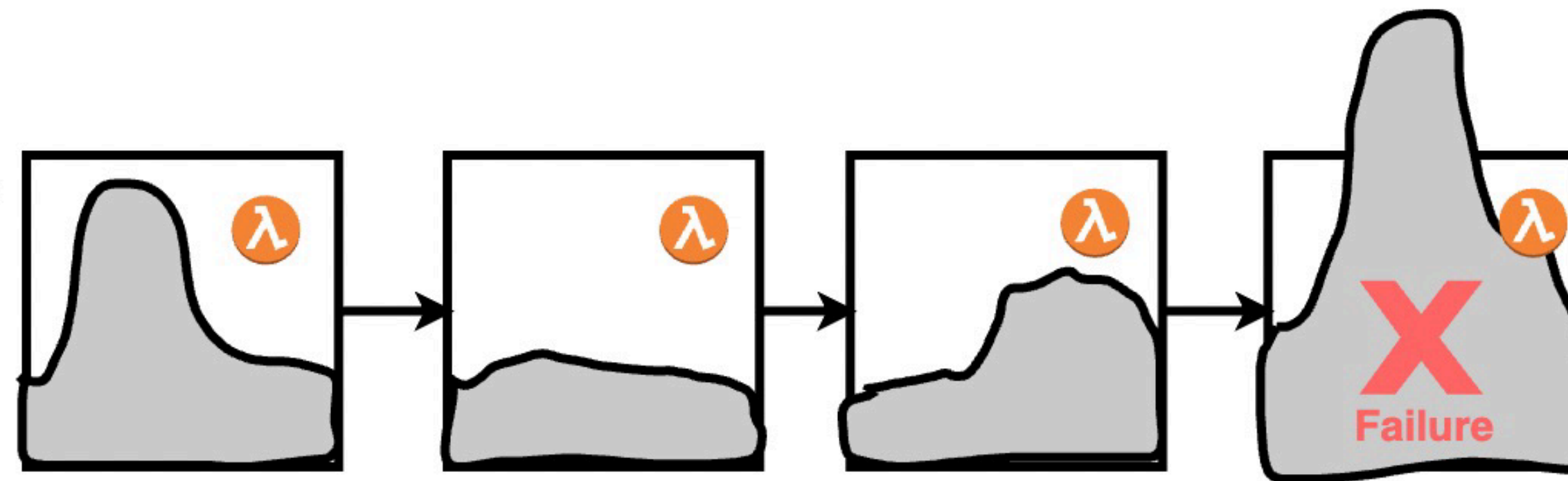
Center serverless computing around the resource features of applications by decomposing them on a disaggregated system

Resource-Centric Serverless Computing

- Decomposing and executing applications according to their *resource needs*

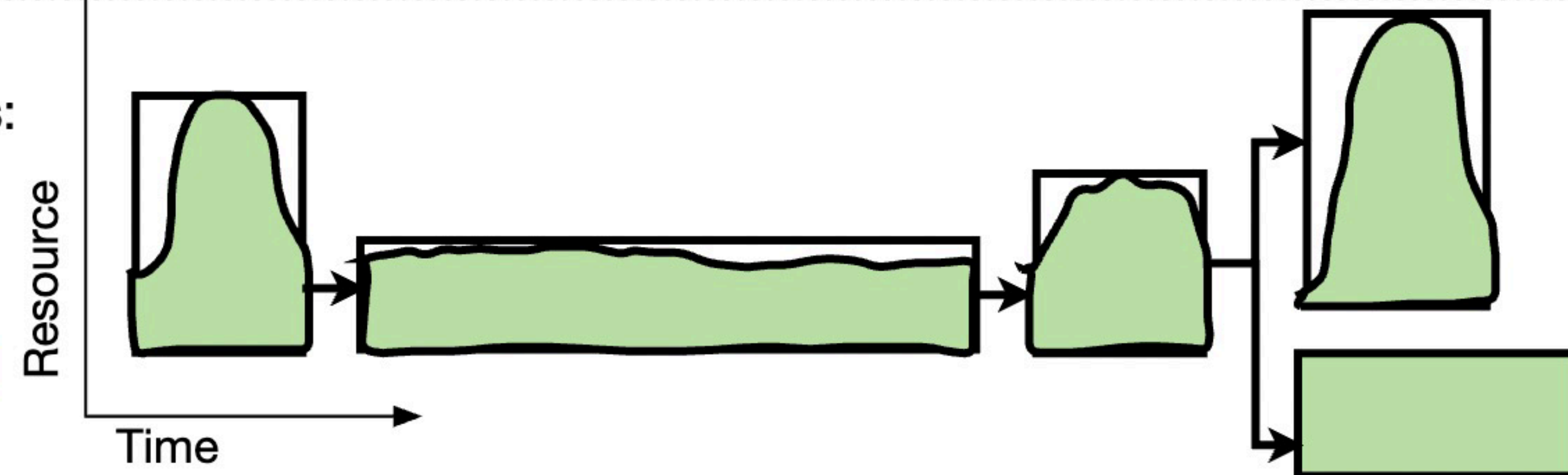
Function-Based Serverless:

Resource wastage
Resource limits
Potential failure
Manual application porting
Runtime overhead



