Making Serverless Pay-For-Use a Reality with Leopard

Tingjia Cao,
Andrea C. Arpaci-Dusseau,
Remzi H. Arpaci-Dusseau, and
Tyler Caraza-Harter

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Making Serverless Pay-For-Use a Reality with Leopard

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Tingjia Cao,
Andrea C. Arpaci-Dusseau,
Remzi H. Arpaci-Dusseau, and
Tyler Caraza-Harter

Serverless Computing (FaaS): Popularity and Benefits

📈 Serverless is popular, AWS Lambda is used in 65% architectures[1]

🧑‍💻 Relieve cloud users from managing servers

💰 Fine-grained, pay-as-you-go billing

1. Cloudscape: A Study of Storage Services in Modern Cloud Architectures, Fast 25, Sambhav Satija et.al, University of Wisconsin Madison

Pay-for-use Billing Model

Providers advertise "pay-for-use" model

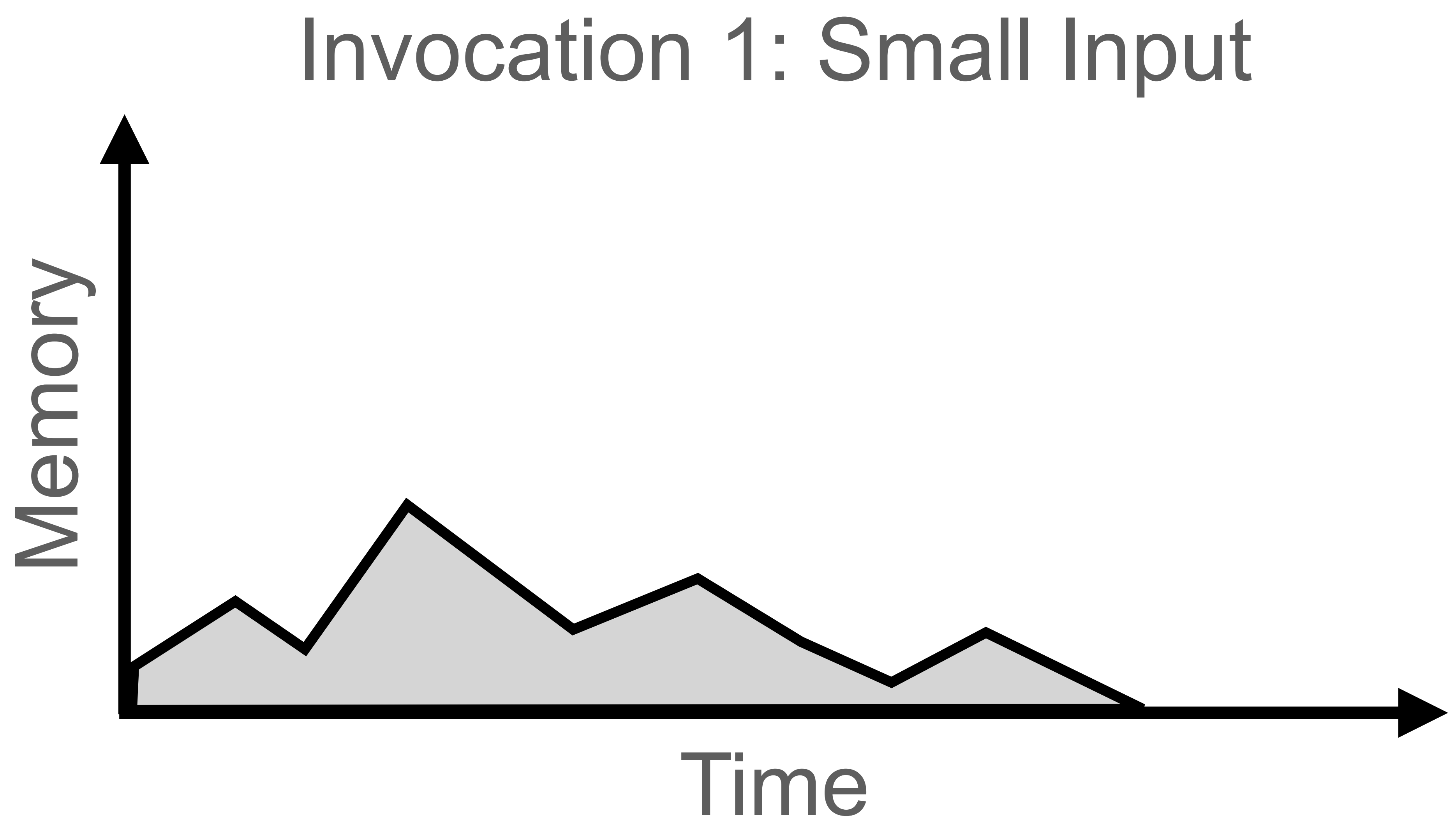
- Azure: "Pay-per-use"
- GCP, AWS Lambda: "Pay only for what you use"

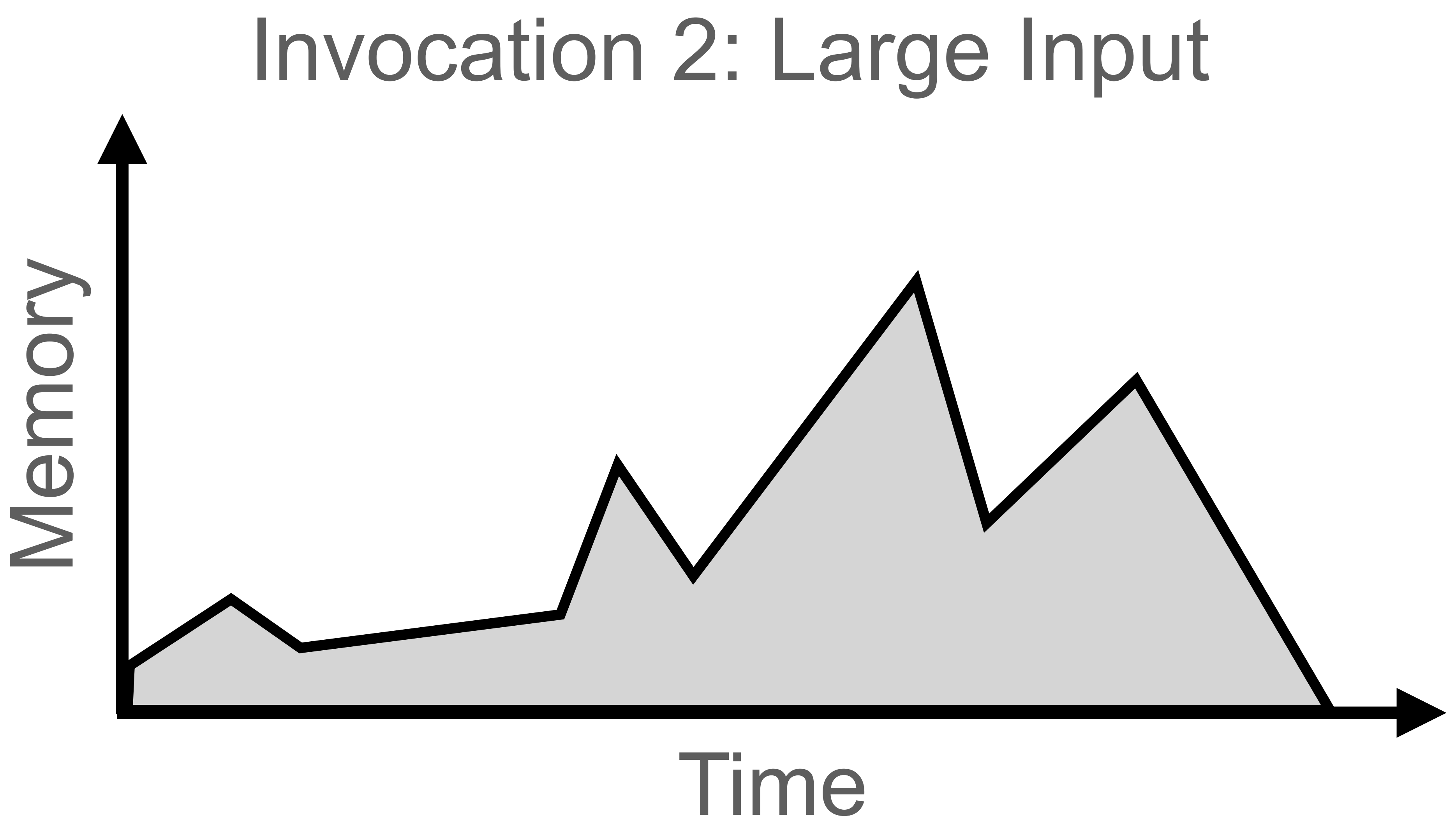
What does pay-for-use actually mean ?

What Does “Pay-for-Use” Actually Mean?

