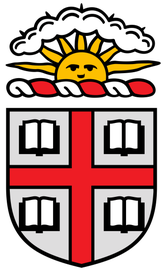
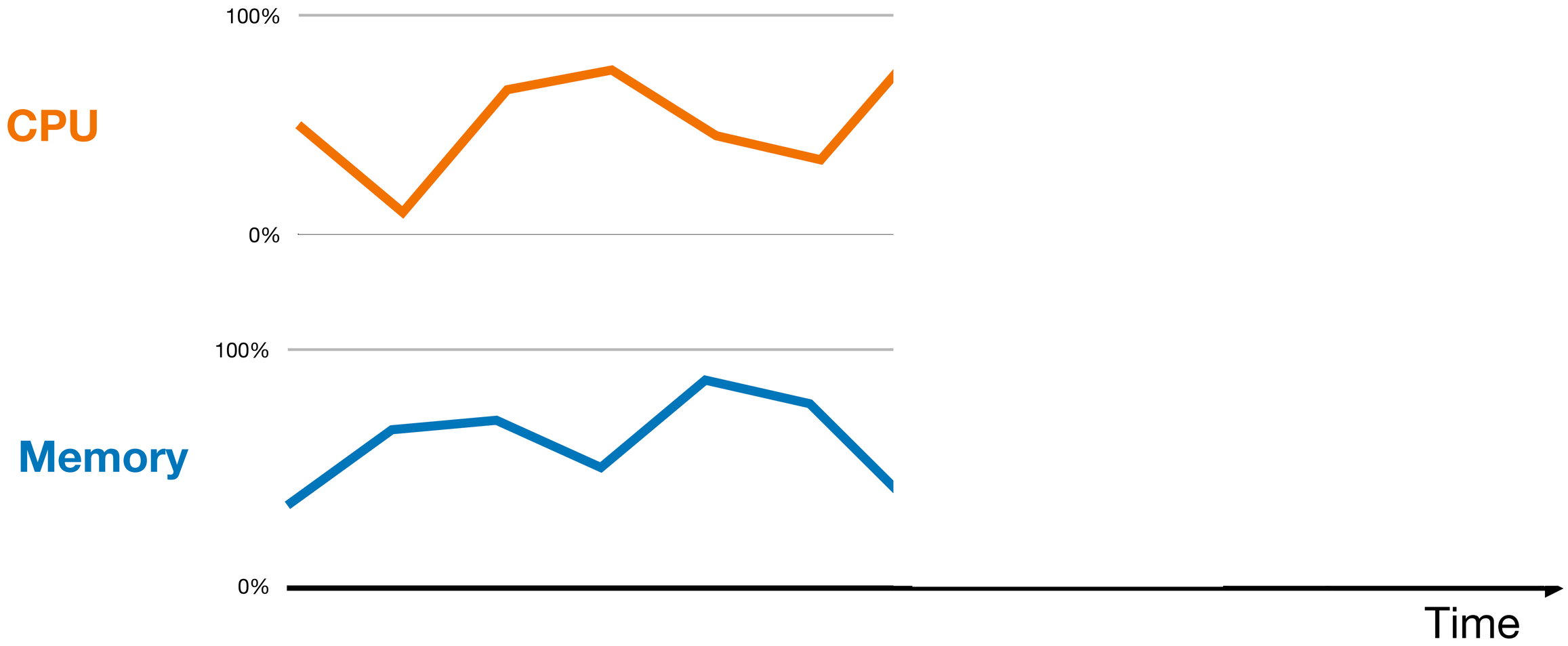
Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing

Zhenyuan (Zain) Ruan[1], **Shihang (Vic) Li[2]**, Kaiyan Fan[1], Marcos K. Aguilera[3],
Adam Belay[1], Seo Jin Park[4], Malte Schwarzkopf[2]

[1]MIT CSAIL [2]Brown University [3]VMware Research by Broadcom [4]USC

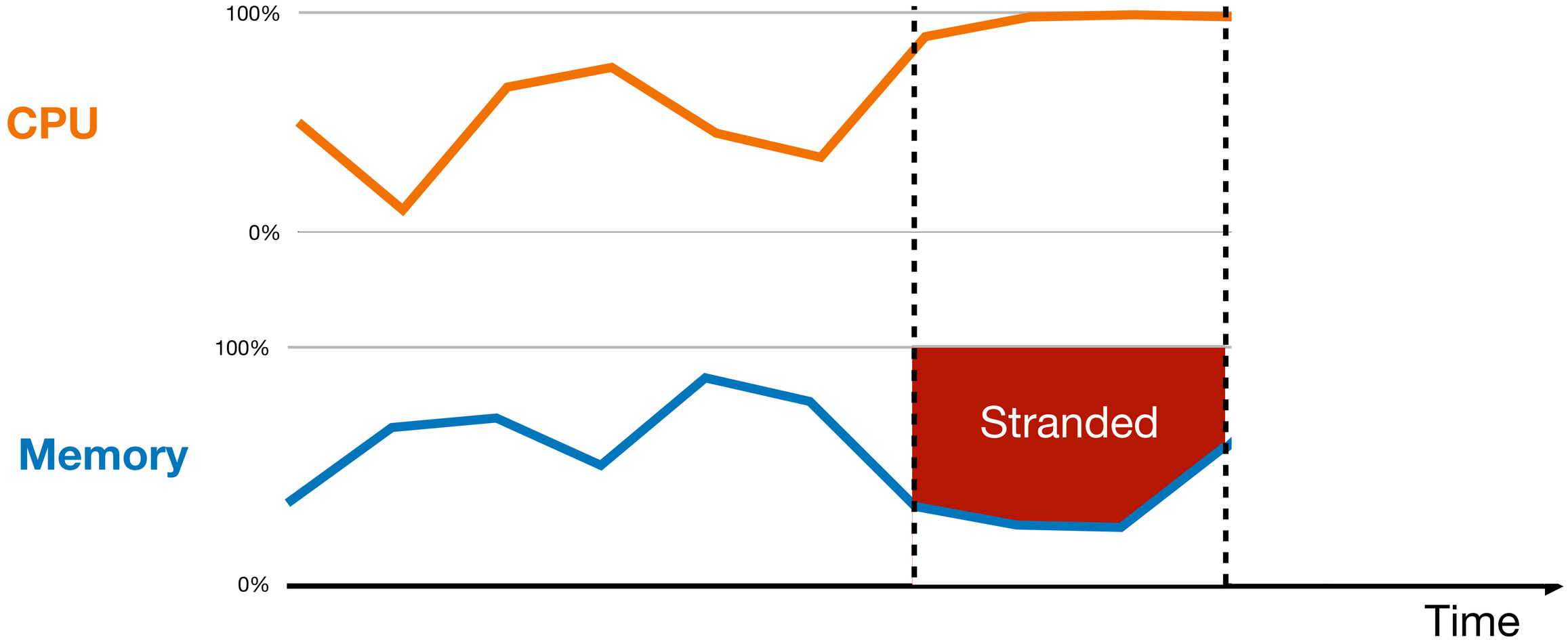


Application Resource Demand Varies Over Time

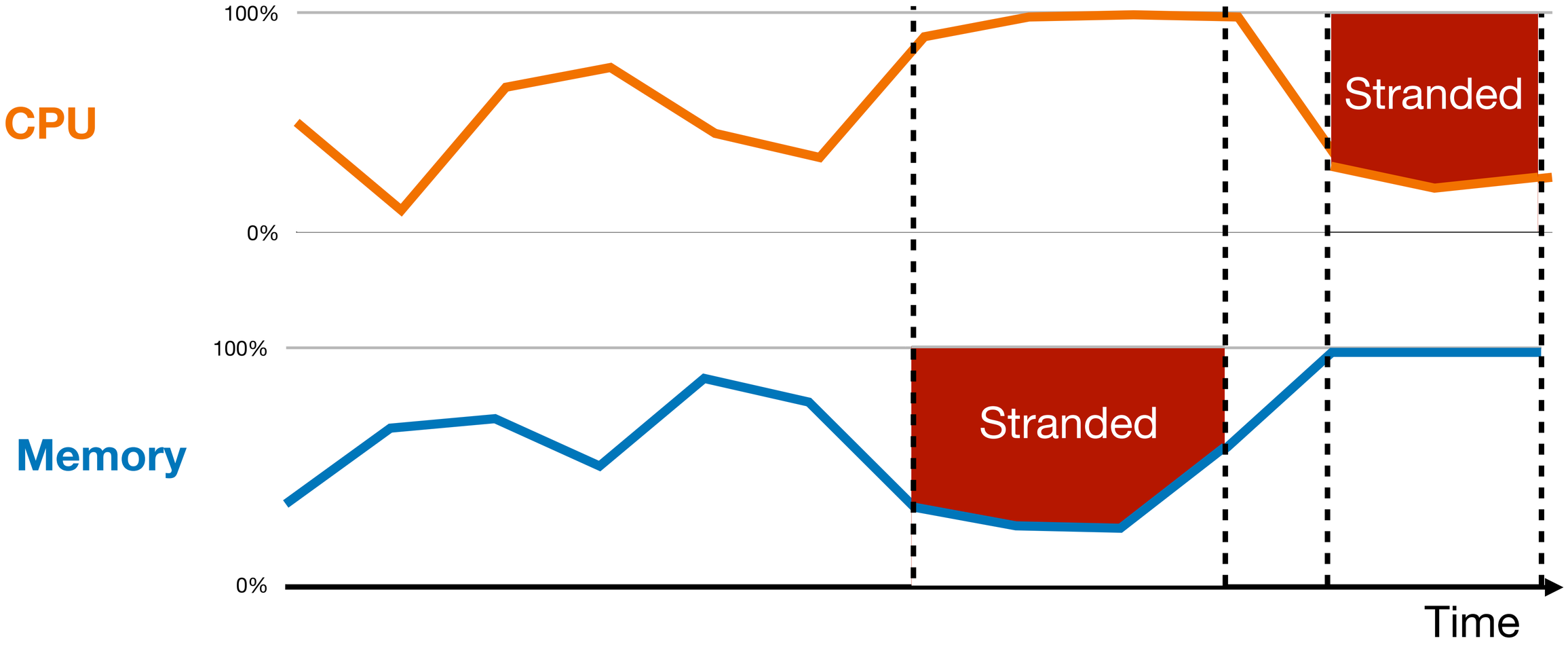


Resource Stranding

As one resource bottlenecks, others are left idle.