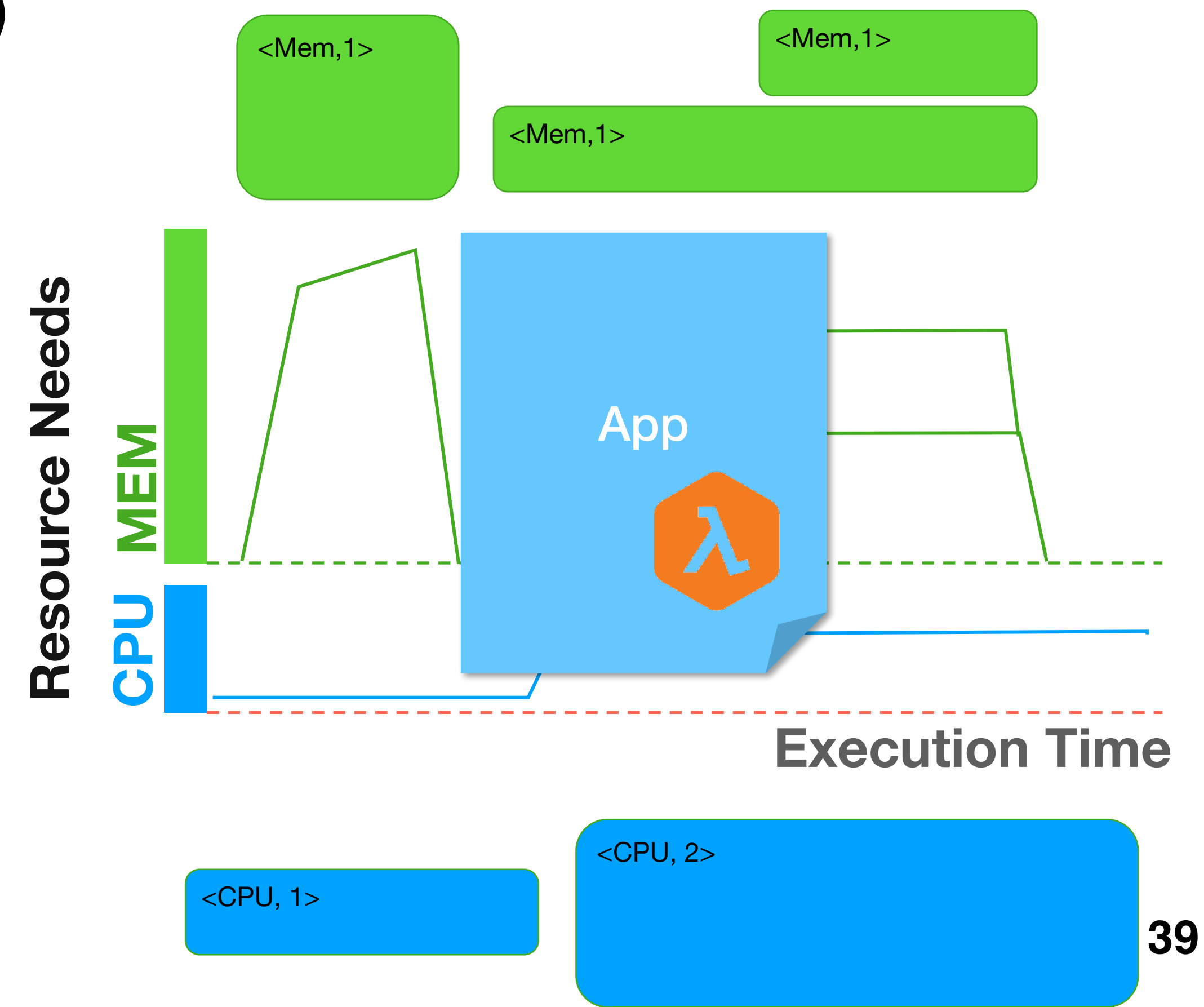
Resource-Based Serverless:

Tight resource packing
No resource limits
Fine-grained autoscaling
No or light manual porting
Efficient runtime perf



Resource Graph: A New Model for Capturing Application Resource Behavior

- Resource component (nodes in a resource graph)
 - A piece of code or data object that captures unique **<type, lifetime, amount>**
 - Each component **autoscaled on its own**
 - Scale *without limits*, and adapt to each input