💡 Intuitive definition

- You pay proportionally to area under the curves

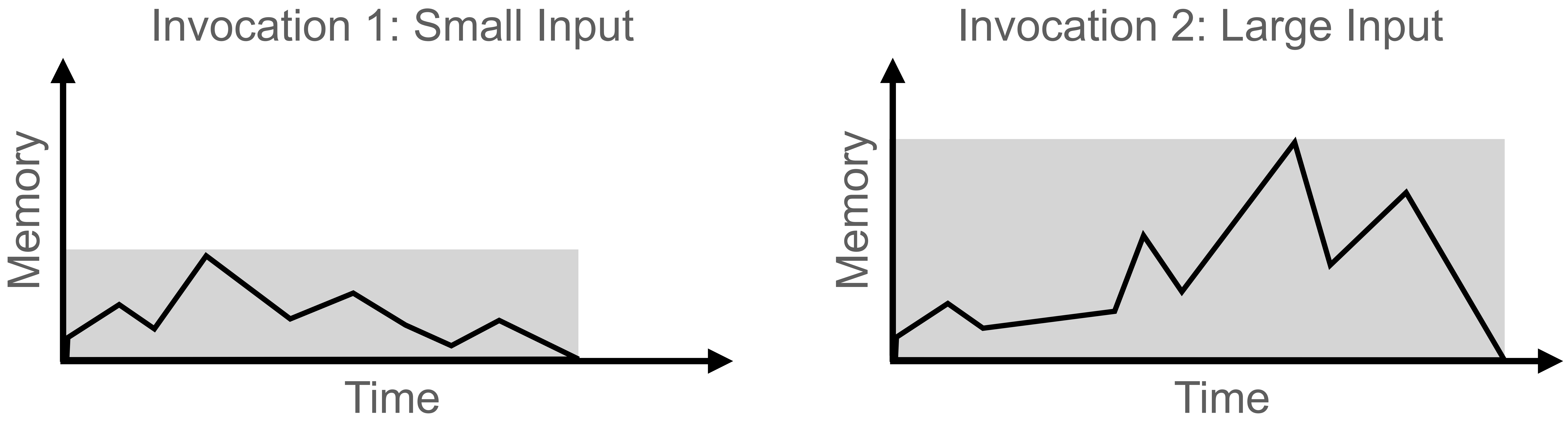




What Does “Pay-for-Use” Actually Mean?

🔧 In practice: you choose a memory limit

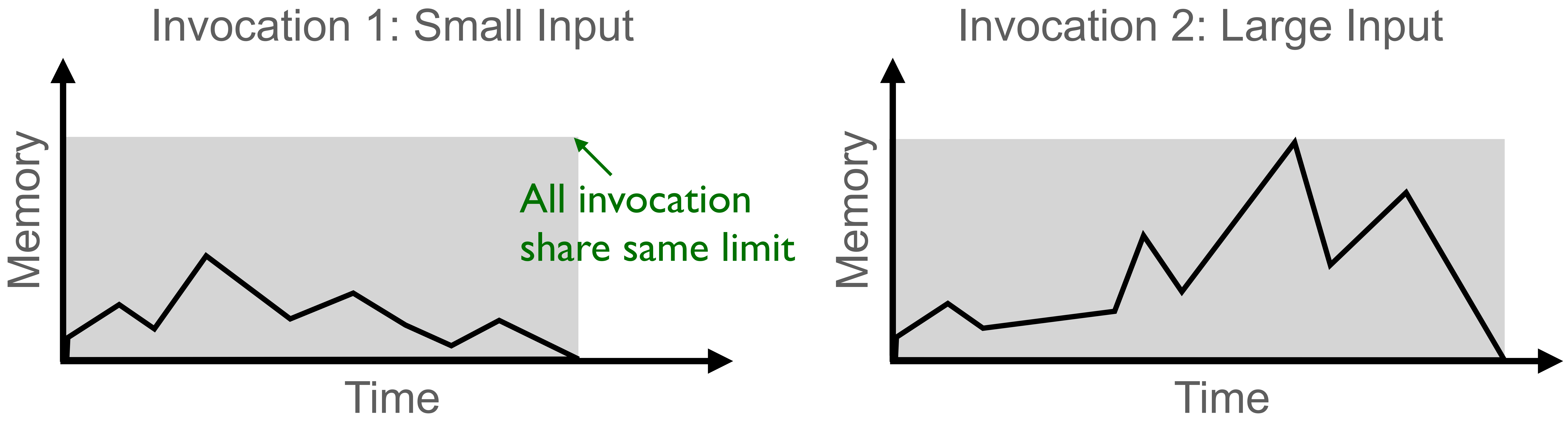
- Pay for execution time × memory limit (hopefully set to max usage)



What Does “Pay-for-Use” Actually Mean?

🔧 In practice: you choose a memory limit

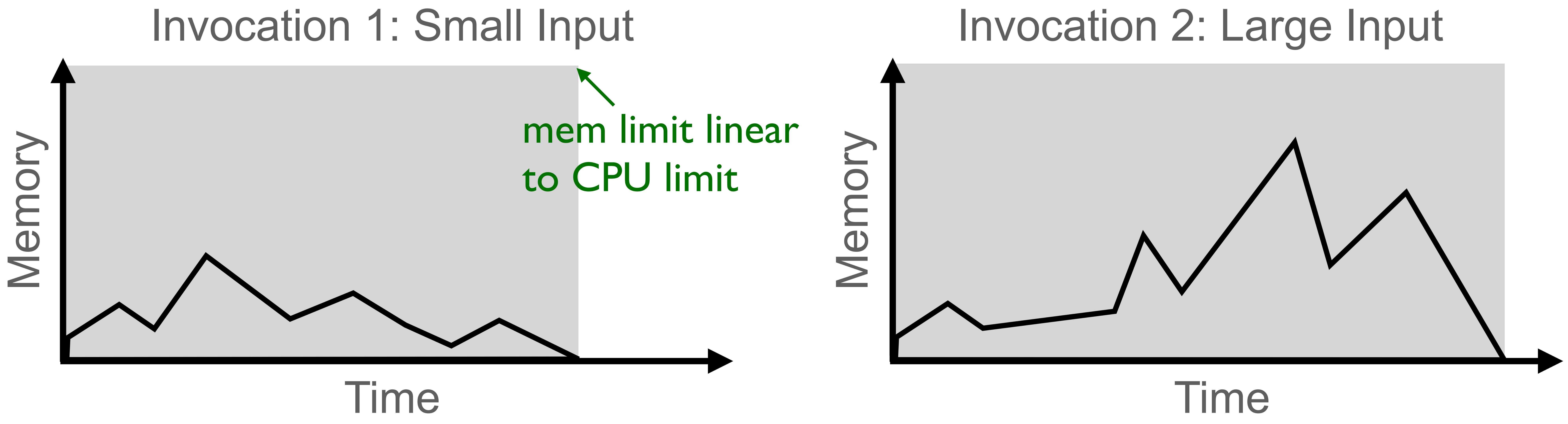
- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit



What Does “Pay-for-Use” Actually Mean?

🔧 In practice: you choose a memory limit

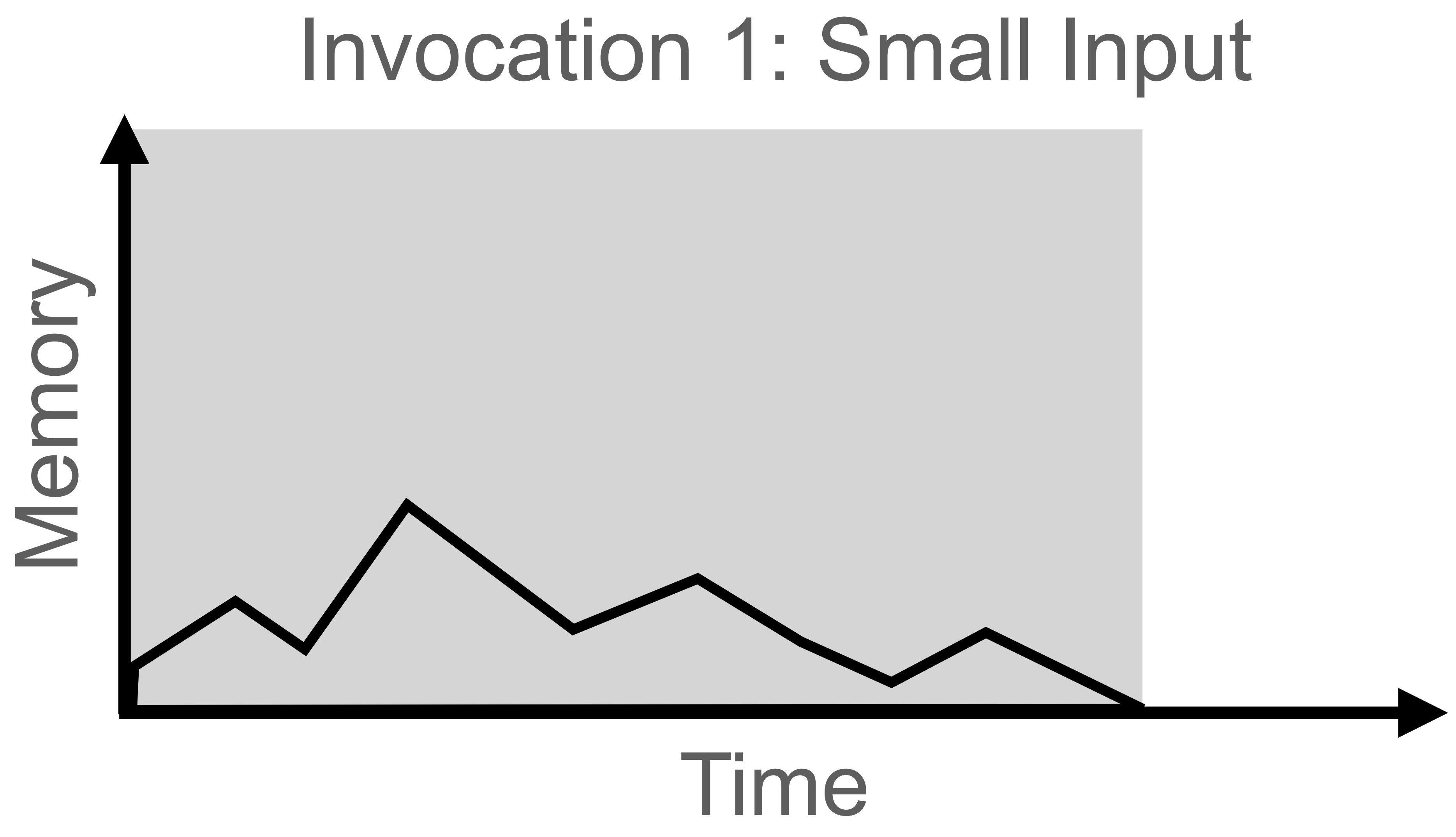
- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit
- Memory limit is linear with CPU reservation