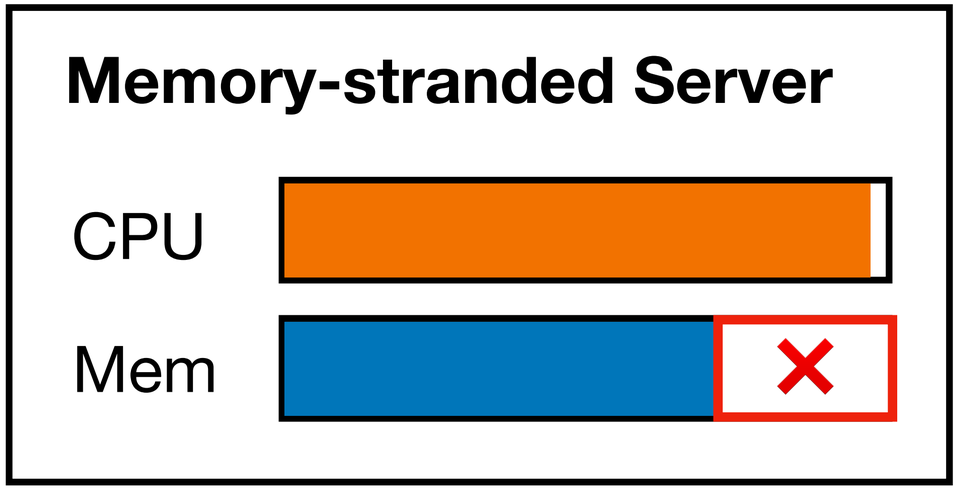
Stranded Resource Varies Over Time



Resource Stranding is Common

Resource Stranding is Common

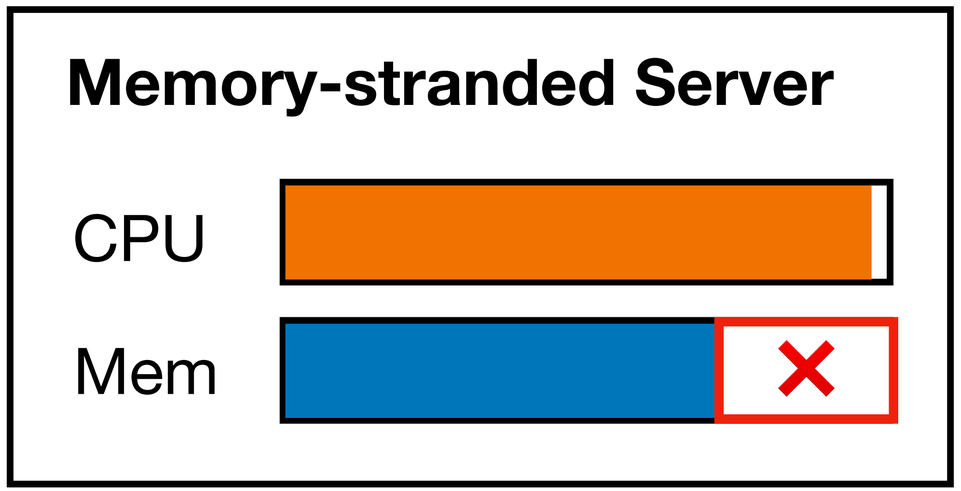
>30% Memory stranded
[Li et. al]



Resource Stranding is Common

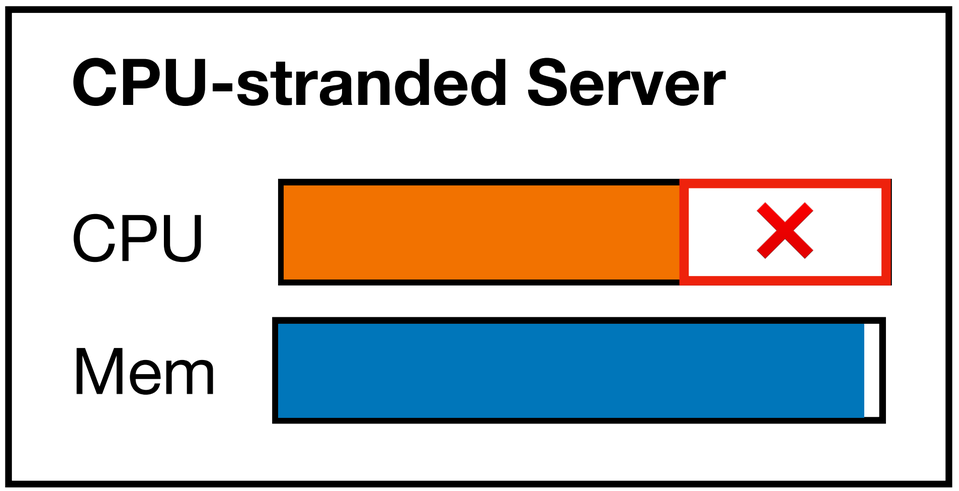
Azure

>30% Memory stranded
[Li et. al]



Alibaba Cloud

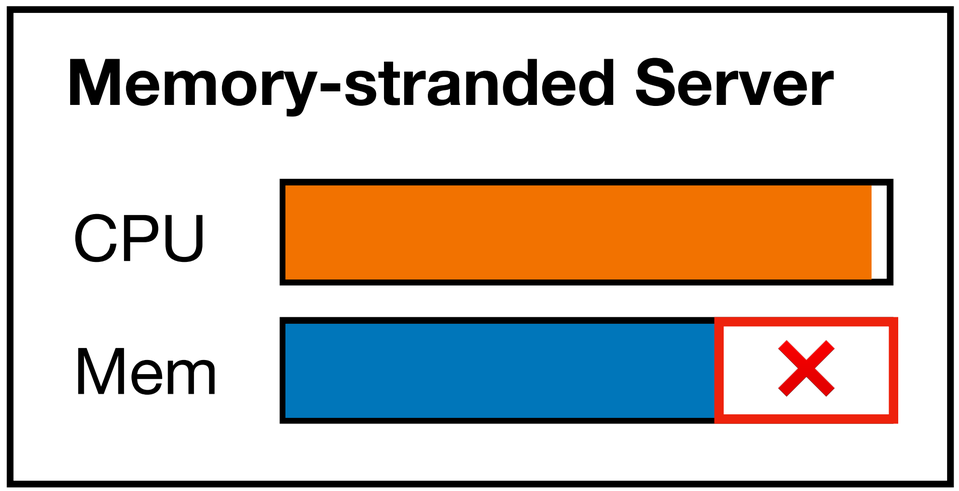
>30% CPU stranded
[Guo et. al]



Resource Stranding is Common and Costly

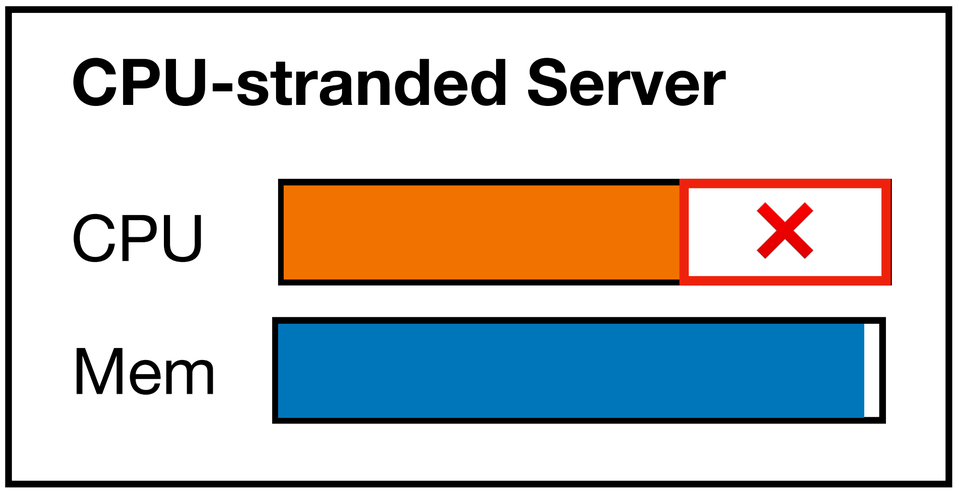
Azure

>30% Memory stranded
[Li et. al]



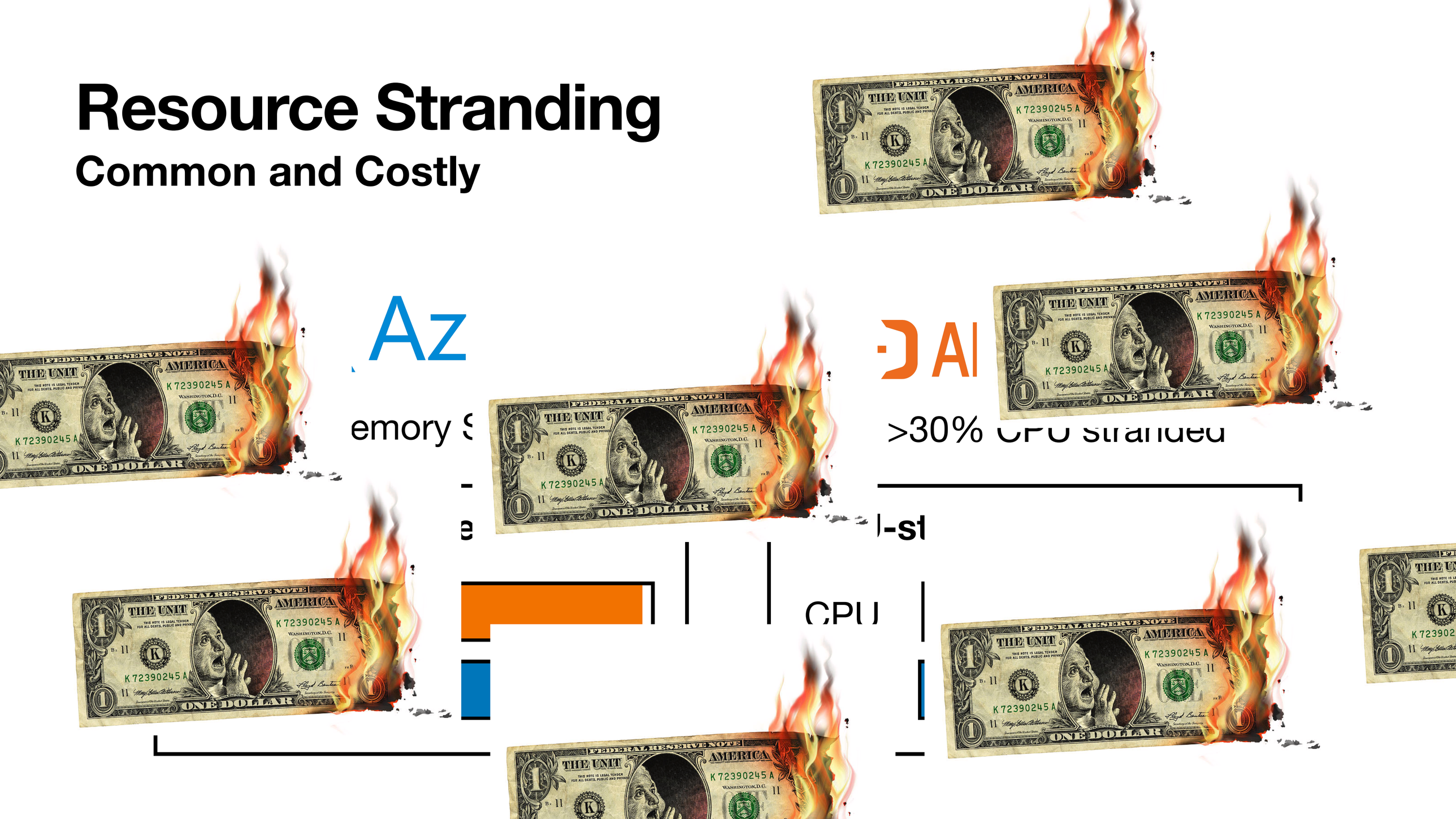
Alibaba Cloud

>30% CPU stranded
[Guo et. al]