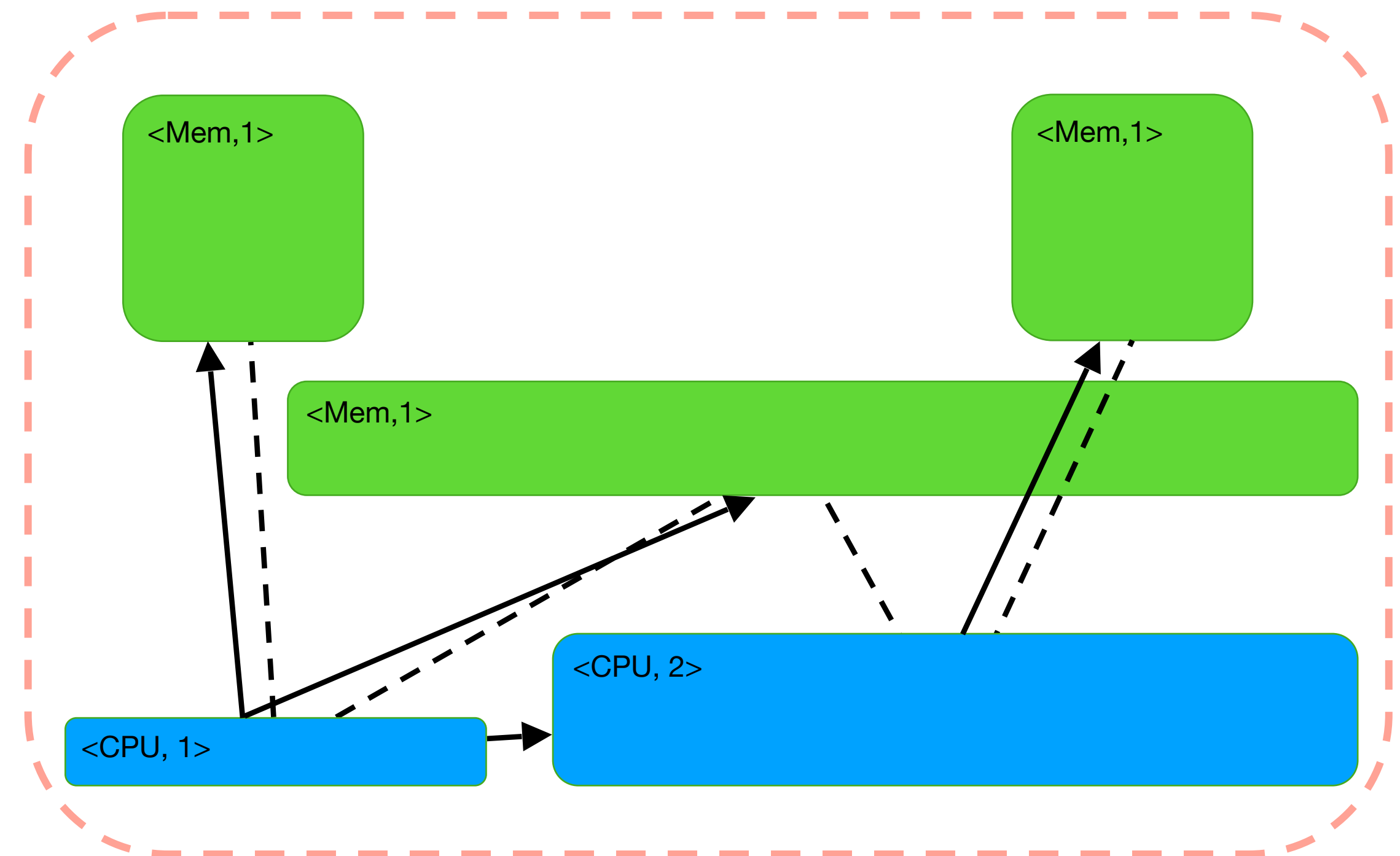


Resource Graph: A New Model for Capturing Application Resource Behavior

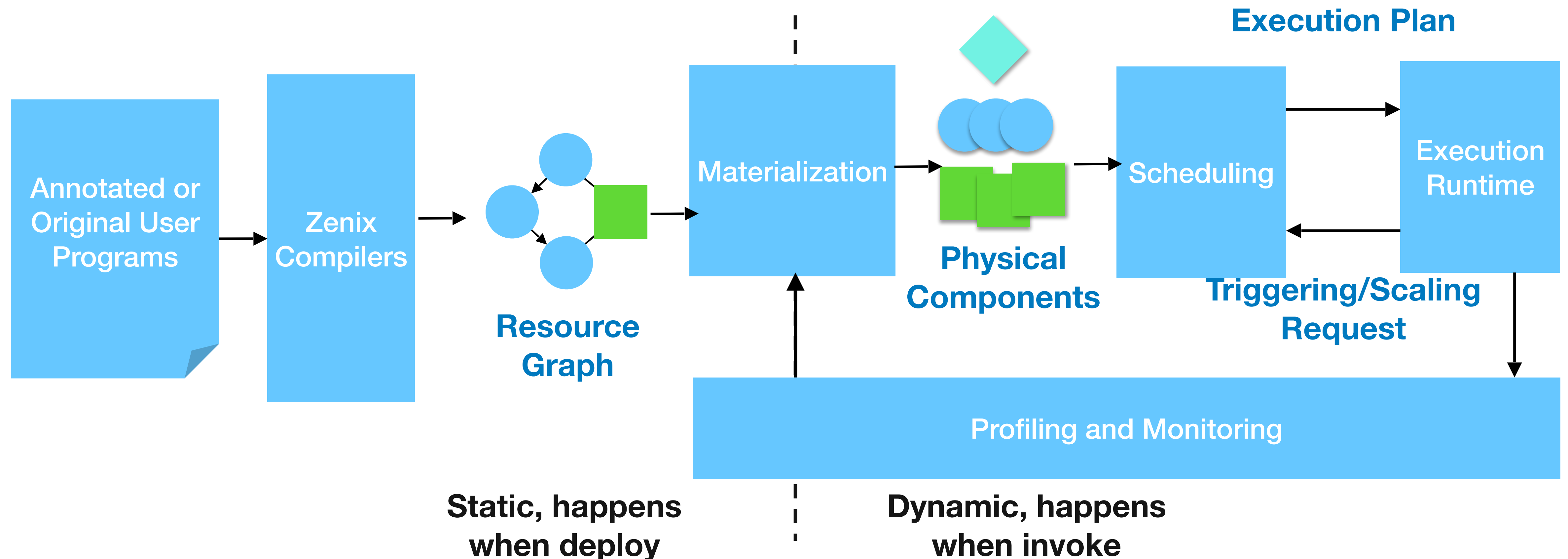
Edges in a resource graph:

→ **Triggering** another component

- - - - - **Communicating** with other components



Zenix: A Resource-Centric Serverless Computing Platform

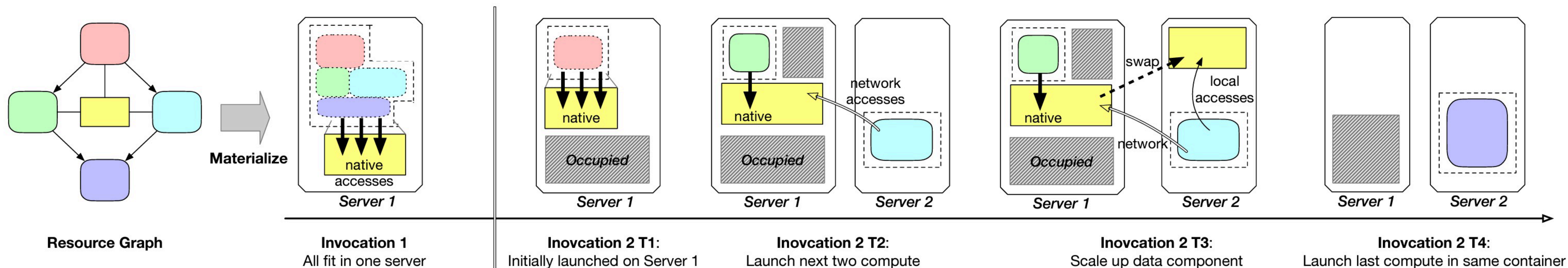


Resource Graph Generation

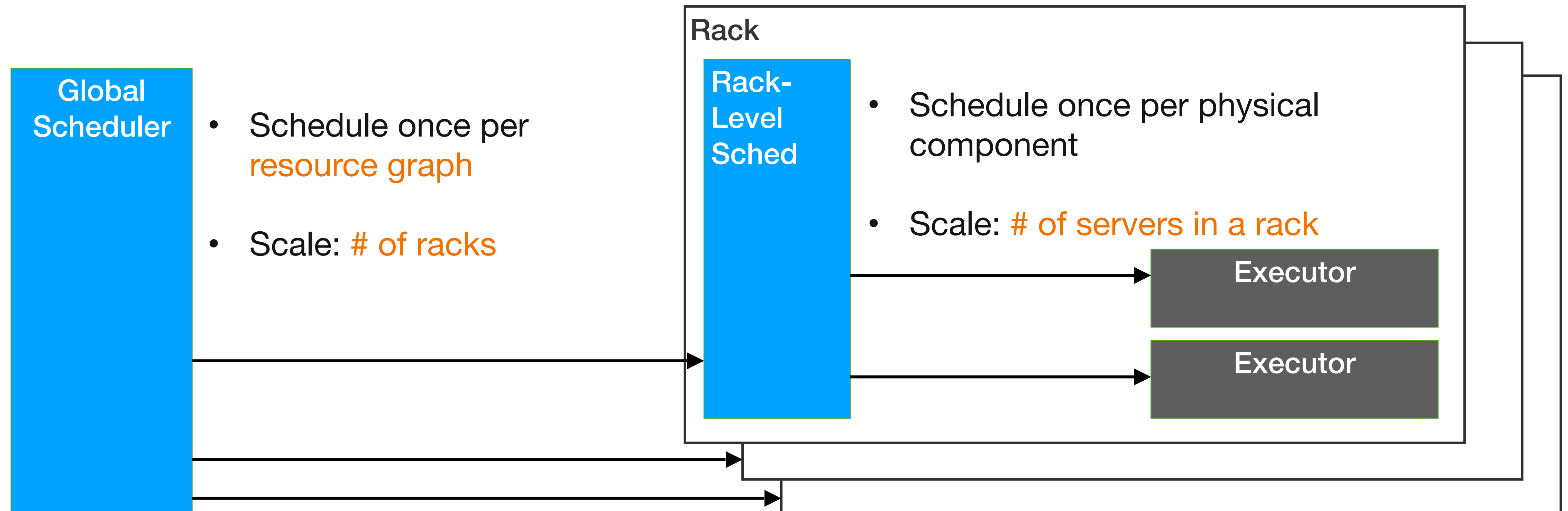
- Users annotate their code with *@compute (par=K)* and *@data*
- or, Zenix automatically inserts annotations at every function and data object
- Profile: object size, CPU utilization
- Determine graph components: group by CPU util and object lifetime
- Compilation: generating both a local and a disaggregated version (with Mira)