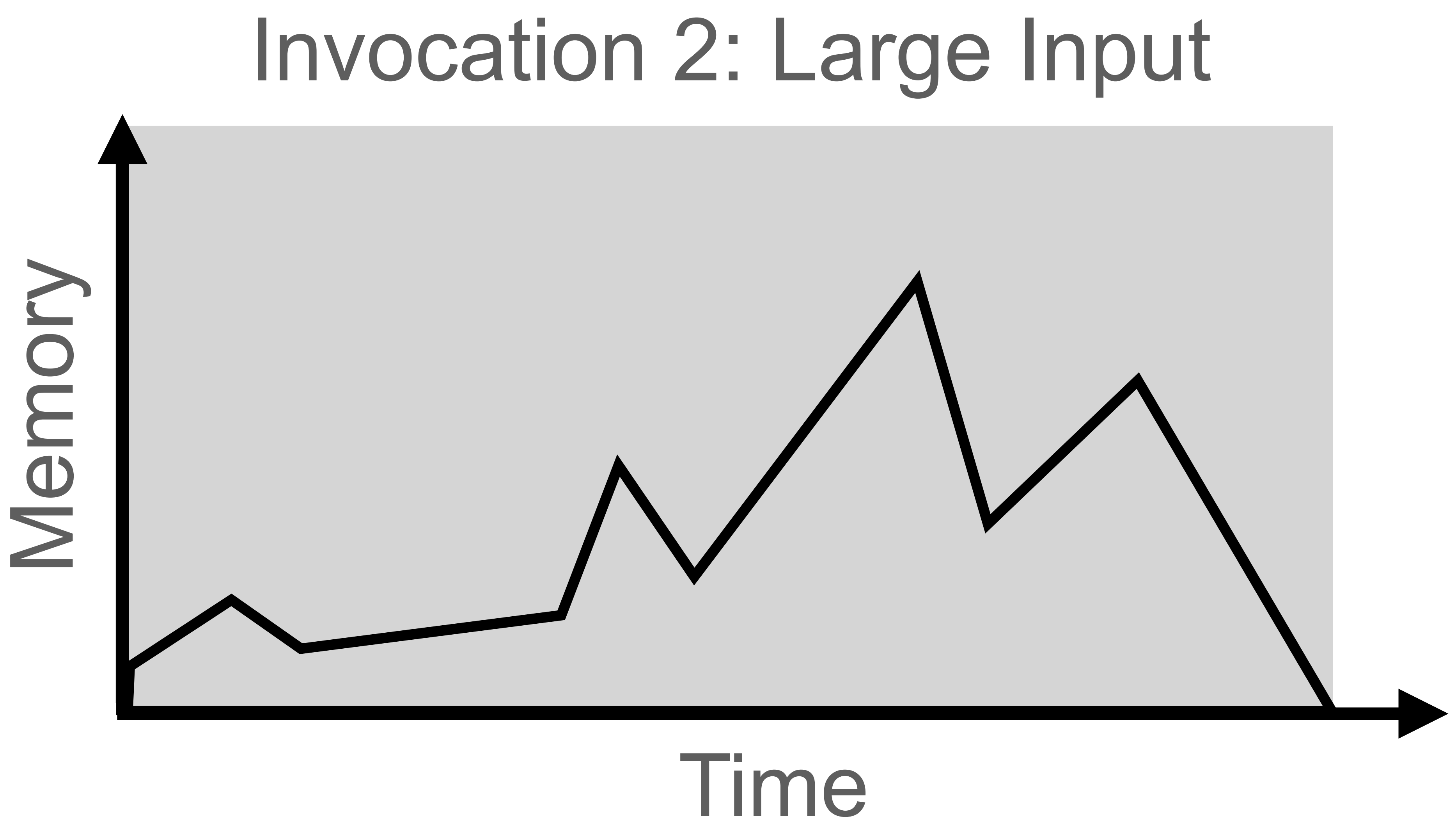


What Does "Pay-for-Use" Actually Mean?

🔧 In practice: you choose a memory limit

- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit
- Memory limit is linear with CPU reservation
- No discount for usage during low-demand time

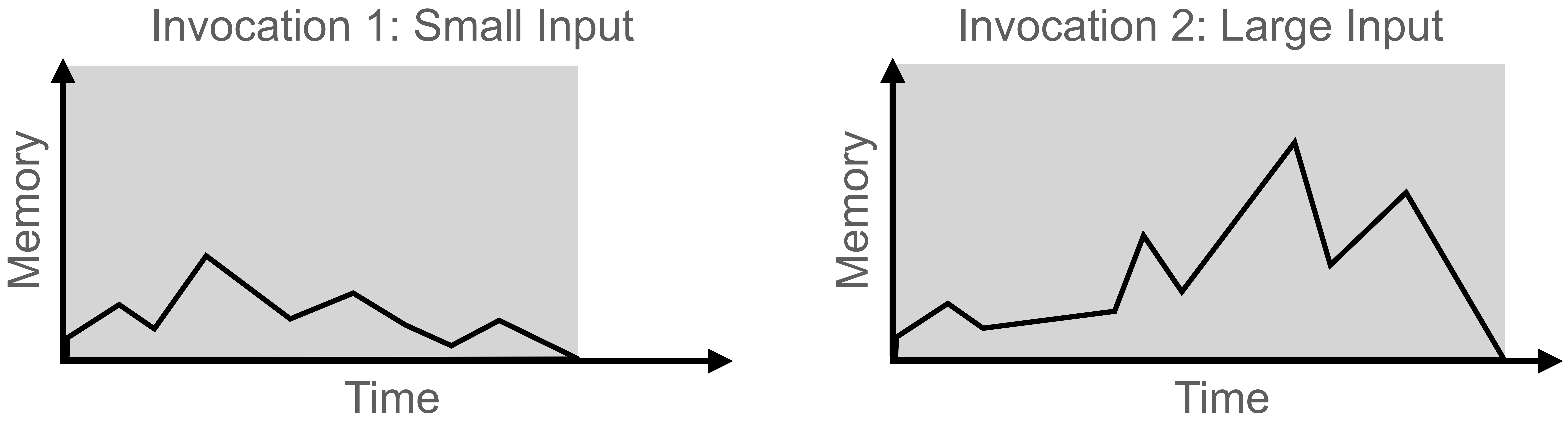




What Does "Pay-for-Use" Actually Mean?

🔧 In practice: you choose a memory limit

- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit
- Memory limit is linear with CPU reservation
- No discount for usage during low-demand time



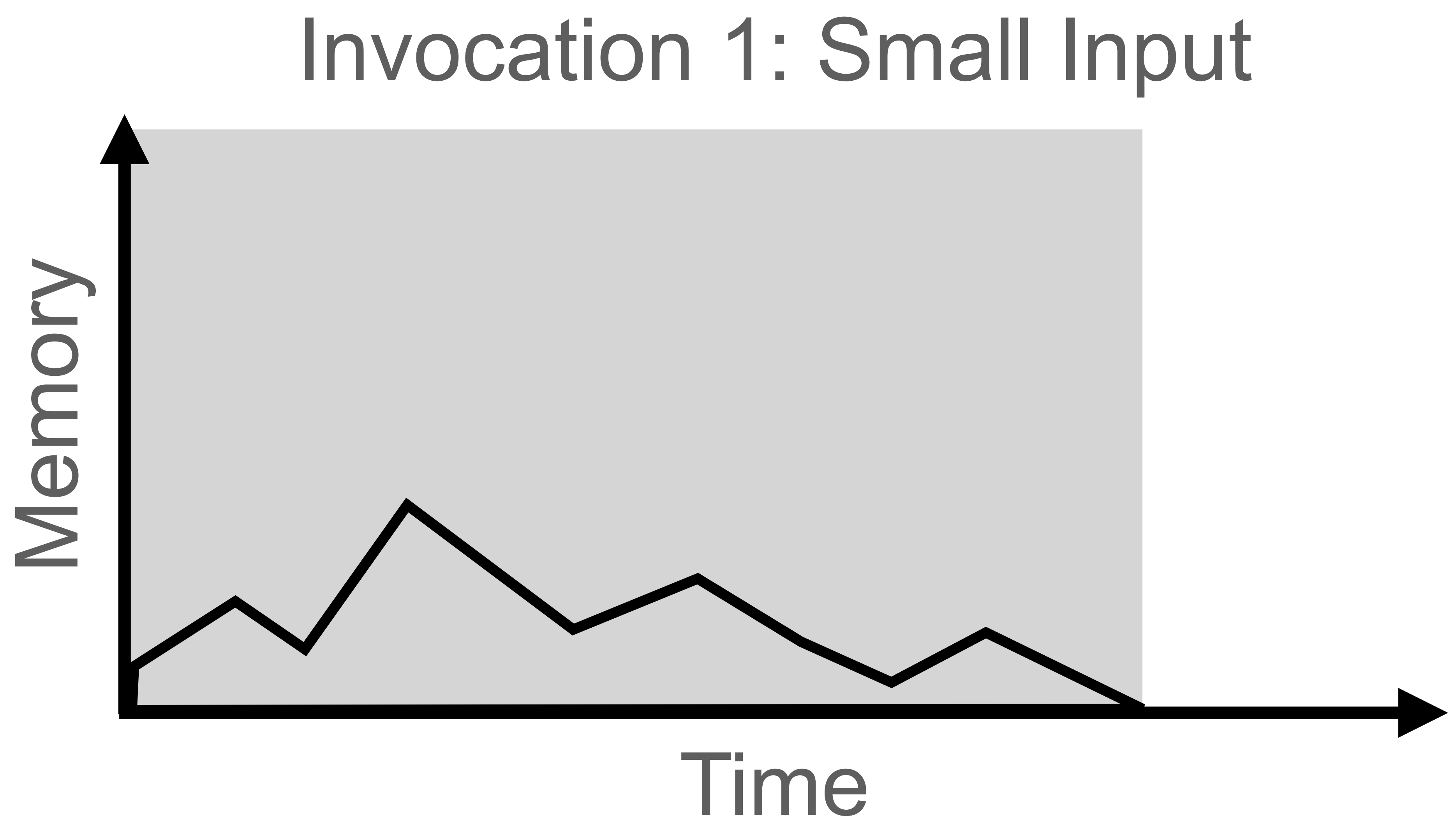
What Does “Pay-for-Use” Actually Mean?