Resource Stranding

Common and Costly

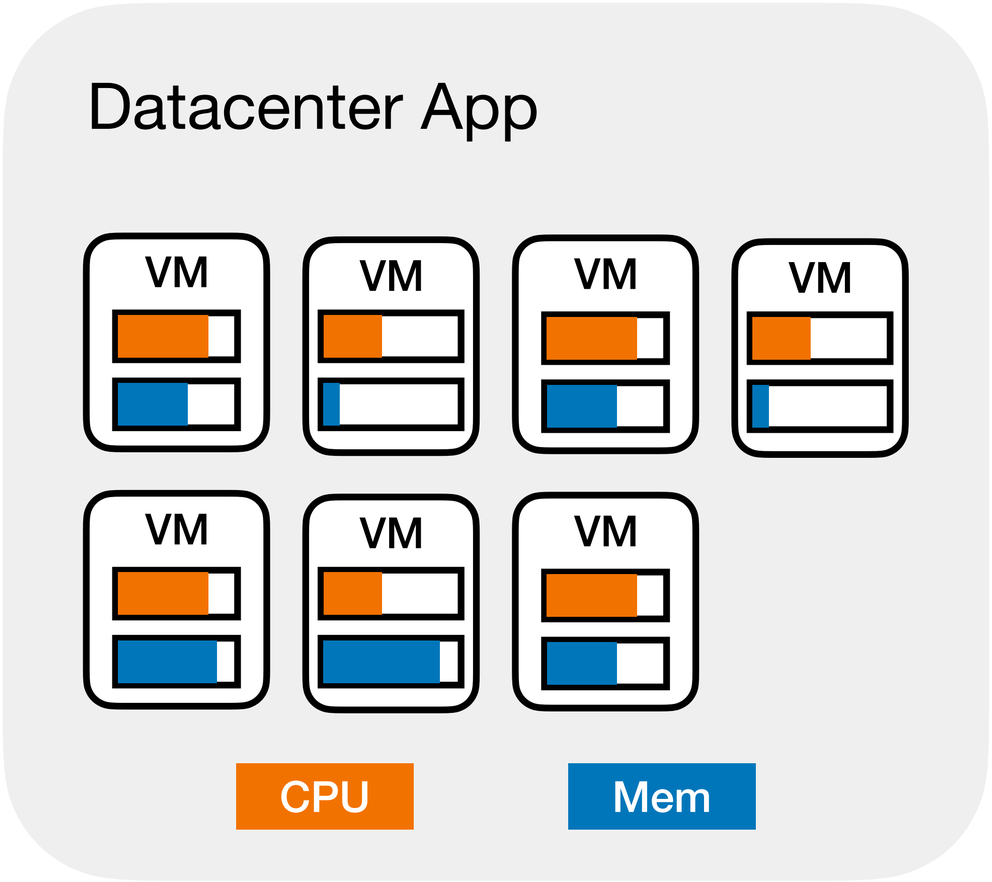


How can applications use stranded resources?