Adaptive Materialization

- Co-locate if possible (no reservation, but mark future resources as low priority for other scheduling)
- Co-locate compute components and their accessed data components => low comm overhead
- Co-locate triggered and triggering compute components => no startup overhead
- Sizes each component according to profiled history
- Adaptively scales components on demand



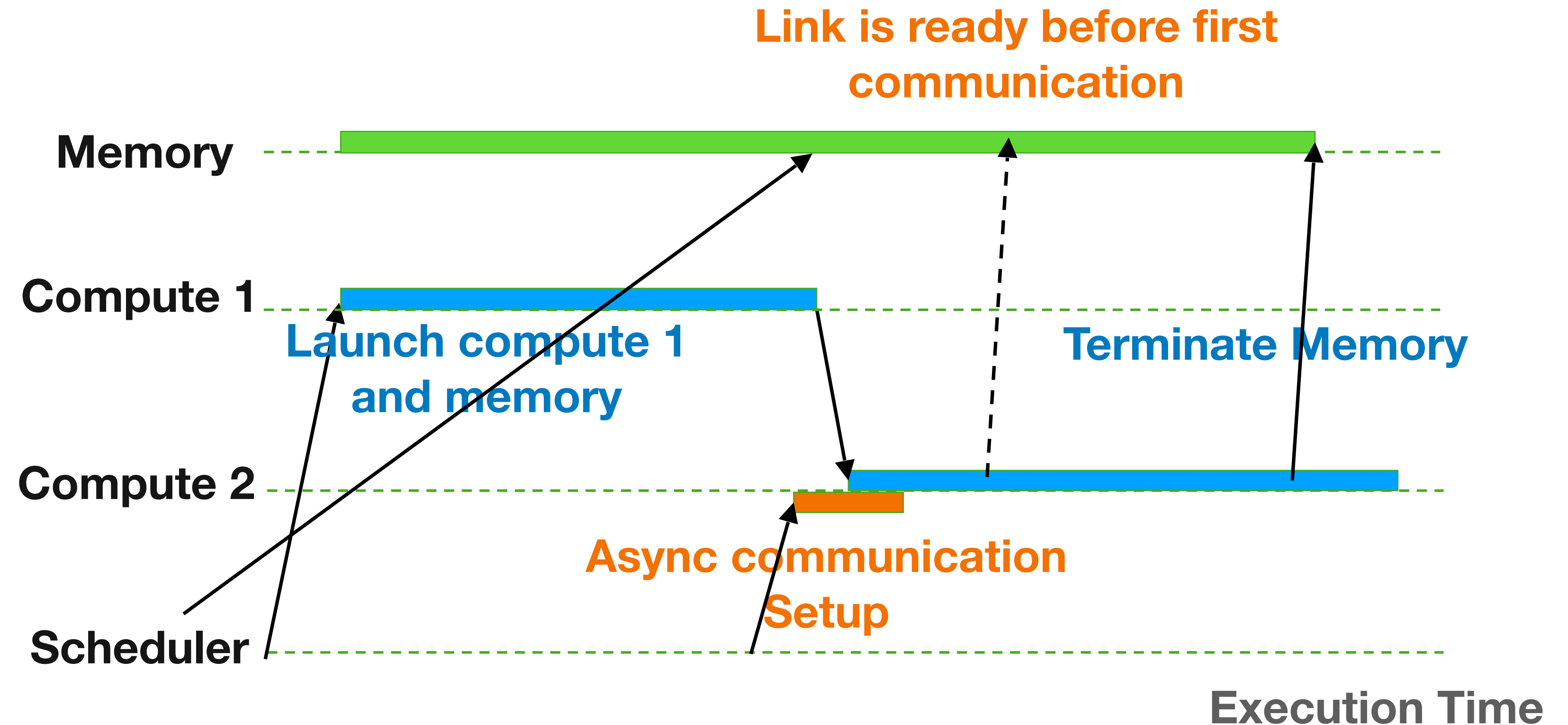
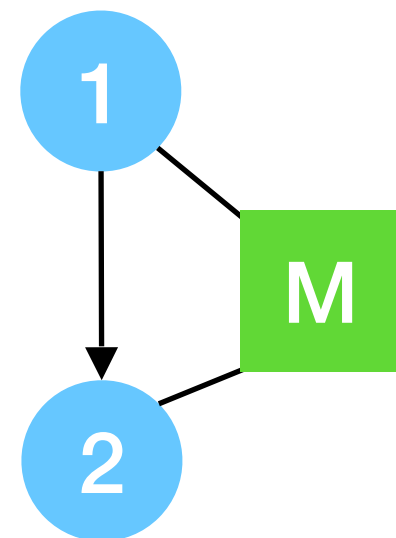
Two-Level Scheduling Mechanism



Decompose scheduling tasks into two levels solves **scheduling scalability problem**

Async Communication Mechanism

Communicating components in graph



Utilize **pre-setup scheduler links** to build connections between auto-scaled components: sub-100 μ s setup time, and largely hidden

Memory Accesses

- Native accesses when compute and data components scheduled locally
- Remote accesses via Zenix API calls (possibly to multiple servers)
- Local accesses via transparent swapping (when compute component scales beyond allocated size)
- Native/swap or API calls chosen at scheduling time and use different compilations

Implementation

- Compiler implemented on Mira (1.4K LOC)
- Scheduling/Monitoring built on top of OpenWhisk with 9K Scala LOC
- Two proof-of-concept add-on compiler: annotation-based and library based
- TCP/RDMA-enabled container runtime library with 6.4K C++/Python LOC.