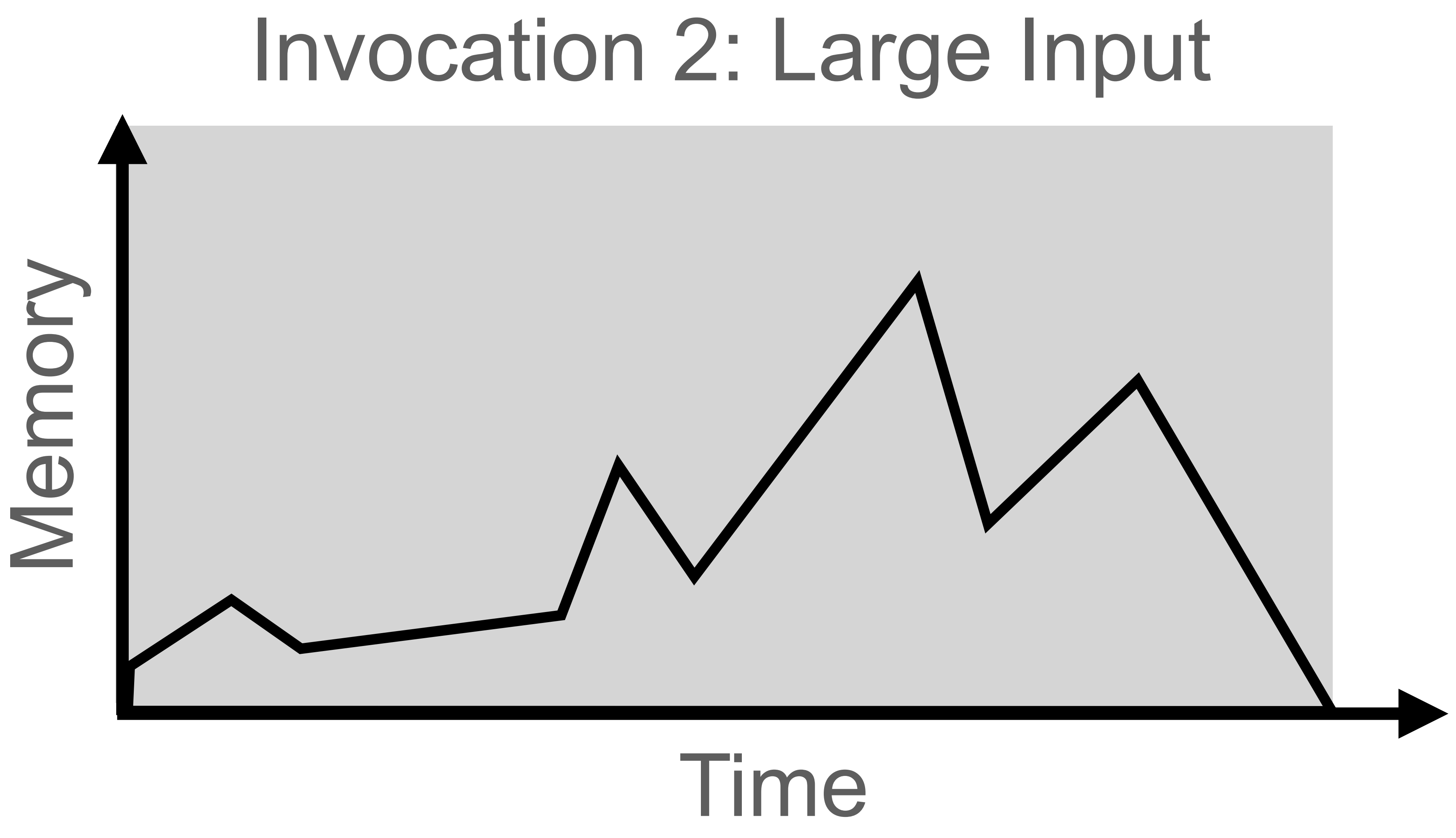
🔧 In practice: you choose a memory limit

- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit

Static

- Memory limit is linear with CPU reservation
- No discount for usage during low-demand time





What Does "Pay-for-Use" Actually Mean?

🔧 In practice: you choose a memory limit

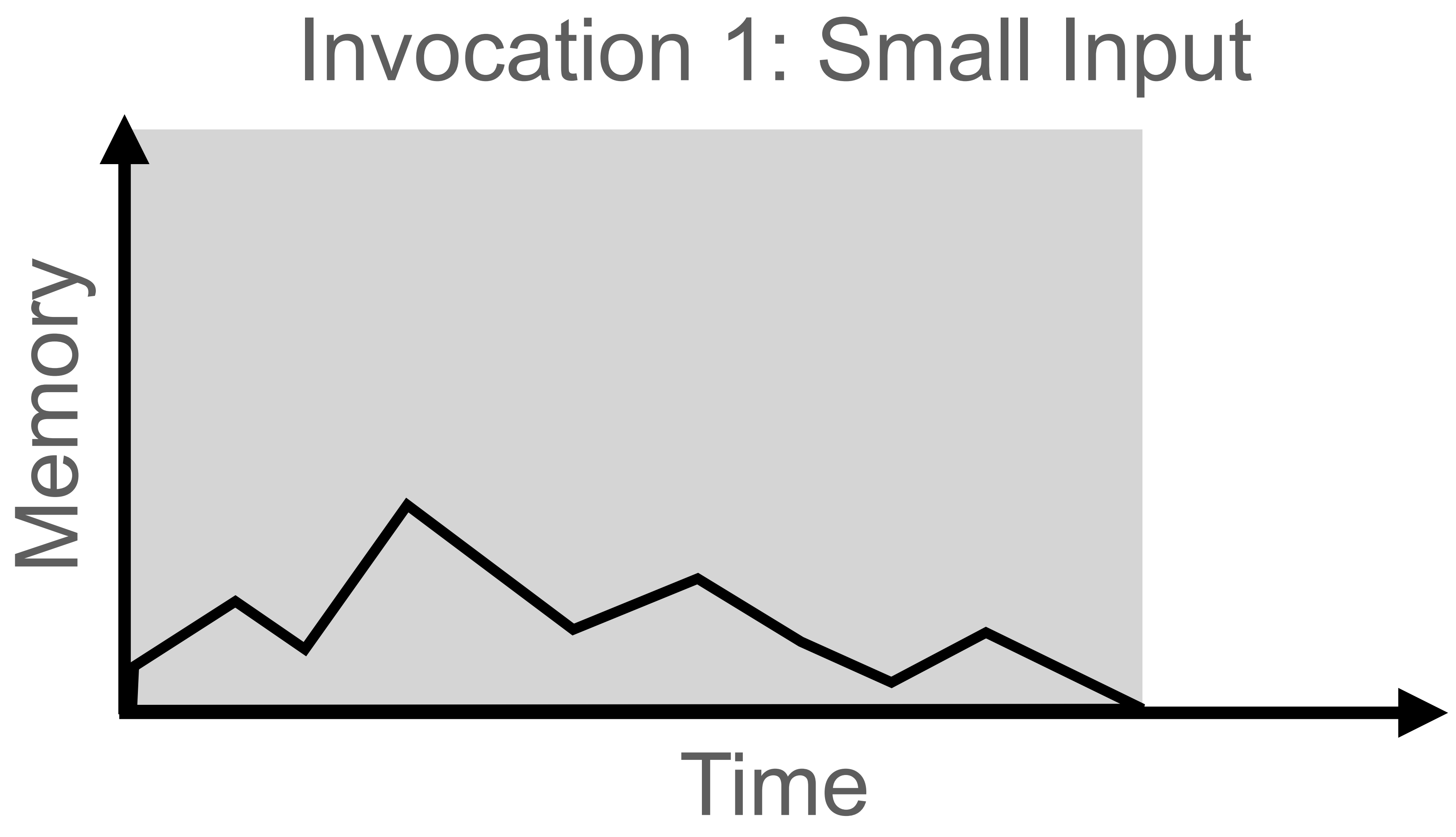
- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit