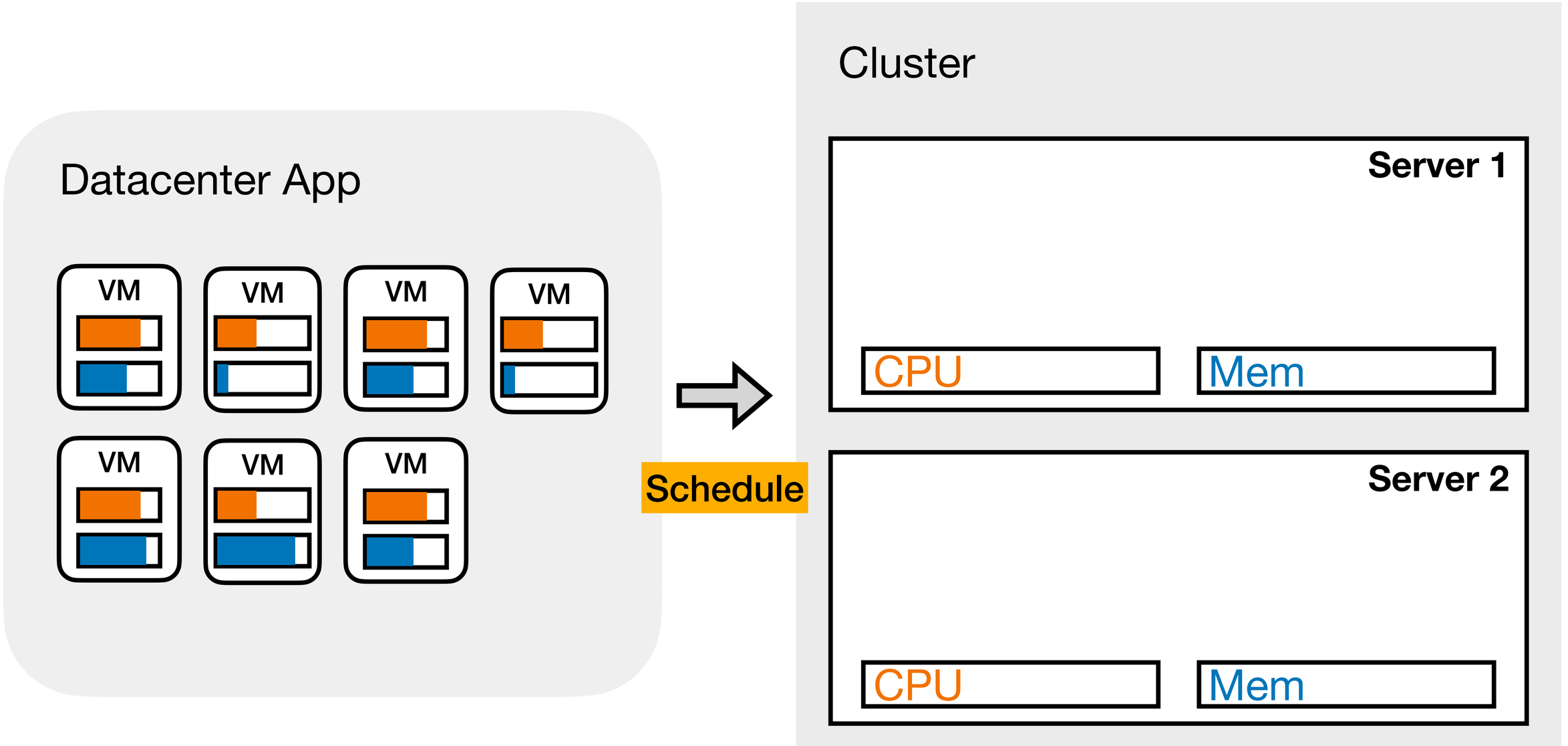
The Status Quo

Datacenter App

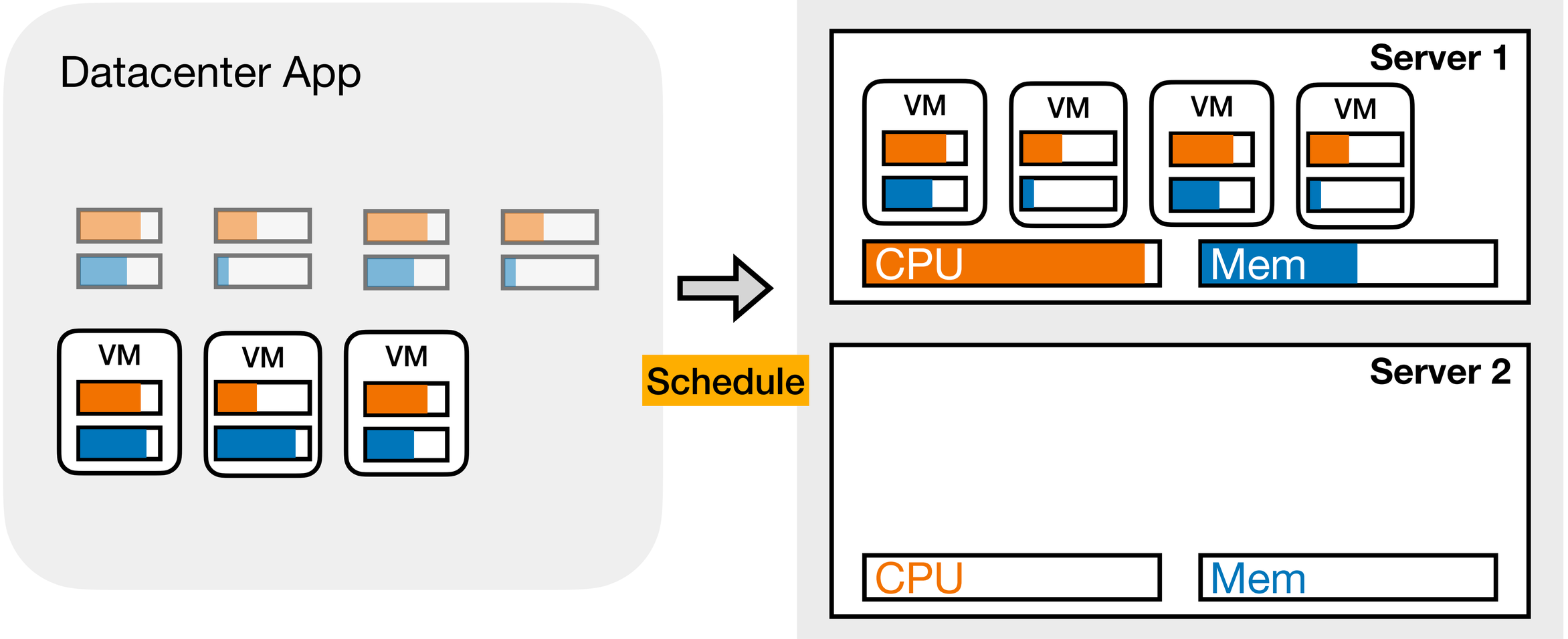
The Status Quo



The Status Quo

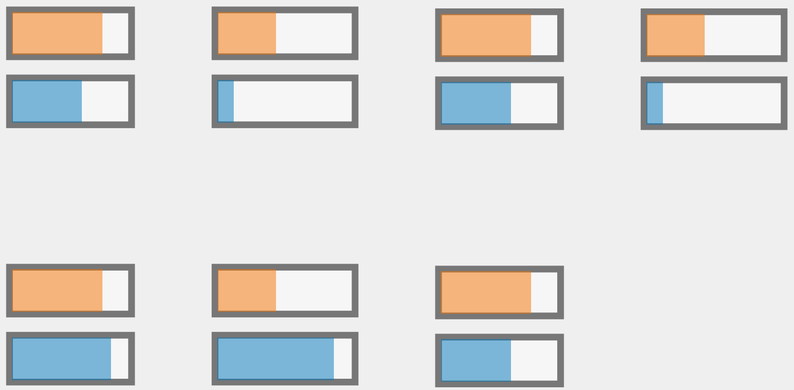


The Status Quo



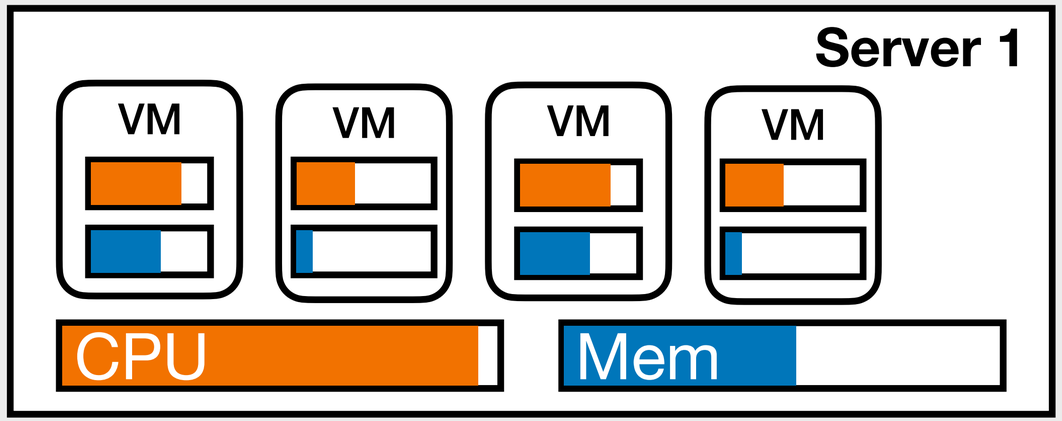
The Status Quo

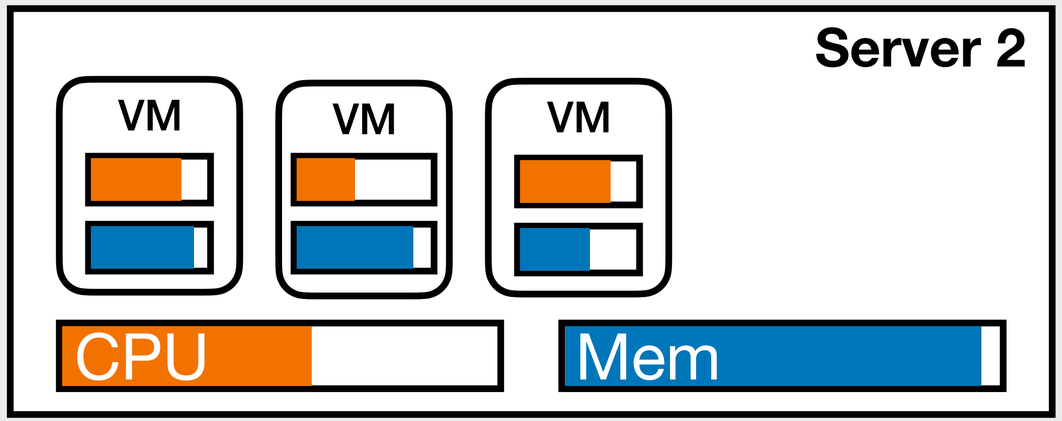
Datacenter App



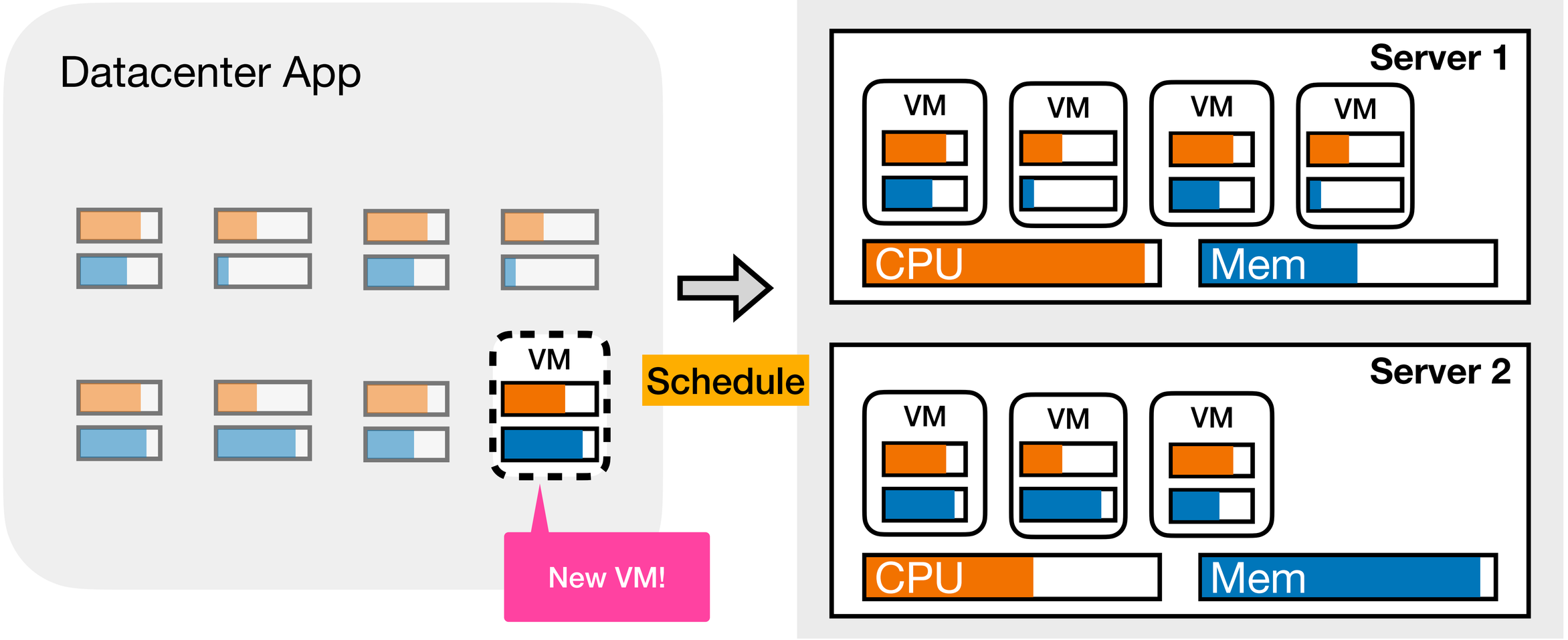