Evaluation Results - ML Training

- Logistic regression with two inputs
- Comparing with AWS Lambda, Lambda+Step Function with Orion resource tuning and with Redis storing intermediate results, OpenWhisk, and FastSwap
- Zenix reduces resource consumption by 40% to 85%

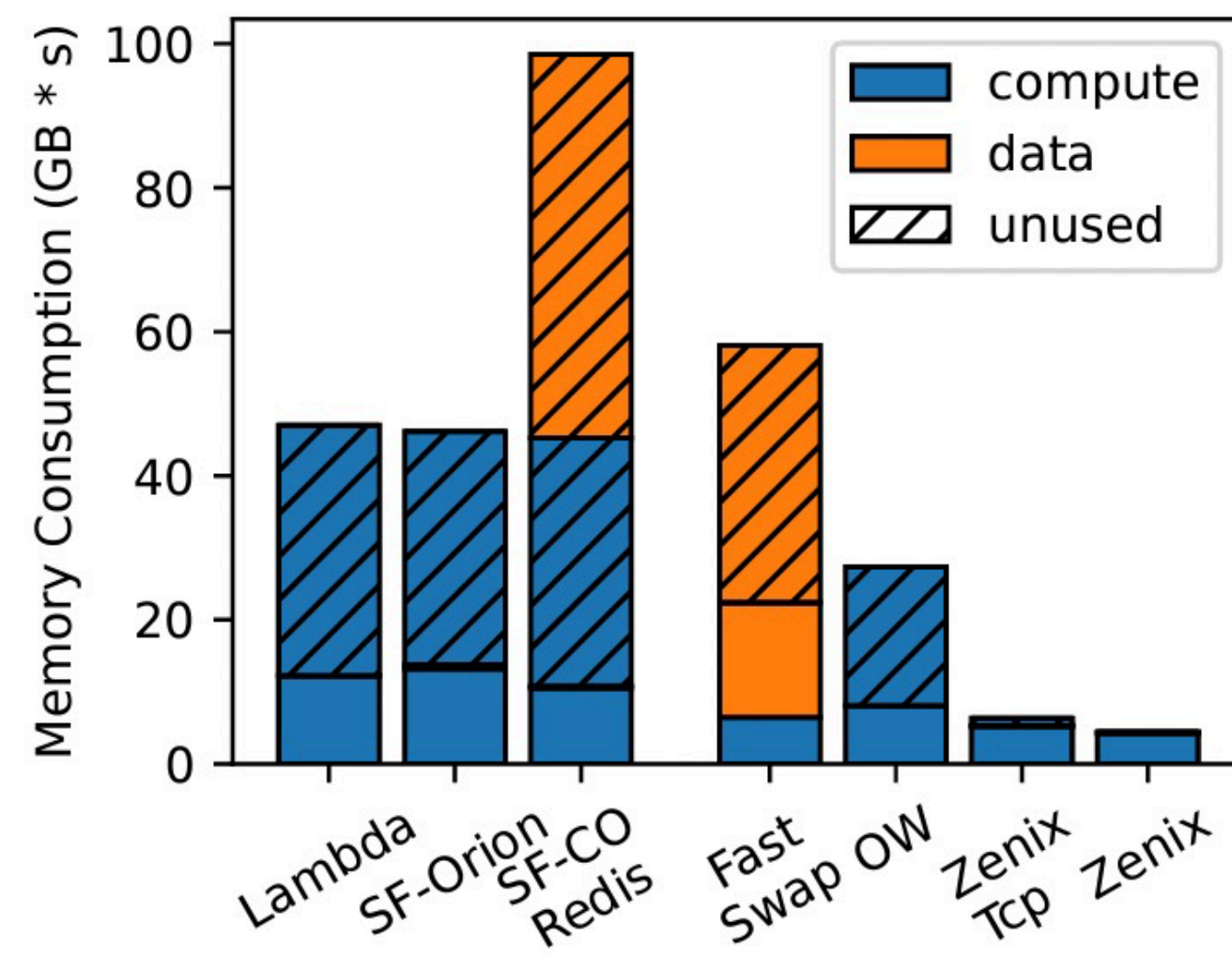


Figure 8: LR Memory Consumption with Small Input.

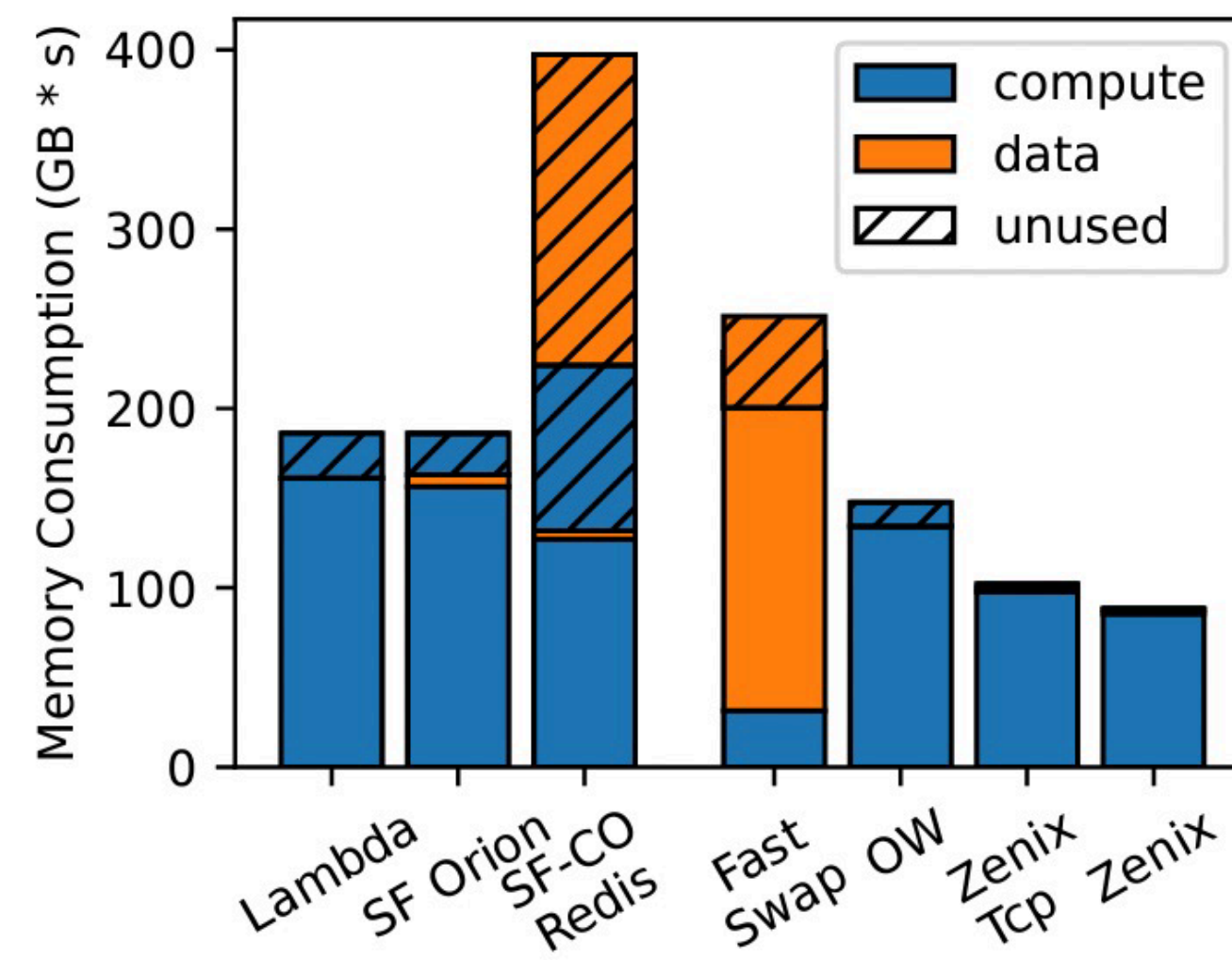


Figure 9: LR Memory Consumption with Large Input.