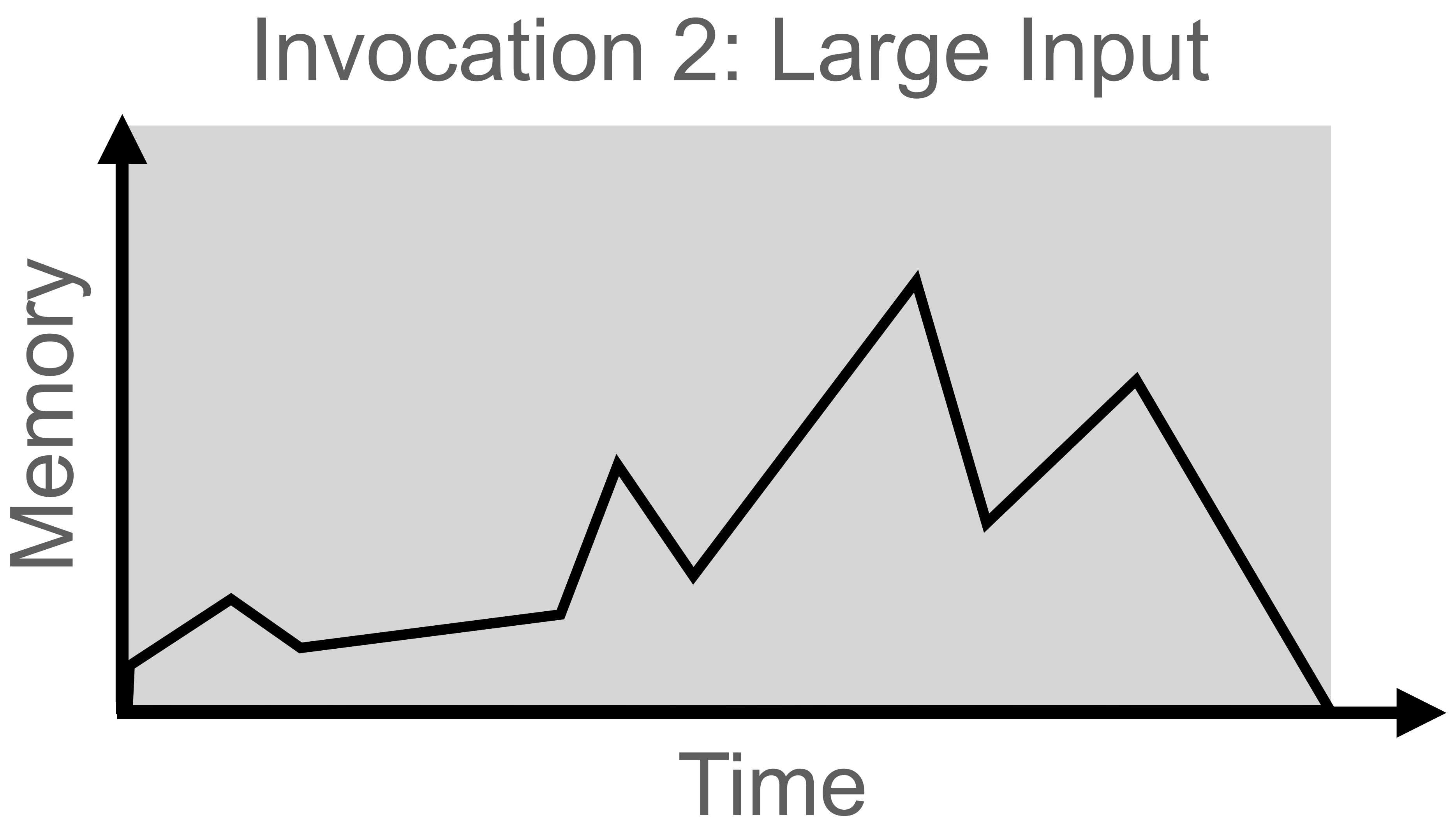
Static

- Memory limit is linear with CPU reservation

Linear

- No discount for usage during low-demand time





What Does “Pay-for-Use” Actually Mean?

🔧 In practice: you choose a memory limit

- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit

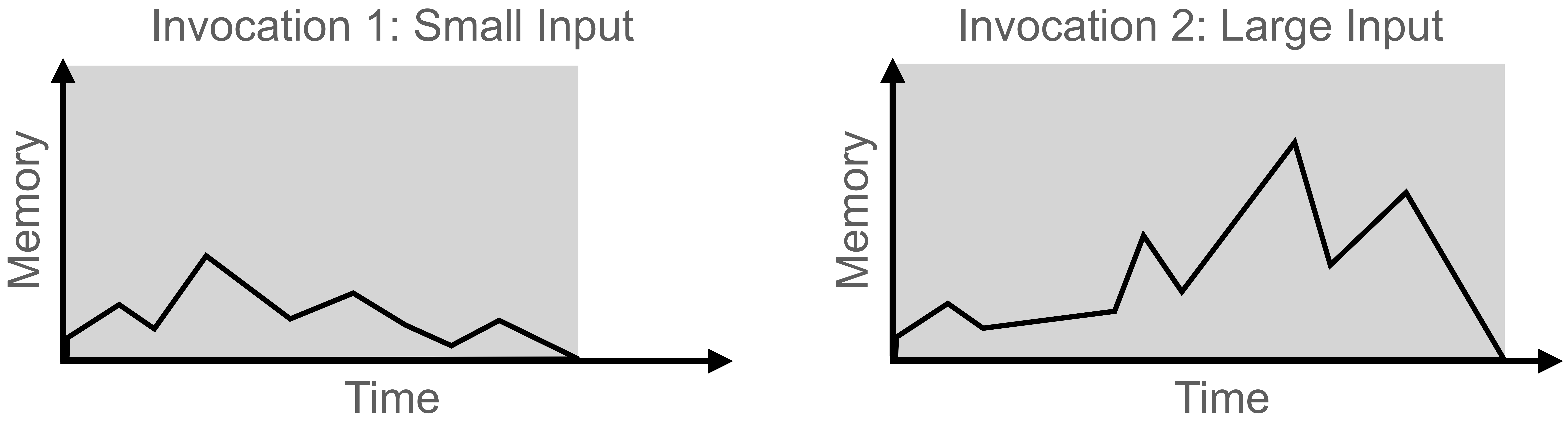
Static

- Memory limit is linear with CPU reservation

Linear

- No discount for usage during low-demand time

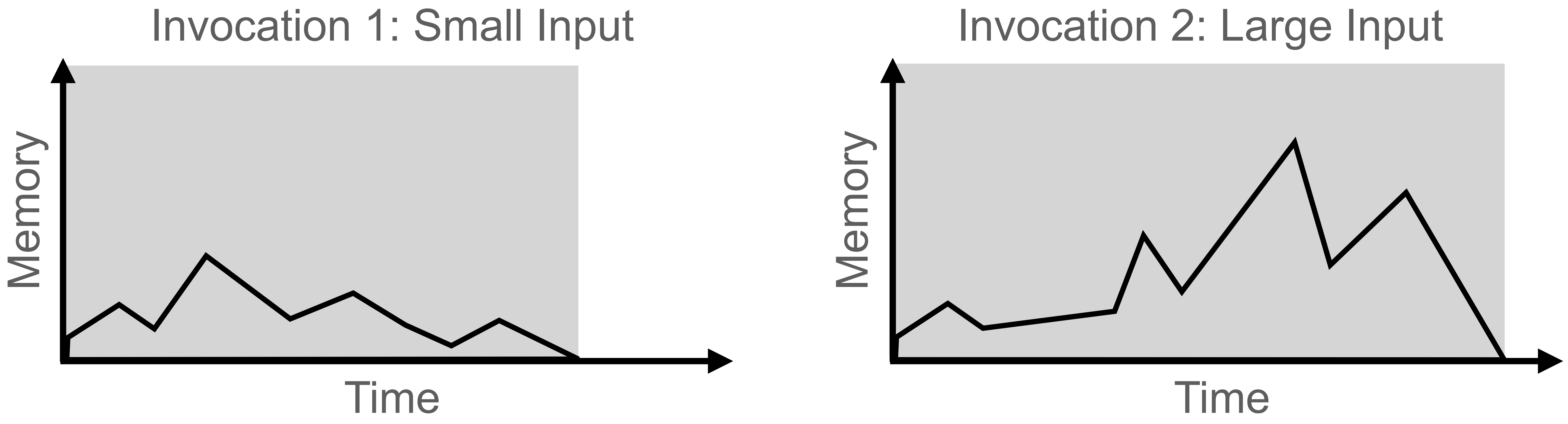
Interactive-only



What Does “Pay-for-Use” Actually Mean?

🔧 In practice: you choose a memory limit Static Linear Interactive-only Model (SLIM)

- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit
- Memory limit is linear with CPU reservation
- No discount for usage during low-demand time



What Does “Pay-for-Use” Actually Mean?