Schedule

Cluster



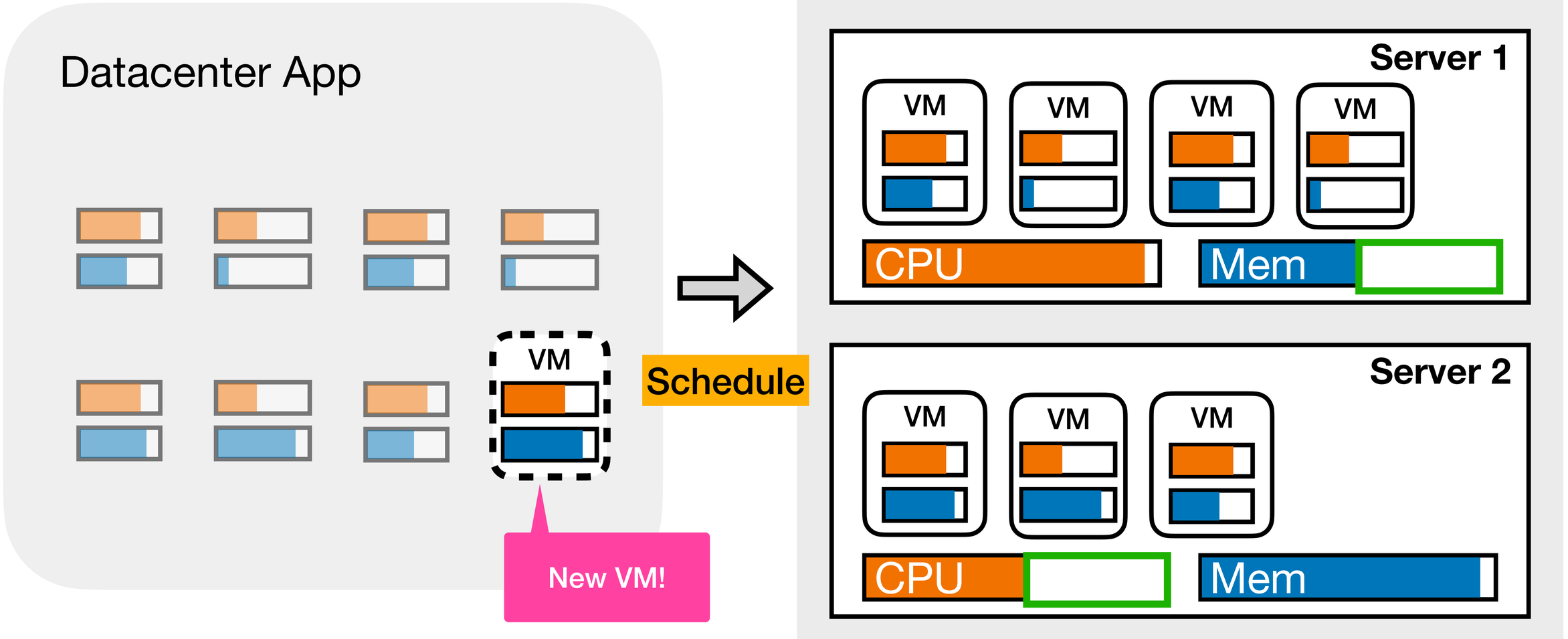


The Status Quo

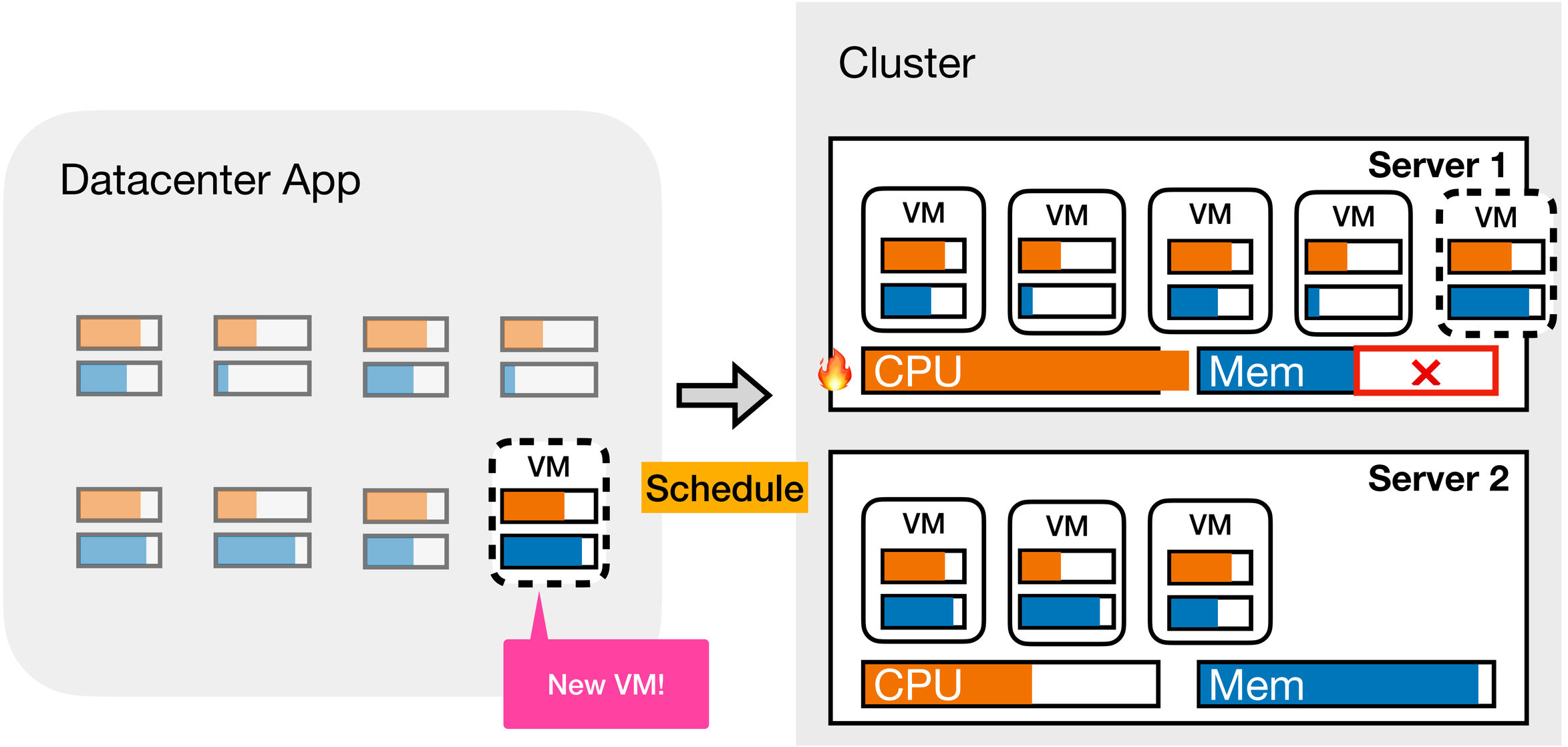


Enough Idle Resources in Aggregate

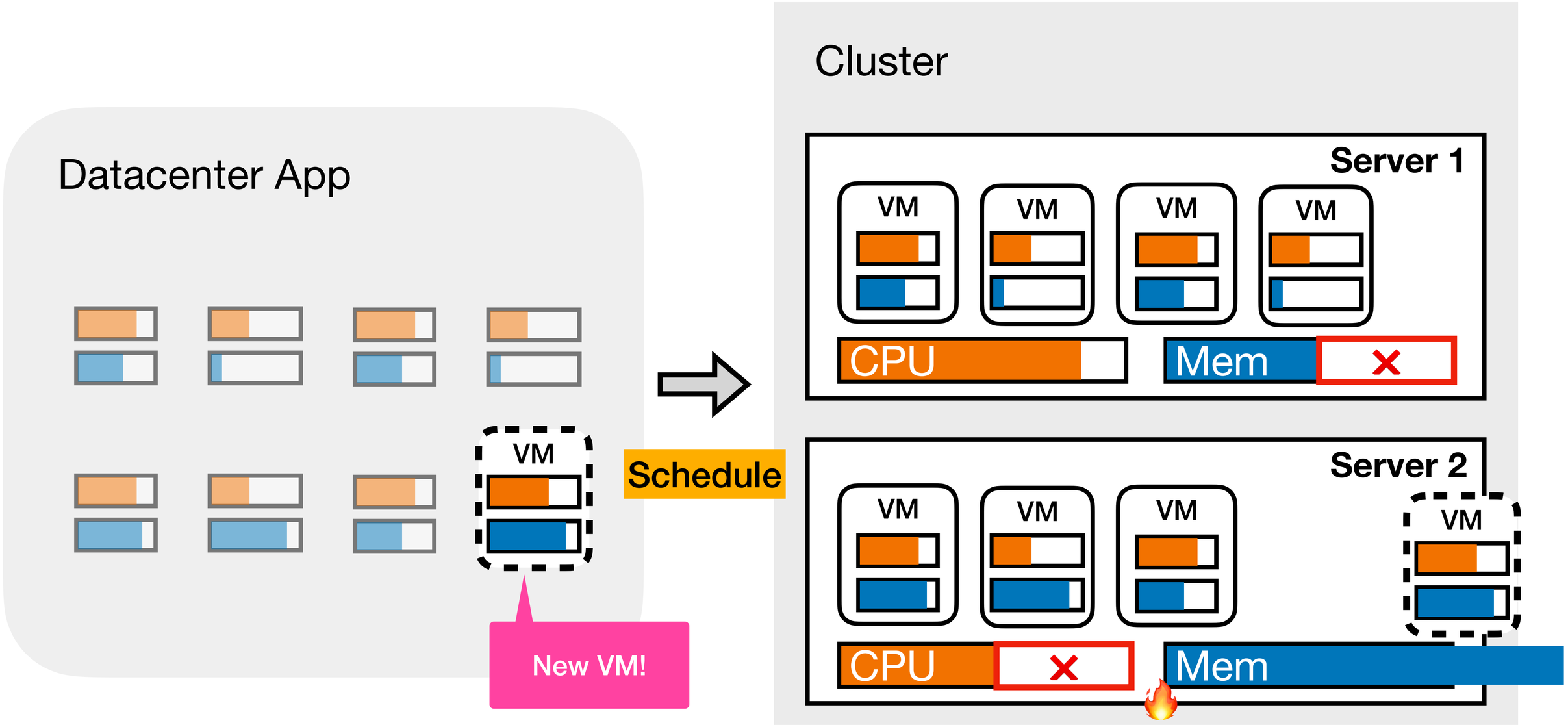
But not usable by VMs



Coarse-Grained Units Can’t Use Stranded Resources



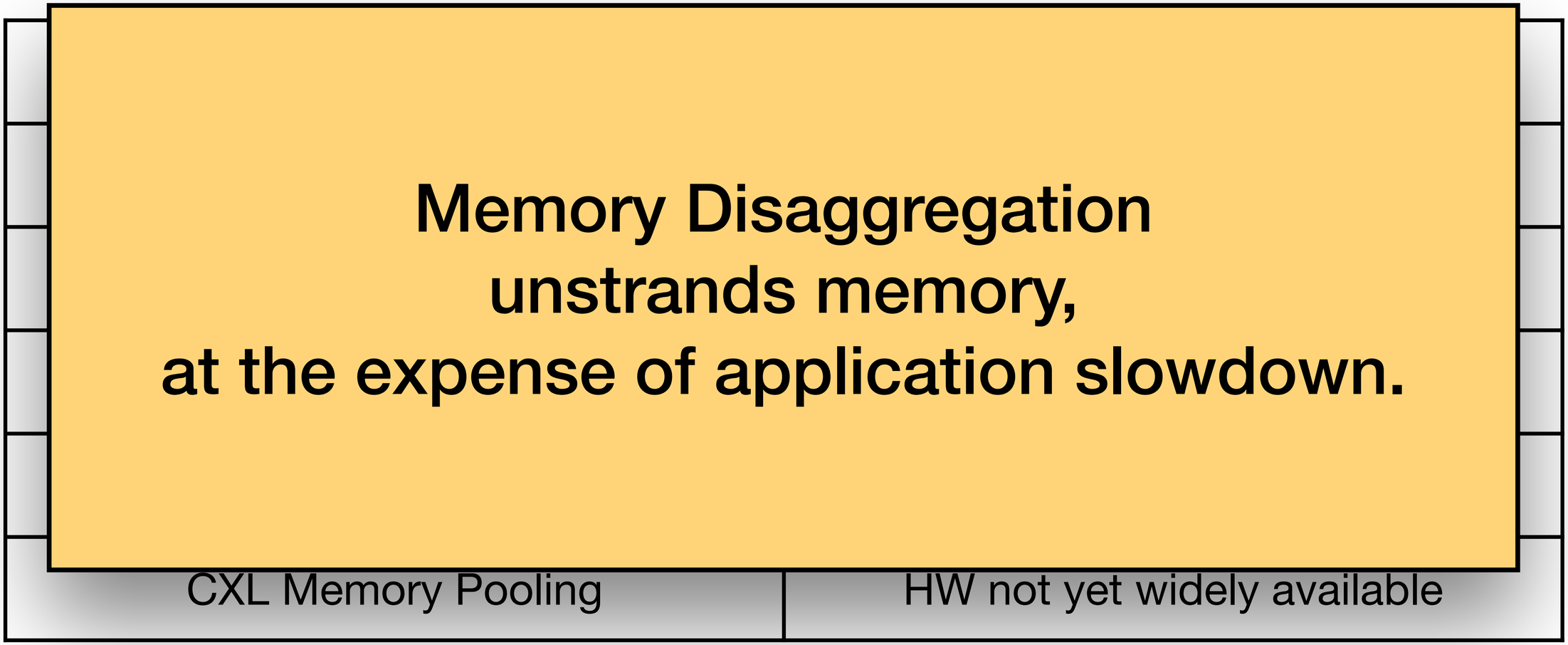
Coarse-Grained Units Can’t Use Stranded Resources