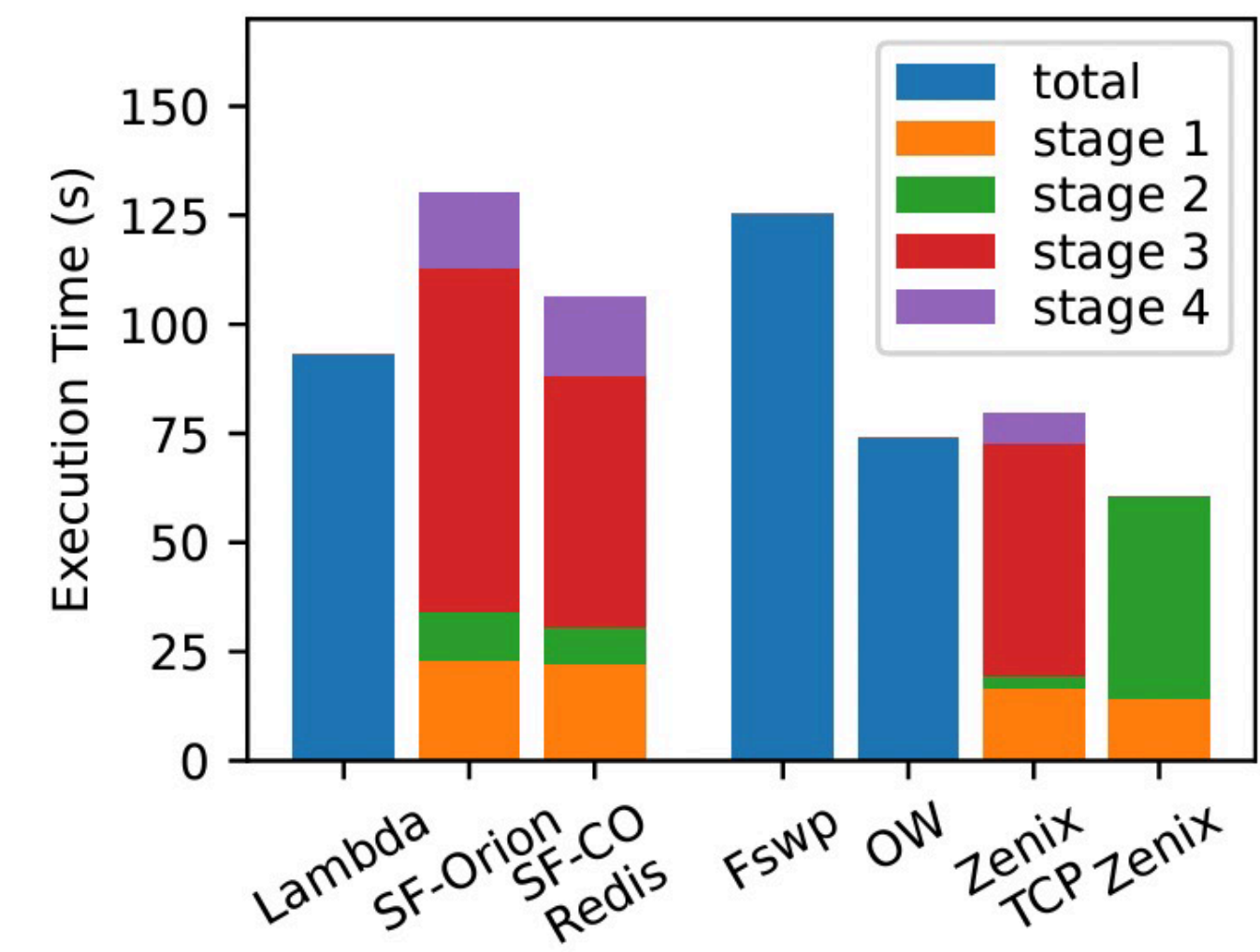


Figure 10: LR Execution Time Breakdown with Large Input.

Evaluation Results - Video Processing

- Video transcoding (ported from ExCamera) with three video sizes
- Compared to gg (serverless for video), vpxenc (running on a single server)