🔧 In practice: you choose a memory limit

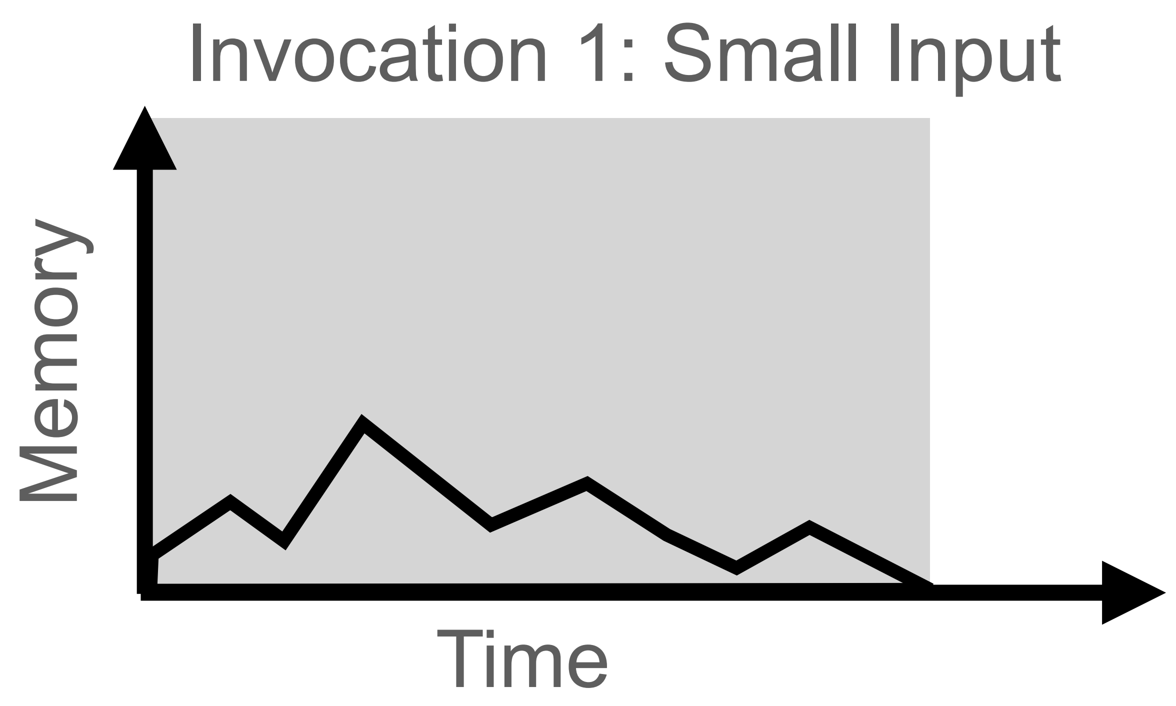
- Pay for execution time × memory limit (hopefully set to max usage)
- All invocations share same limit
- Memory limit is linear with CPU reservation
- No discount for usage during low-demand time

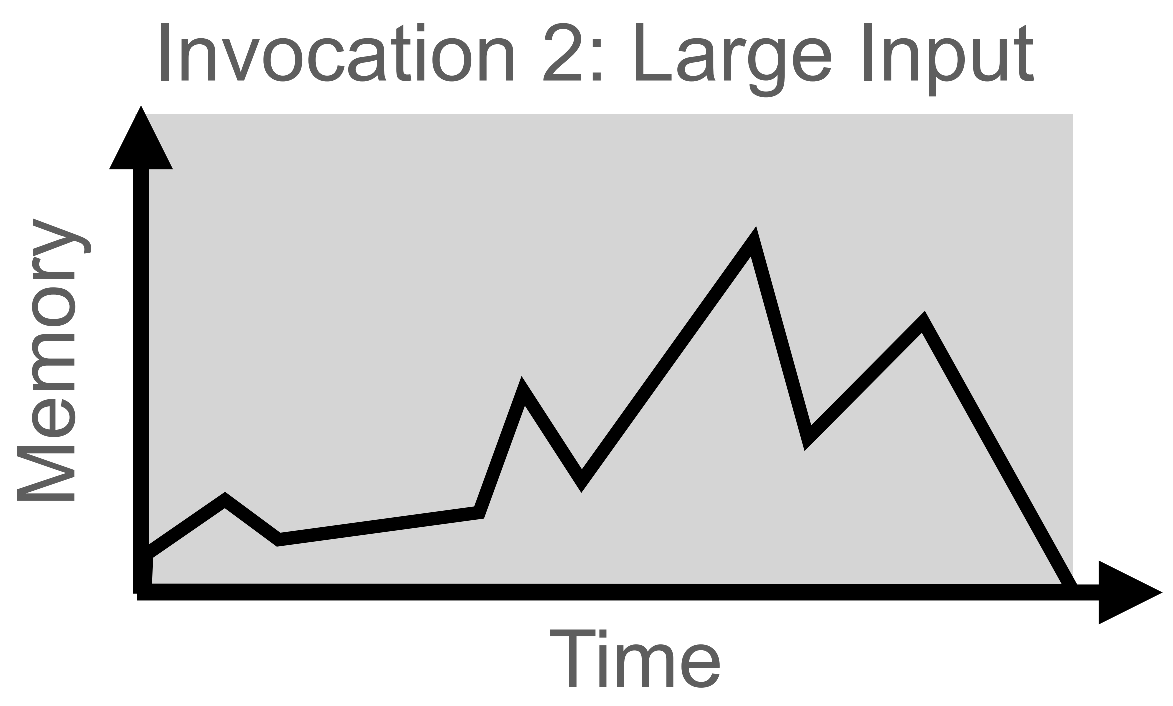
👤 Customer side:

- ✅ Simple ❌ Not true pay-for-use

🏢 Provider side:

- ✅ Profitable





Contribution: Better Billing Model and FaaS System to Support it

New model: Nearly-PFU

- Benefits both providers and customers

New system: Leopard

- 🐧 Linux techs: new cgroup APIs, modified CFS scheduler, customizable OOM killer
- FaaS techs: improved admission controller, load balancer and sandbox evictor

Evaluation highlights

- 🚀 Provider throughput ↑ 2.3×
- 💰 Customer cost ↓ 34% (interactive), ↓ 59% (batch)

Outline

Introduction

Nearly Pay-for-Use model

Leopard FaaS system

Evaluation highlights

Goals to Build Better Serverless Billing Model

⚙️ Appropriate number of knobs

💰 Billing function

- Closely approximates ideal pay-for-use
- Maintains provider profitability

Intuitions to Build Better Serverless Billing Model

💡 Break the limitations of *static, linear interactive model (SLIM)*

- Not linear

  ⇒ Decouple CPU and memory knobs

- Not interactive-only

  ⇒ Allow users to set urgency levels per resource subset

- Not static

  ⇒ Allow users to **lend** idle-but-reserved resources to others for non-urgent needs

CPU Knobs in Nearly-PFU

CPU-cap:

- Maximum number of CPUs a function is allowed to use

Spot-CPU:

- Subset of CPU-cap that a function does **not** need immediately

CPU-cap – spot-CPU = reserved-CPUs:

- CPUs that a function need full, immediate access to when needed

CPU Knobs in Nearly-PFU

CPU-cap:

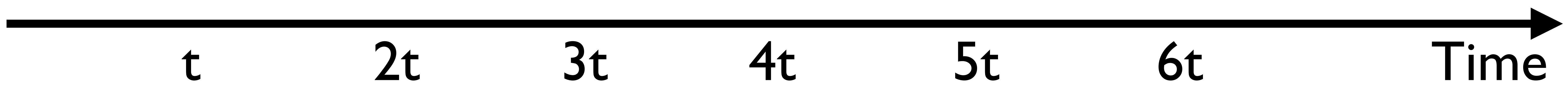
- Maximum number of CPUs a function is allowed to use

Spot-CPU:

- Subset of CPU-cap that a function does **not** need immediately

CPU-cap – spot-CPU = reserved-CPUs:

- CPUs that a function need full, immediate access to when needed



CPU Knobs in Nearly-PFU

CPU-cap:

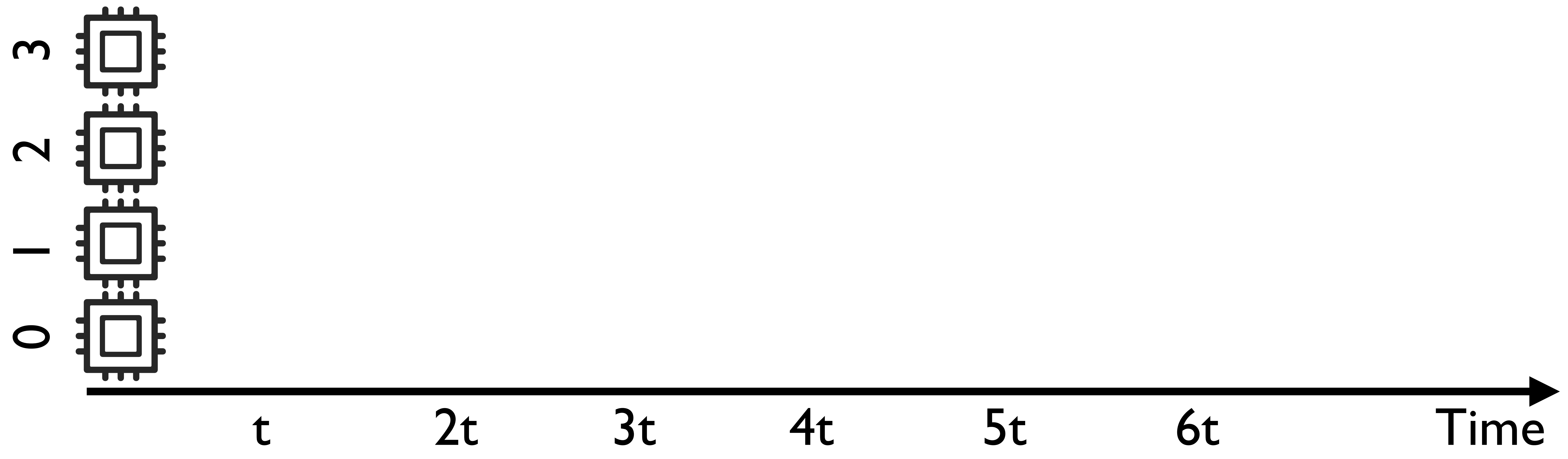
- Maximum number of CPUs a function is allowed to use