Conventional Wisdom: Mem Disaggregation

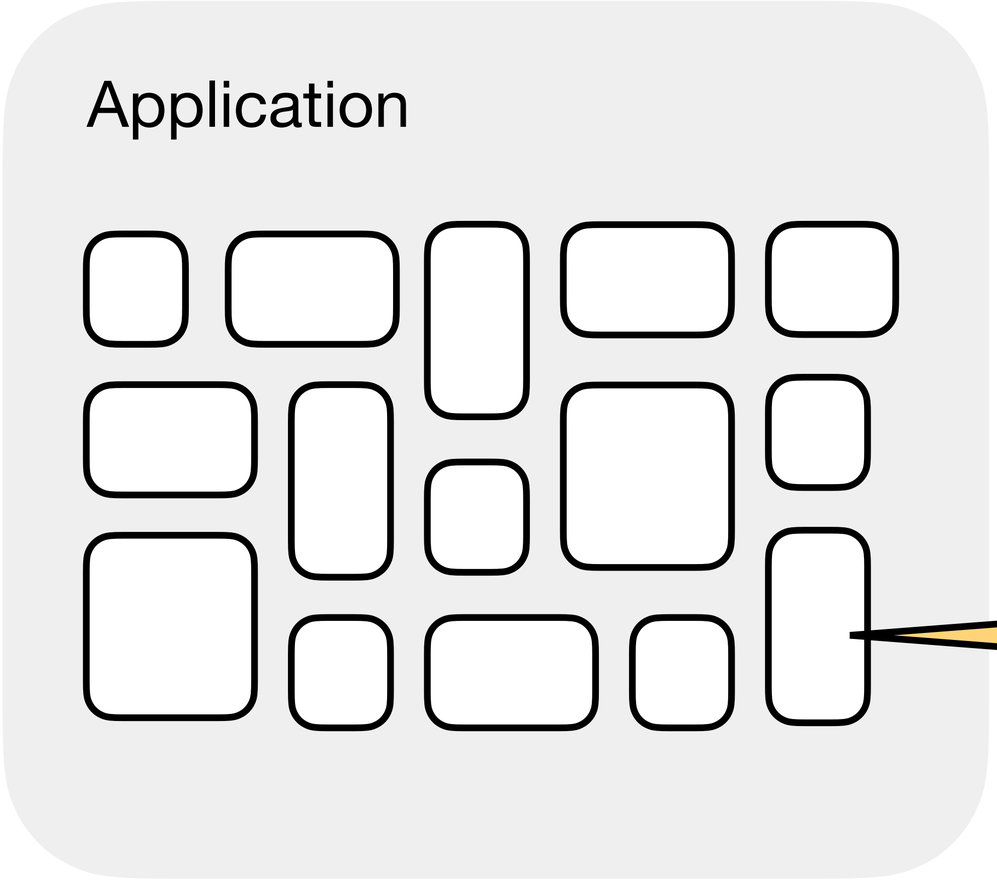
System	Overhead compared to No Disaggregation
Infiniswap [NSDI ’17]	50% (VoltDB (TPC-C), 50% working set in memory)
LegoOS [OSDI ’18]	68% (TensorFlow, 25% working set in memory)
FastSwap [EuroSys ’20]	67% (Spark, 60% working set in memory, data from Hermit [NSDI ’23])
Hermit [NSDI ’23]	43% (Spark, 60% working set in memory)
CXL Memory Pooling	HW not yet widely available

Conventional Wisdom: Mem Disaggregation



Granular Programming Frameworks

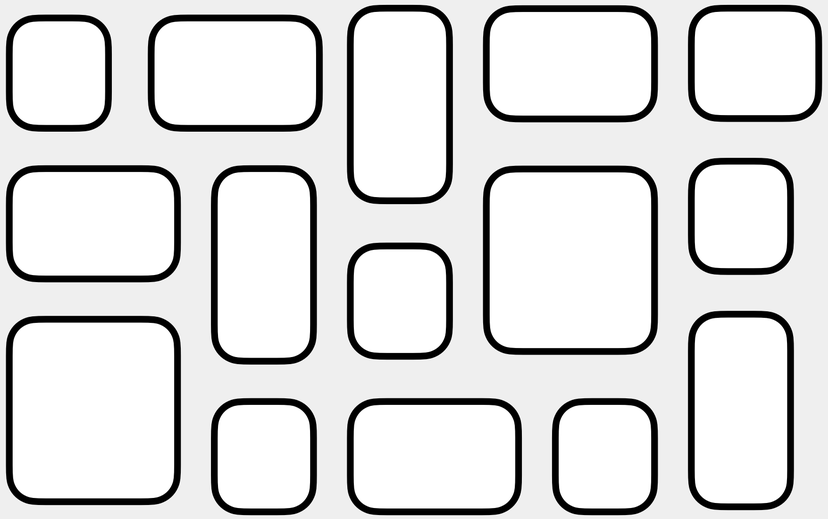
Decompose Apps into Fine-Grained Units



- Serverless Functions
- Actors (Ray [OSDI ’18])
- Proclets (Nu [NSDI ’23])
- Components (ServiceWeaver [HotOS ’23])

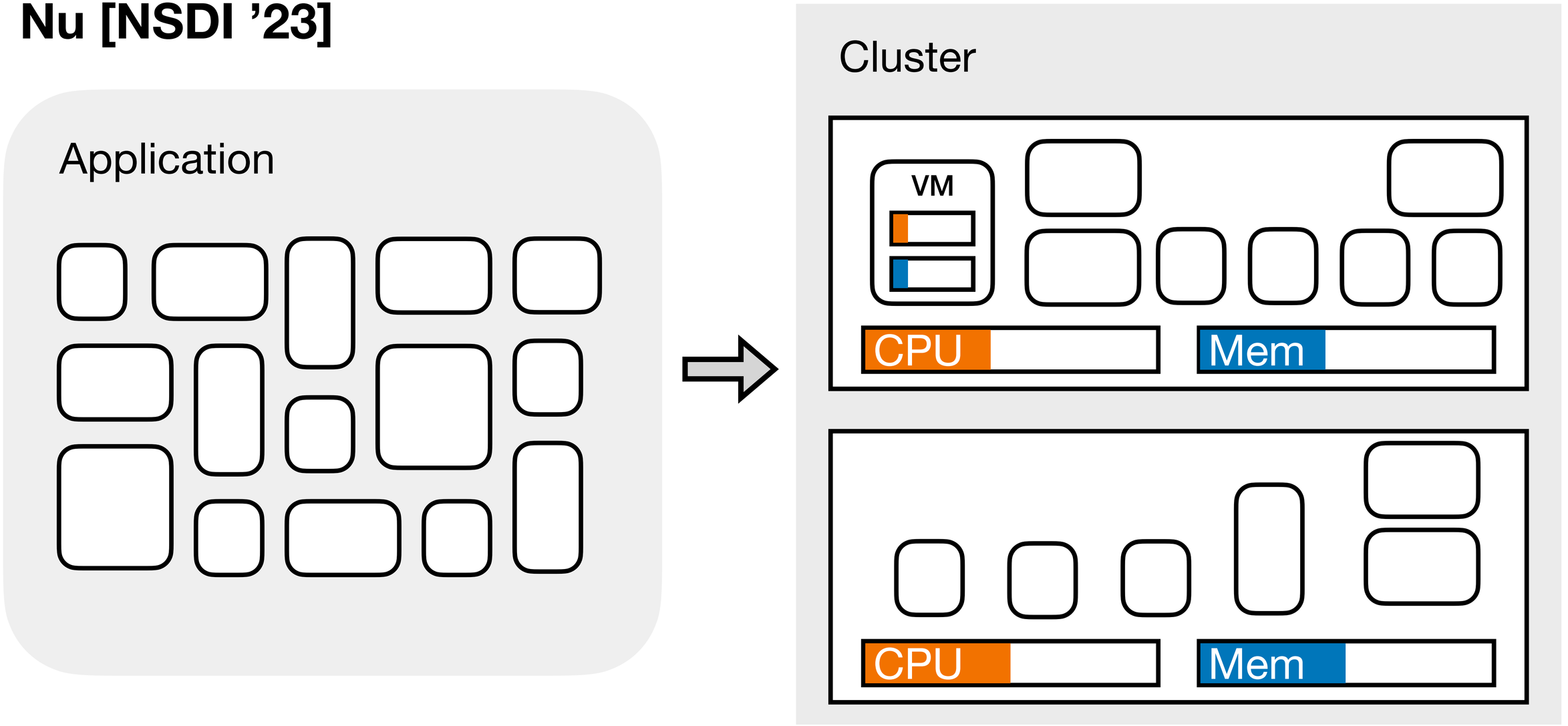
Can Fine-Grained Units Solve Stranding?

Application



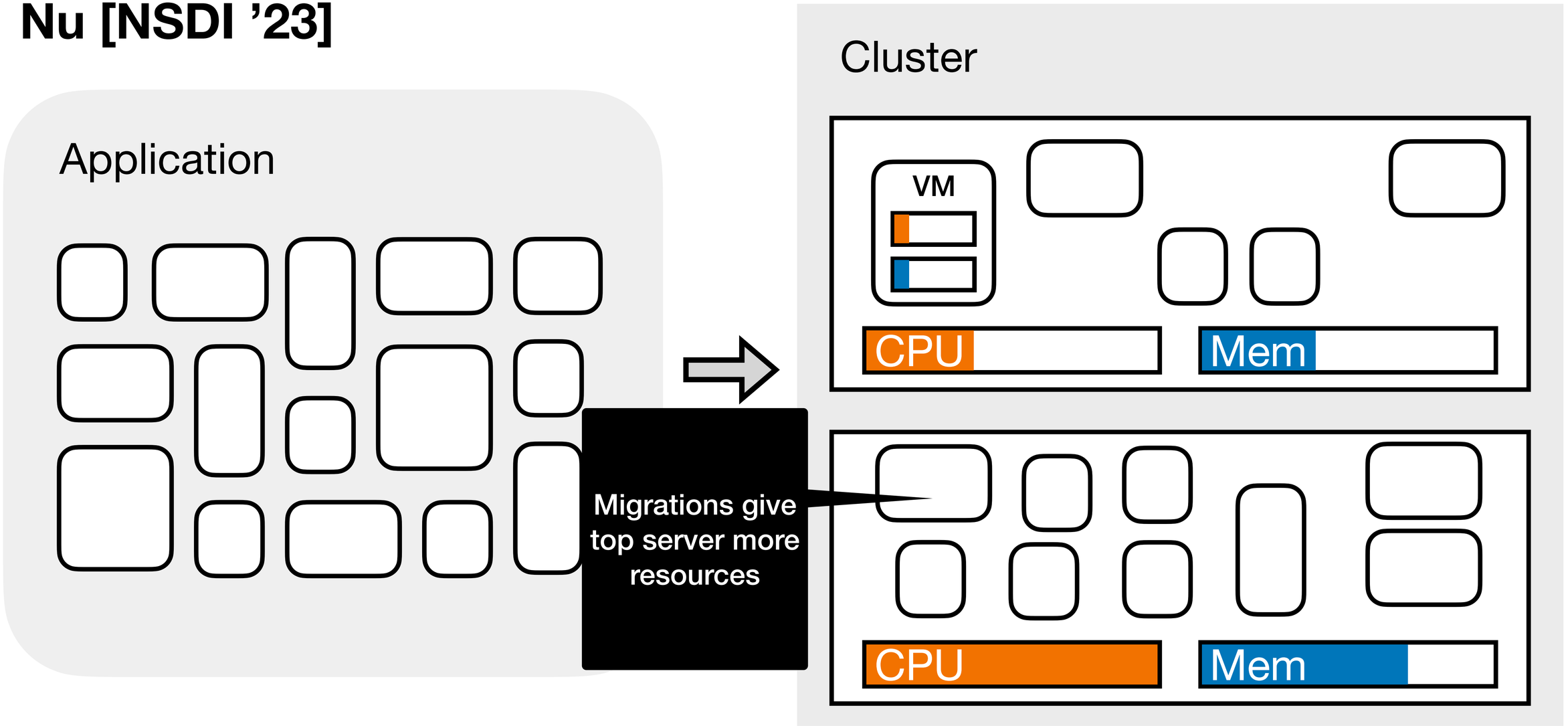
Fine-Grained Units Can Shift Resource Usage Quickly