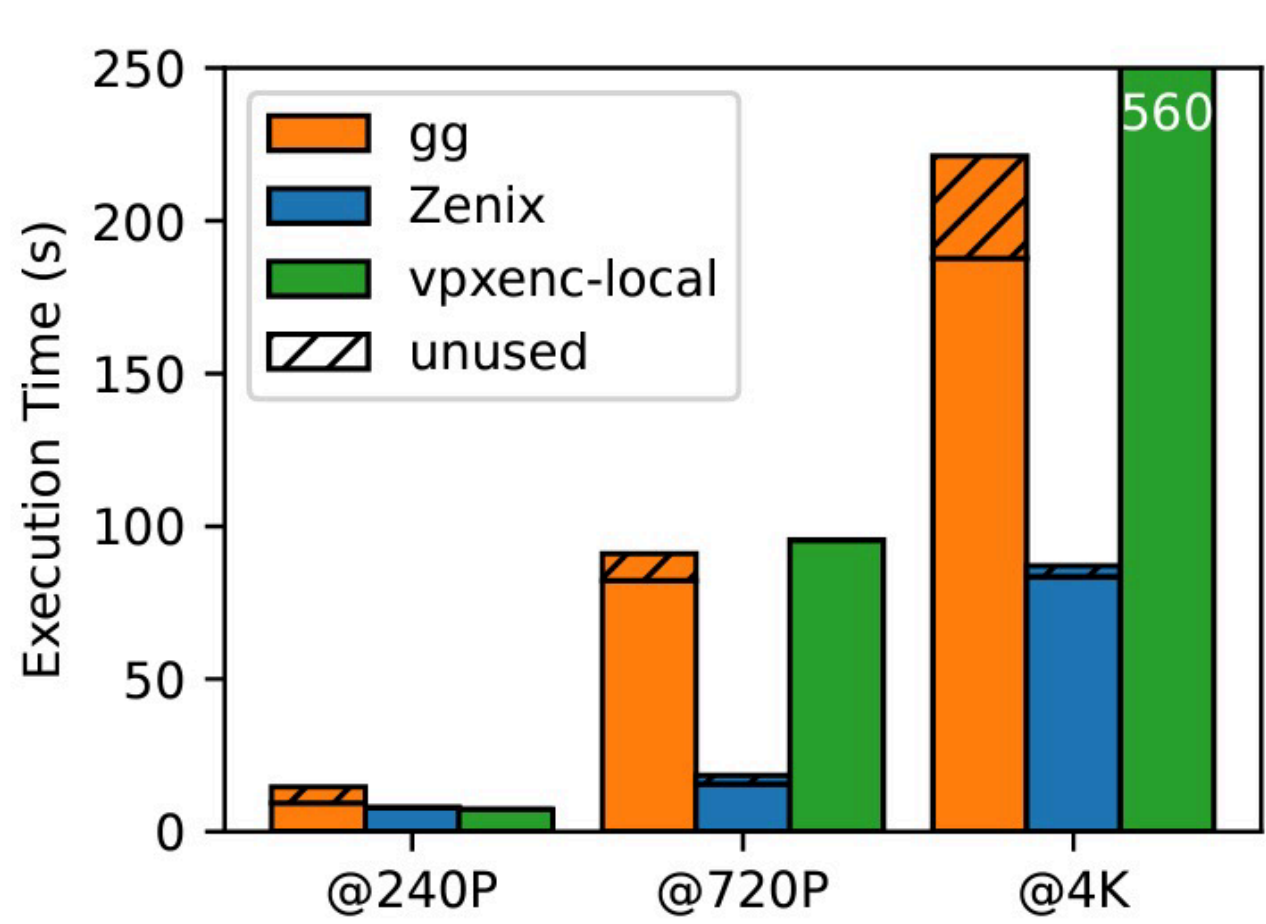


Figure 13: Video Transcoding Execution Time. With 3 video resolutions.

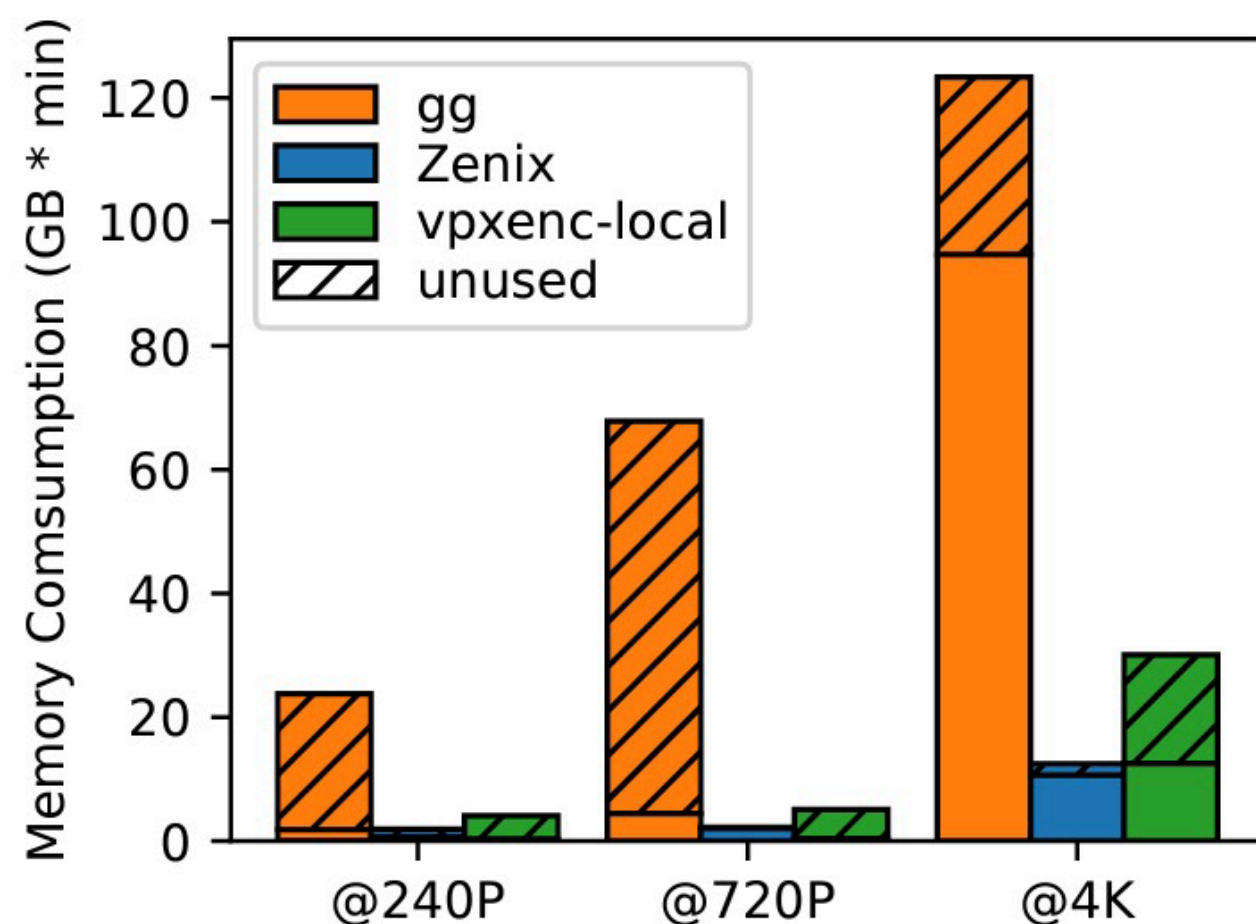


Figure 14: Video Transcoding Memory Consumption.