Spot-CPU:

- Subset of CPU-cap that a function does **not** need immediately

CPU-cap – spot-CPU = reserved-CPUs:

- CPUs that a function need full, immediate access to when needed



CPU Knobs in Nearly-PFU

CPU-cap:

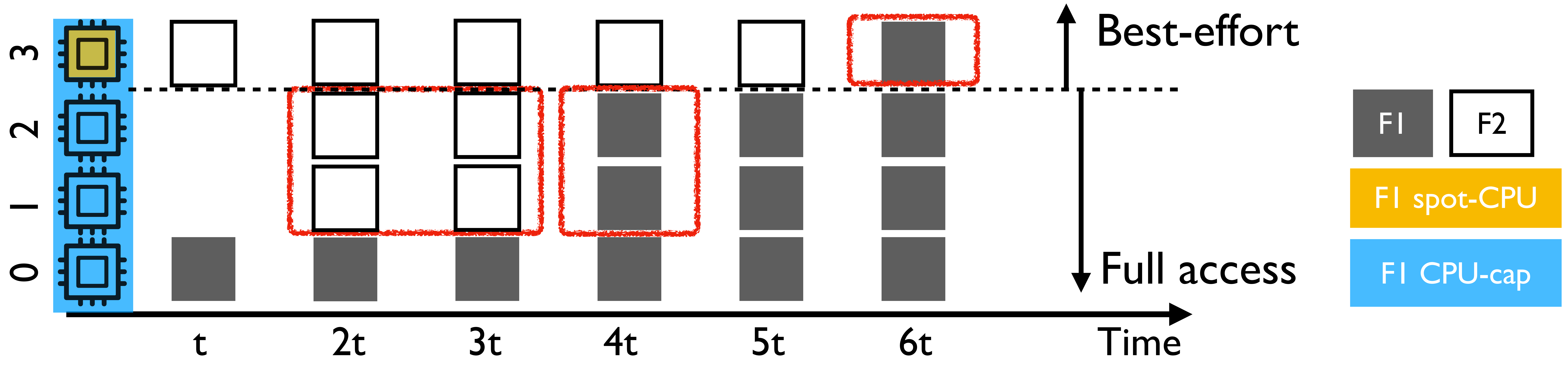
- Maximum number of CPUs a function is allowed to use

Spot-CPU:

- Subset of CPU-cap that a function does **not** need immediately

CPU-cap – spot-CPU = reserved-CPUs:

- CPUs that a function need full, immediate access to when needed



Memory Knobs in Nearly-PFU

CPU-cap:

- Maximum number of CPUs a function is allowed to use

Spot-CPU:

- Subset of CPU-cap that a function does **not** need immediately

Mem-cap:

- Maximum memory size a function is allowed to use

Preemptible-mem:

- Whether an instance can be preempted during execution

CPU Billing in Nearly-PFU

$$\text{Cost} = \text{Reserved-CPUtime} \times Cr + \text{Borrowed-CPUtime} \times Cs - \text{Lent-CPUtime} \times Cs$$

Base cost

Lower price for using spot-CPUs than reserved-CPUs

Give discounts when sharing your "allocated-but-idle" CPUs

Benefits of Nearly-PFU

✅ Closely approximate ideal pay-for-use

- No more static, linear interactive-only constraints

✅ Maintain provider profitability

- Lent resource discounts are paid by the borrower

Outline

Introduction

Nearly Pay-for-Use model

Leopard FaaS system

Evaluation highlights

🐆 “A leopard can’t change its spots”,

but our Leopard can!

Typical FaaS Implementation

FaaS platform implementation:

- Load balancer
  Routes invocations to physical nodes
- Admission controller
  Decides when to admit queued invocations

  Find or create a sandbox[1]
- Sandbox evictor
  Decides when to evict cached sandboxes

Linux kernel Implementation:

- cgroup APIs
  Enforces CPU and memory limits for function instances
- CFS scheduler
  Handles CPU time allocation and balances tasks across cores
- OOM Killer
  Terminates overcommitted processes when memory exceeds limits

Sandboxes to execute functions can be Docker, Firecracker, Kubernetes pods, OpenLambda's SOCK, etc.

Key Requirements to Support Nearly-PFU

Requirements for FaaS platform:

- Load balancer and admission controller: schedule non-linear, QoS aware instances
- Sandbox evictor: firstly kill preemptible instances during heavy memory

Requirements for the Linux:

- CPU reservation: full access on reserved-CPU and best-effort sharing on spot-CPUs
- Linux OOM killer: give control to the user-space sandbox evictor when OOM

Key Requirements to Support Nearly-PFU

Requirements for FaaS platform:

- Load balancer and admission controller: schedule non-linear, QoS aware instances
- Sandbox evictor: firstly kill preemptible instances during heavy memory

Requirements for the Linux:

- CPU reservation: full access on reserved-CPU and best-effort sharing on spot-CPUs
- Linux OOM killer: give control to the user-space sandbox evictor when OOM

See Leopard's solution for other requirements in the paper!

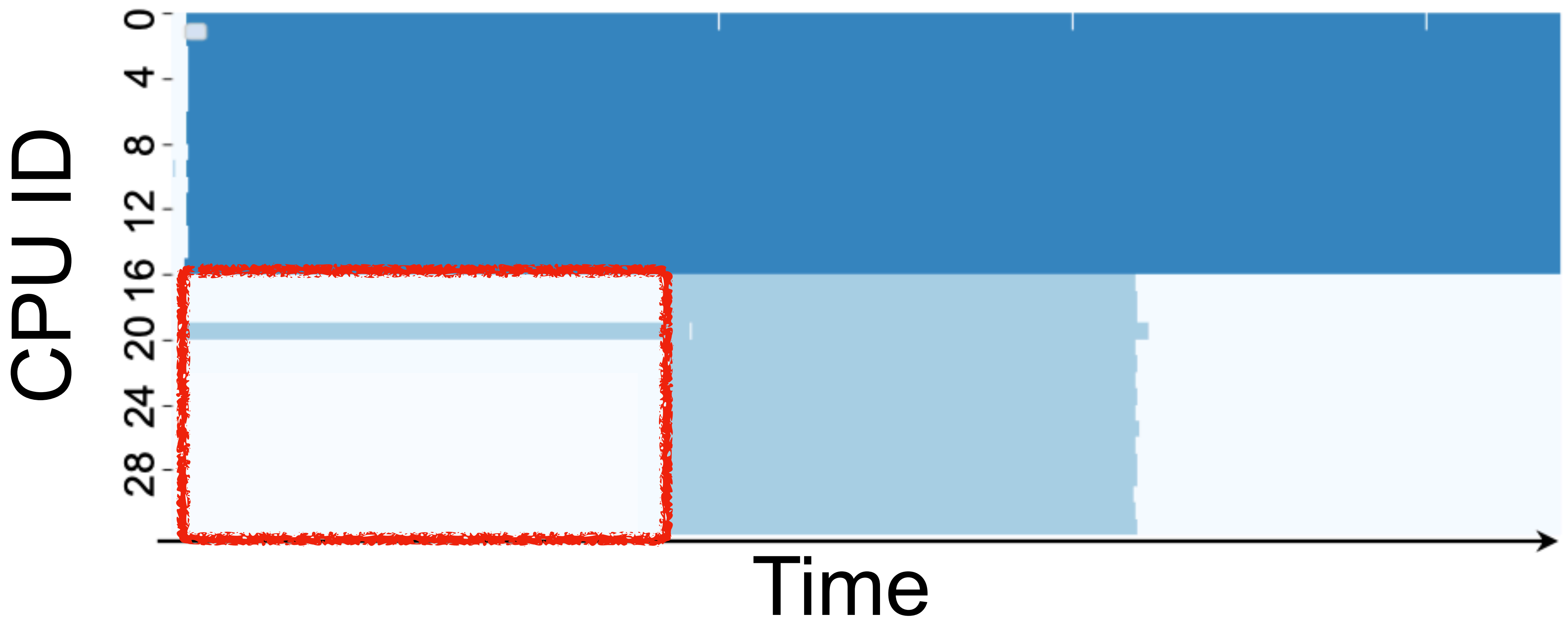
Why Linux Cannot Support Efficient CPU Reservation?