Nu [NSDI ’23]



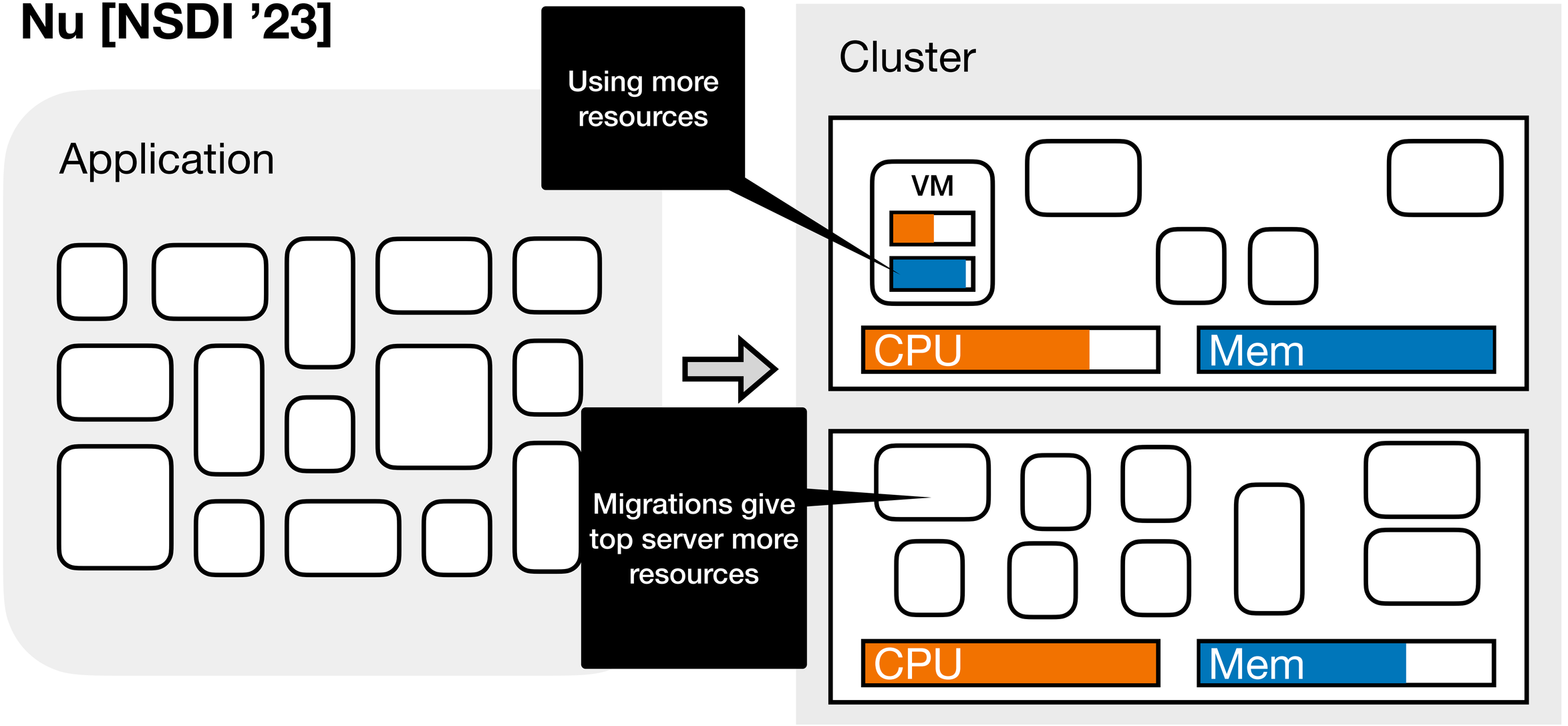
Fine-Grained Units Can Shift Resource Usage Quickly

Nu [NSDI ’23]



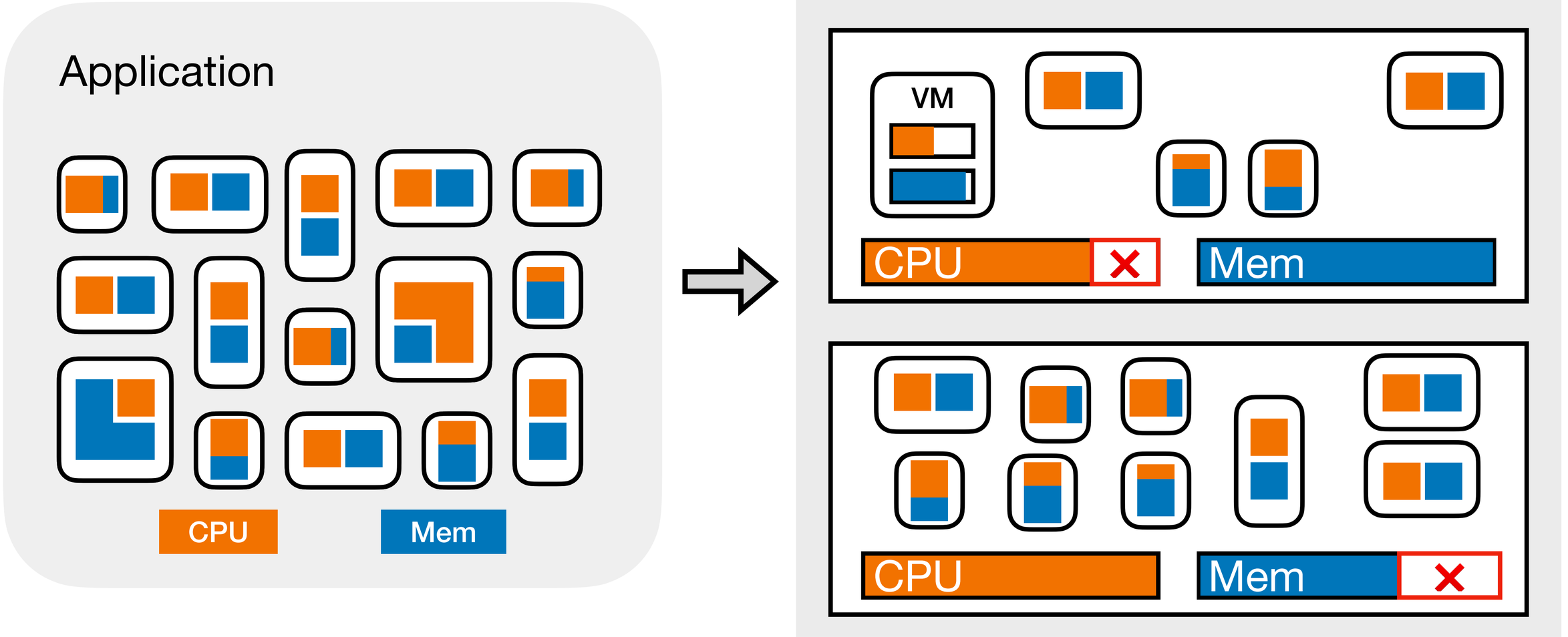
Fine-Grained Units Can Shift Resource Usage Quickly

Nu [NSDI ’23]



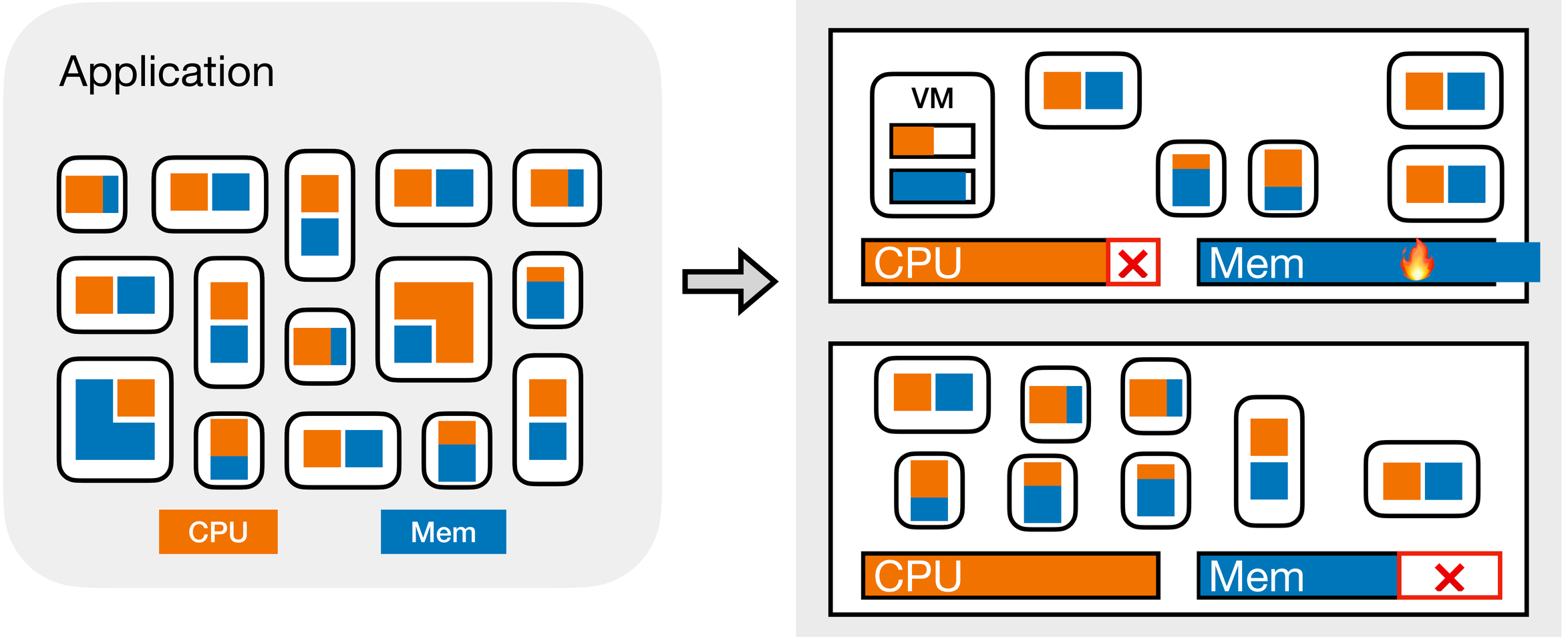
Can Fine-Grained Units Solve Stranding?