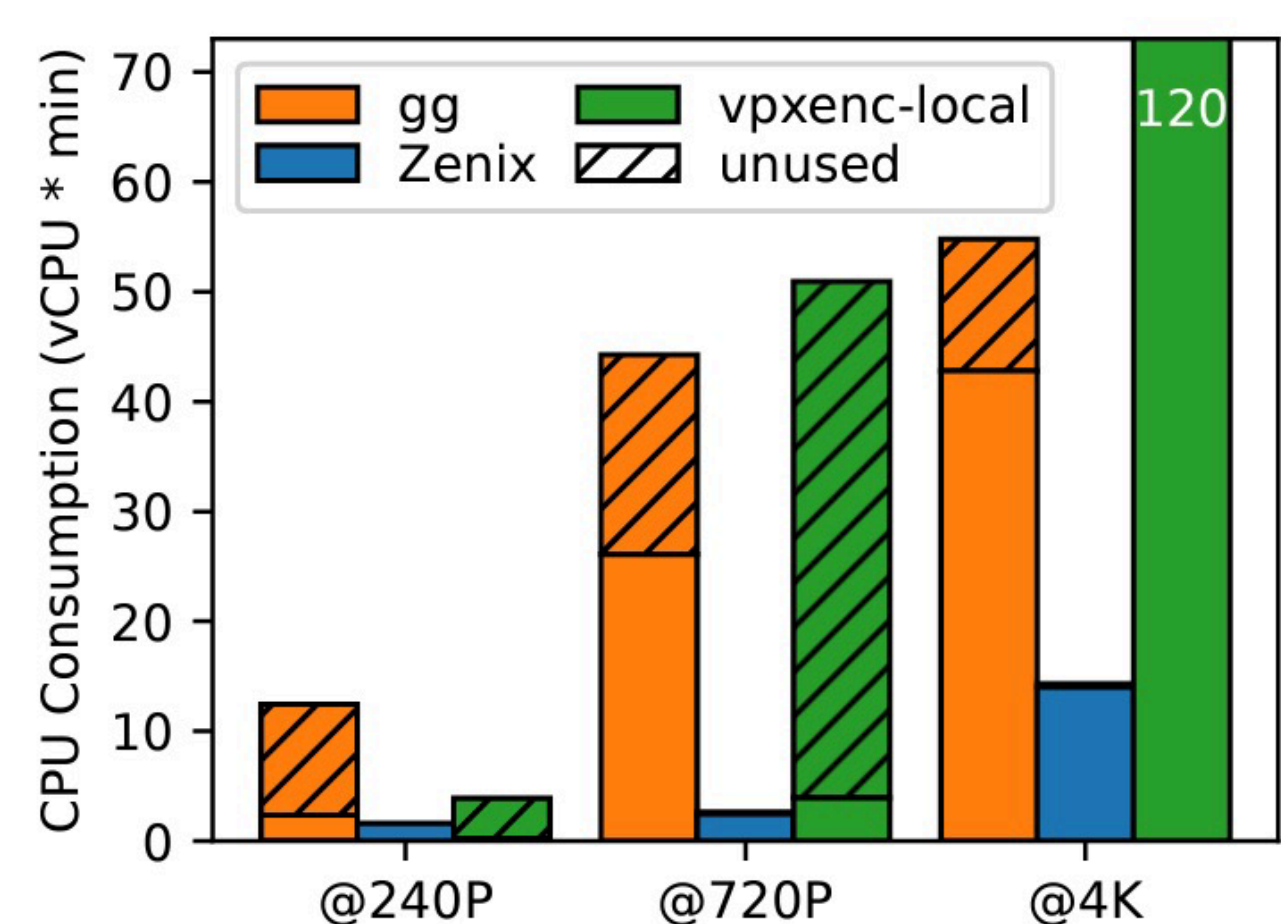
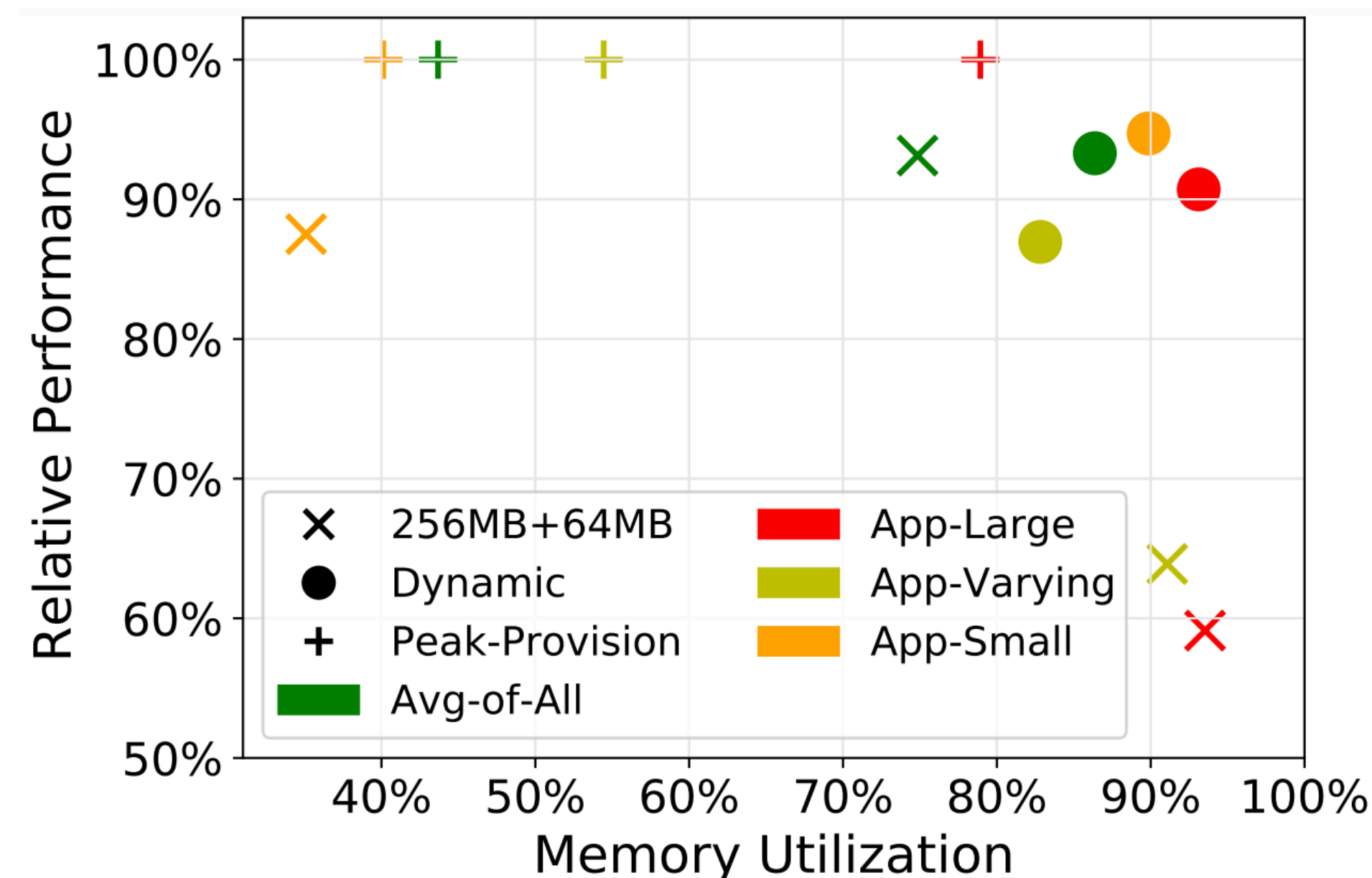


Figure 15: Video Transcoding CPU Consumption.

Evaluation Results - Resource Materialization Policies

- Zenix monitor system could adjust allocation policy at resource granularity to reduce the performance.
- For both small, single element applications and large multistage operations, Scad could achieve over 90% resource utilization and performance.



Summary of Zenix

Serverless computing can and benefit from being resource centric!

We built **Zenix**, a resource-centric serverless computing platform
with **resource graphs** executed with **disaggregation and materialization**
achieves high performance, scalability, and resource utilization

Check our paper for more details!
<https://arxiv.org/abs/2206.13444>