CPU pinning	✅ Provides exclusive CPU access	❌ Disallows sharing
Weighted sharing	✅ Sharing-friendly	❌ Incorrect reservation

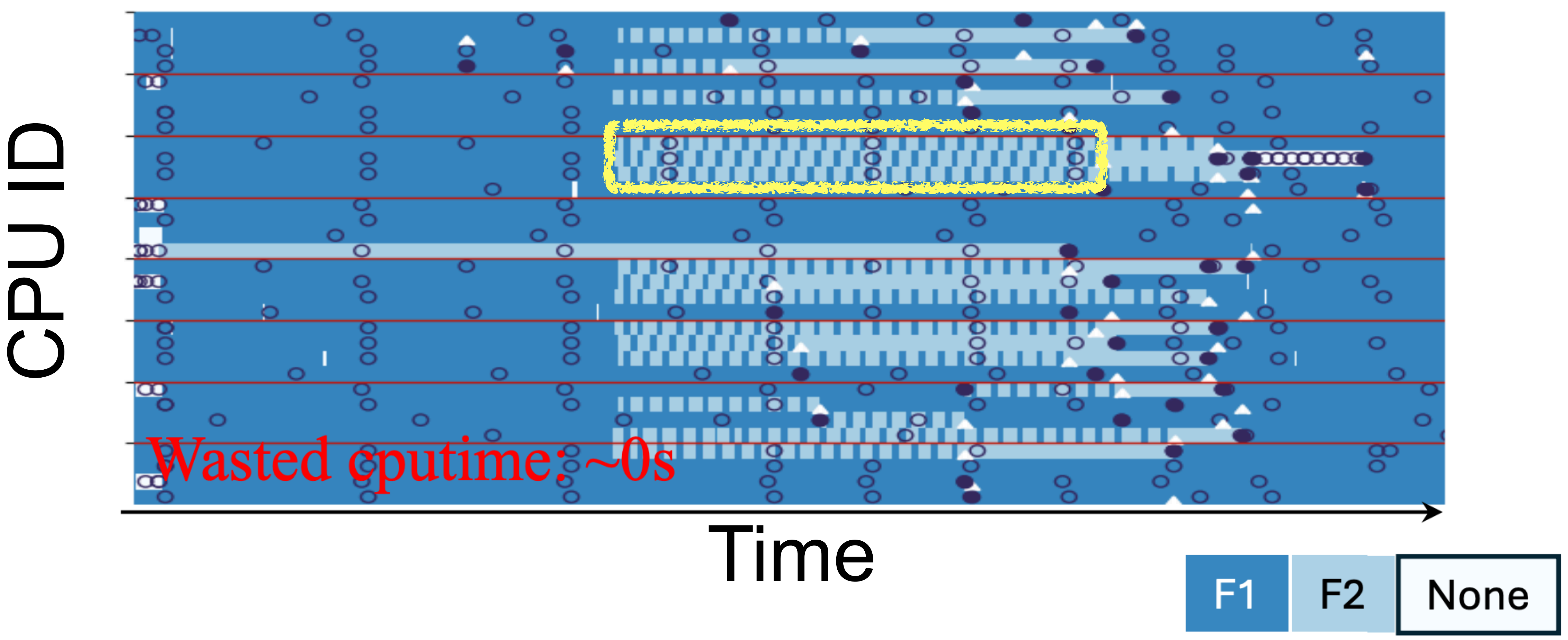
Example: F1 and F2 runs on a 32-CPU worker

- F1: 32 long-running threads, "paid" to reserve 16 CPUs
- F2: 1 thread, fans out to 16 threads, "paid" to reserve 16 CPUs

CPU pinning: Pin functions to their reserved CPUs



Weighted sharing: Give F1 and F2 equal share



Leopard's Solution

New cgroup interface

- cpu.resv_cpuset specifies reserved CPUs for a cgroup

Requirements for the Linux CPU scheduler

- Highest priority access to CPUs in a cgroup's cpu.resv_cpuset
- Non-exclusive on CPUs outside the resv_cpuset

Modified CFS scheduler

- No longer relies on fairness to achieve isolation
- Allows flexible policies on different cores

✅ Full access on reserved-CPUs and best-effort sharing on spot-CPUs

Outline

Introduction

Nearly Pay-for-Use model

Leopard FaaS system

Evaluation highlights

Experiment Setup

Workloads

- Invocations with CPU/memory usage changes overtime

Billing Models:

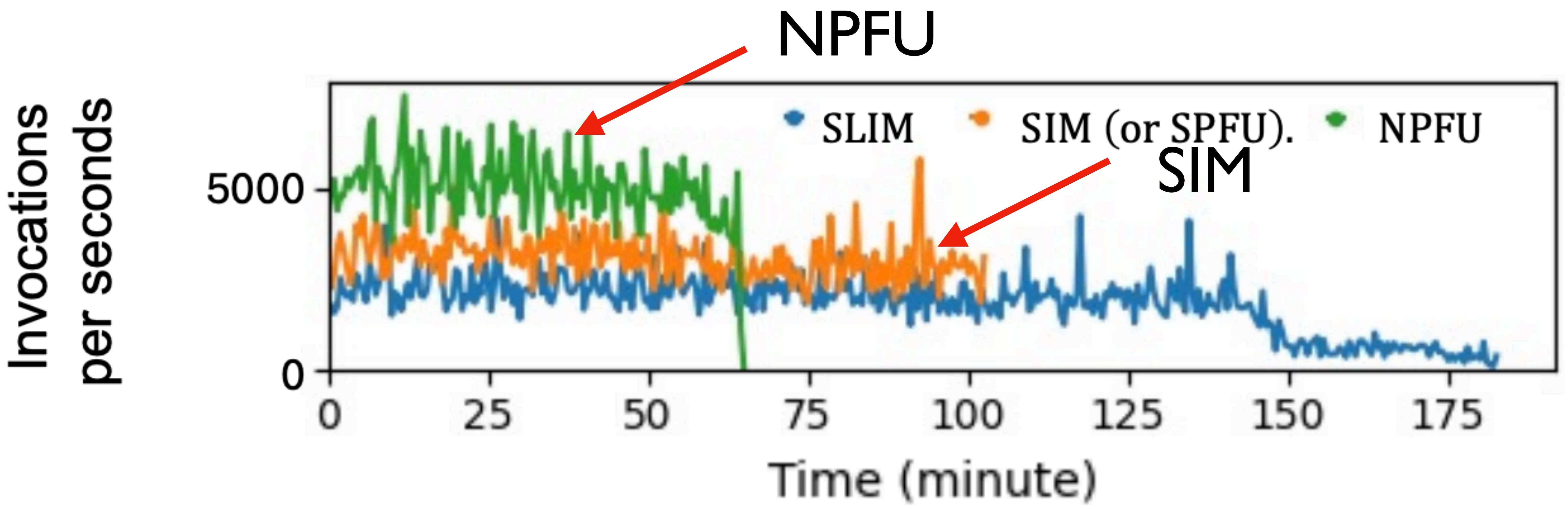
- Static Linear Interactive-only Model(SLIM): ***cost = duration × (C memory limit)***
- Static Interactive-only Model(SIM): ***cost = duration × (C_1 memory limit + C_2 CPU limit)***
- Strict-PFU(SPFU): ***cost = duration × (C_1 avg memory + C_2 avg CPU)***
- Nearly-PFU(NPFU): ***4 knobs, used/lent billing function***

Cluster set:

- 1 client node and 9 Leopard nodes

How Does Leopard (w Nearly-PFU) Perform on Provider Side?

The throughput for SLIM, SIM, and Nearly-PFU billing models

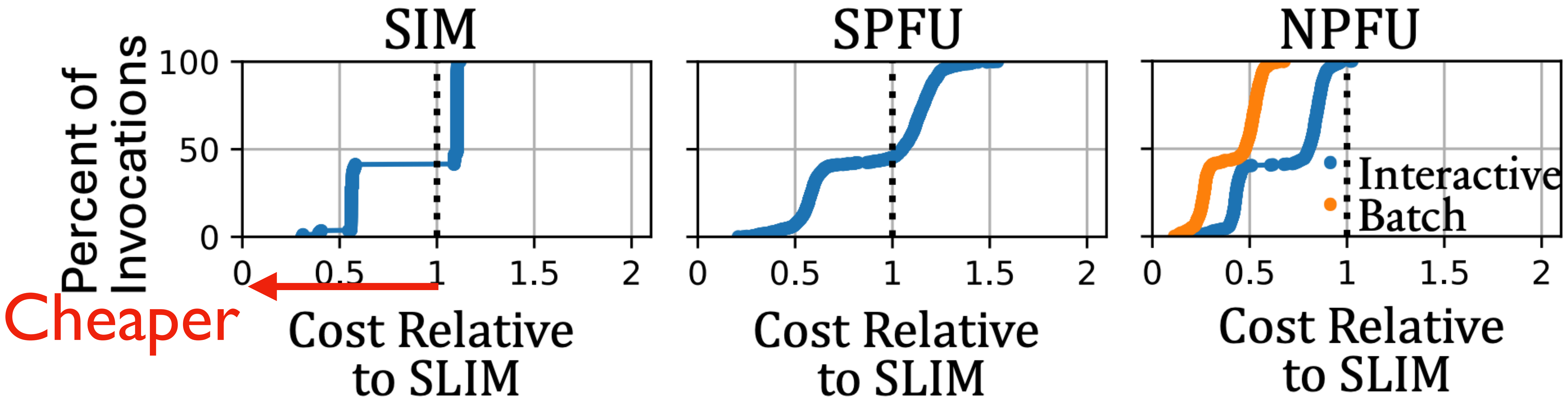


- Going from SLIM to SIM leads to a 1.3x increase in throughput
- Switching to Nearly-PFU provides an additional 1.6x improvement
  - One function's idle resources can be used to satisfy another's non-urgent demand ⇒ higher overall utilization

Can Leopard (w Nearly-PFU) Save Customer Cost?

Fix provider revenue and only compare customer cost

The CDF of invocation cost relative to those running with SLIM



- With SIM, approximately 50% of invocations save money
- For SPFU, some functions cost more than 50%
- Nearly-PFU reduces the cost of nearly every invocation
  - Give discount on idle or non-urgent resources without effecting the provider revenue

More detailed experiments in the paper!

Conclusion

🔍 **We found**

- Current serverless billing models are not real pay-for-use

🧩 **We designed** *Nearly Pay-for-use*

- For customers: approximate ideal PFU closer
- For providers: as profitable as today's models

🐆 **We built** *Leopard*

- Support Nearly-PFU billing model
- Kernel-level changes and platform-level changes on OpenLambda

⇒ **Billing models should be considered not as an afterthought,**

but as a central part of system design