No, because of resource coupling

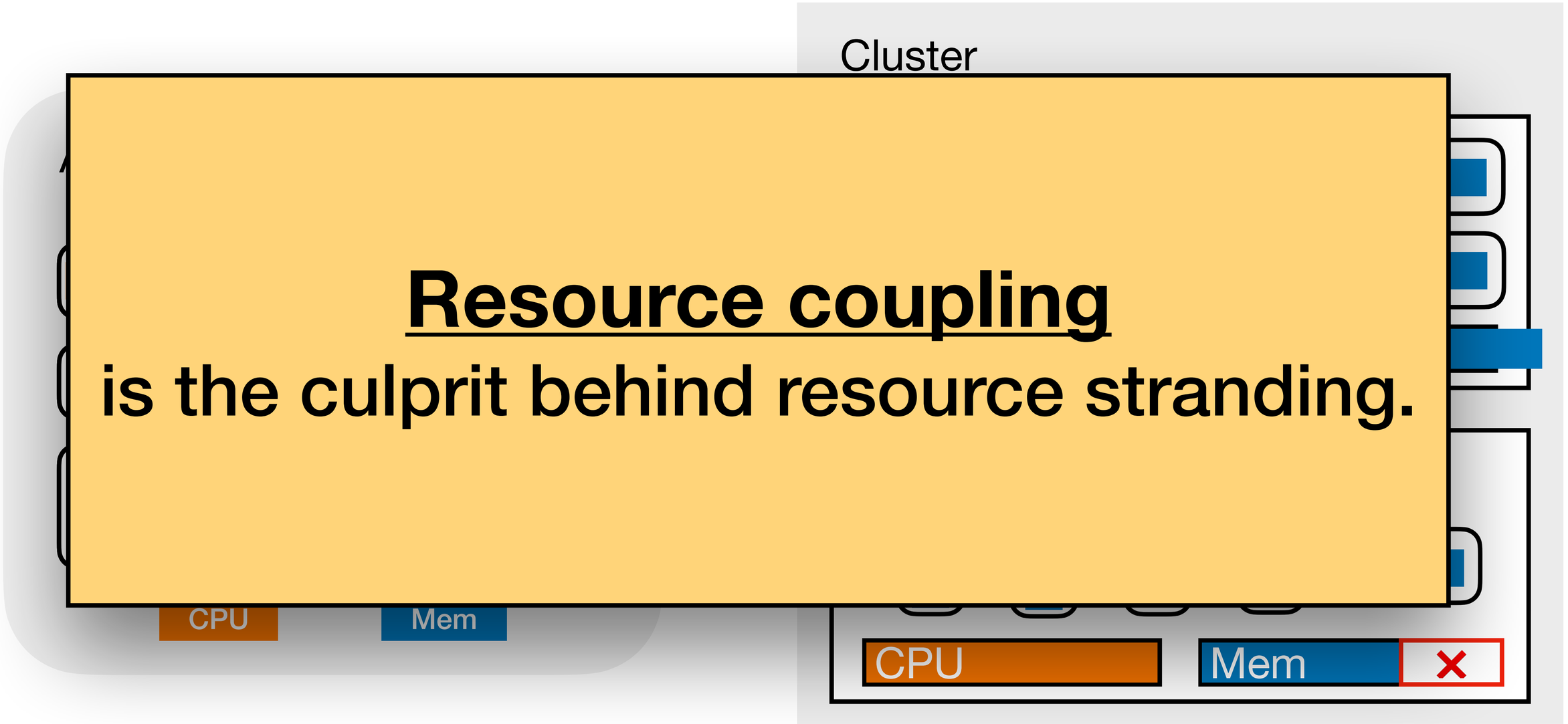


Can Fine-Grained Units Solve Stranding?

No, because of resource coupling



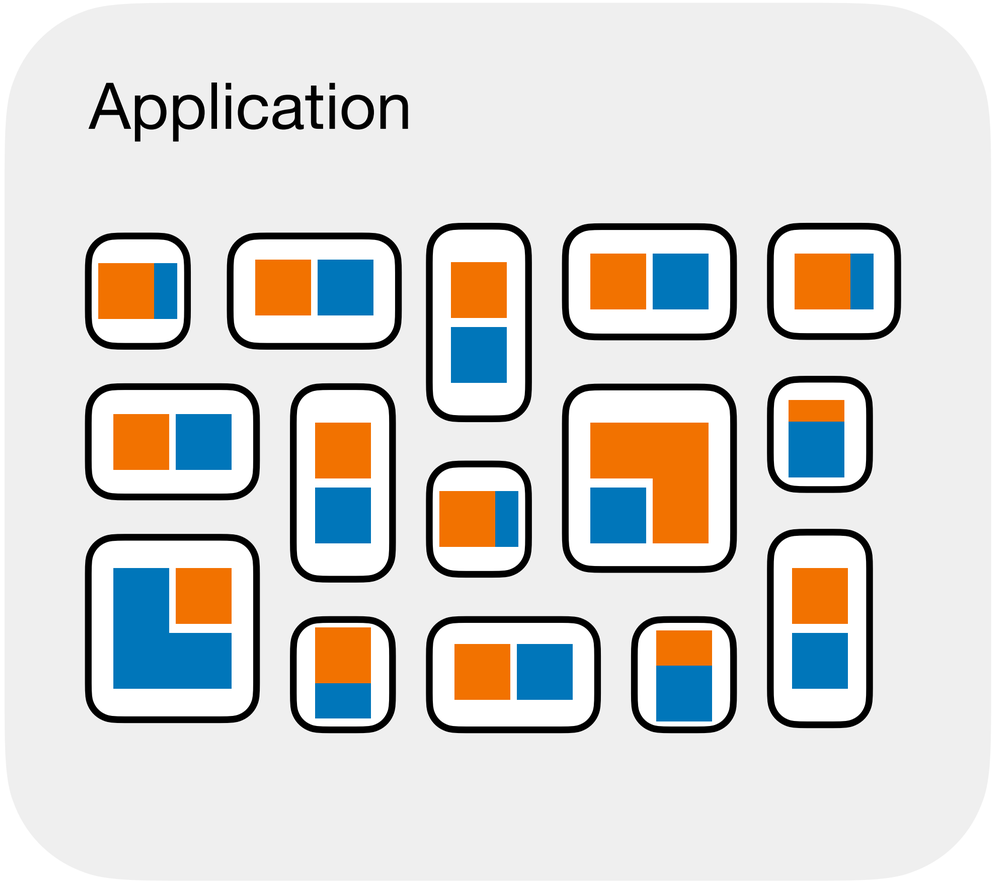
Fine-Grained Units Alone Cannot Unstrand



<u>Resource coupling</u>

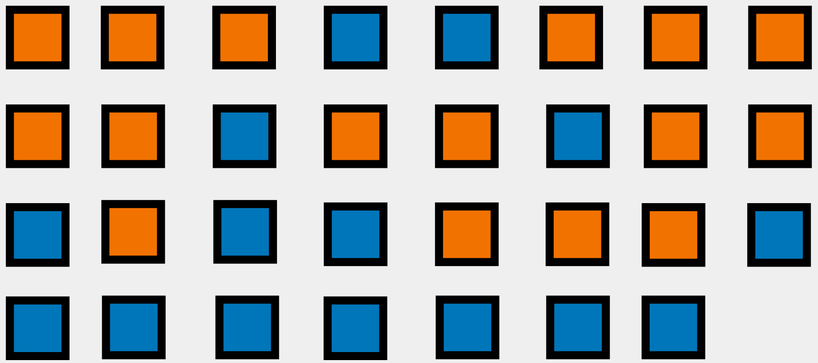
is the culprit behind resource stranding.

Resource Disaggregation, through Granular Programming

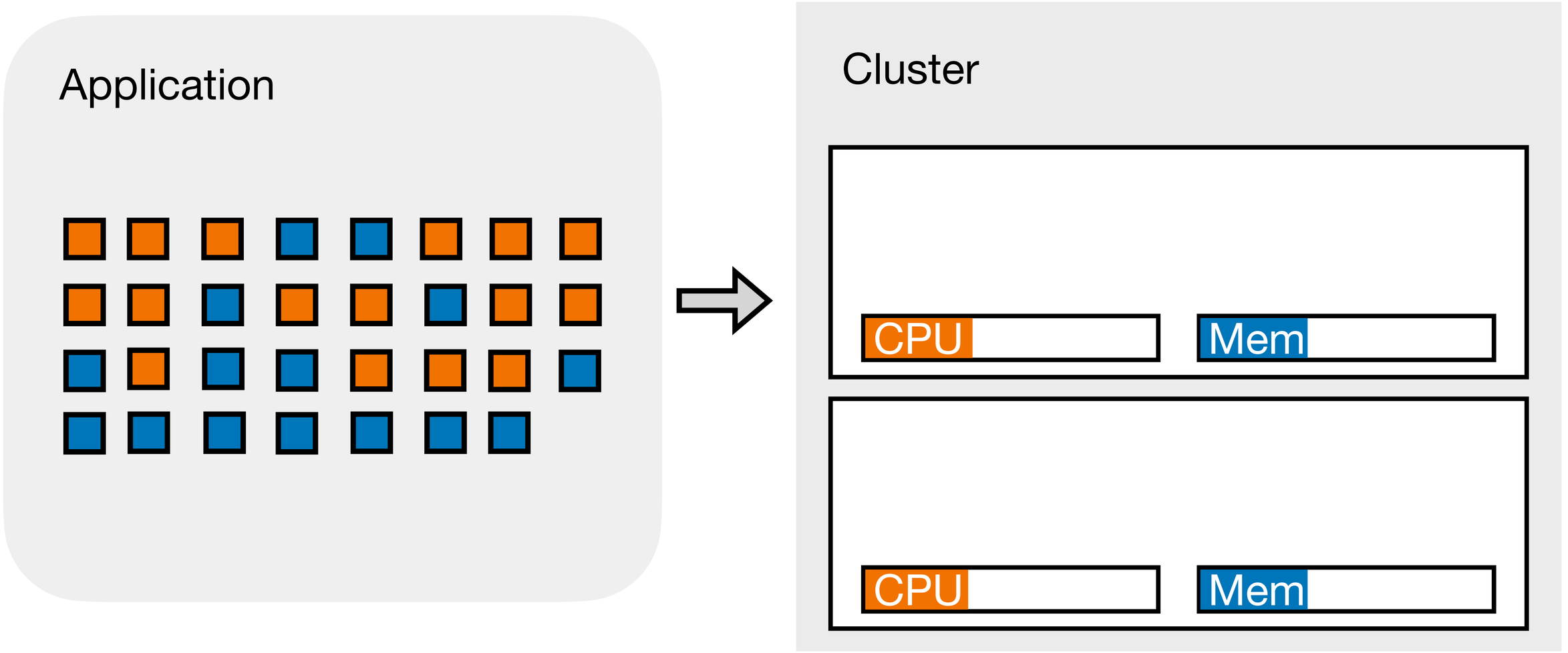


Resource Disaggregation, through Granular Programming

Application

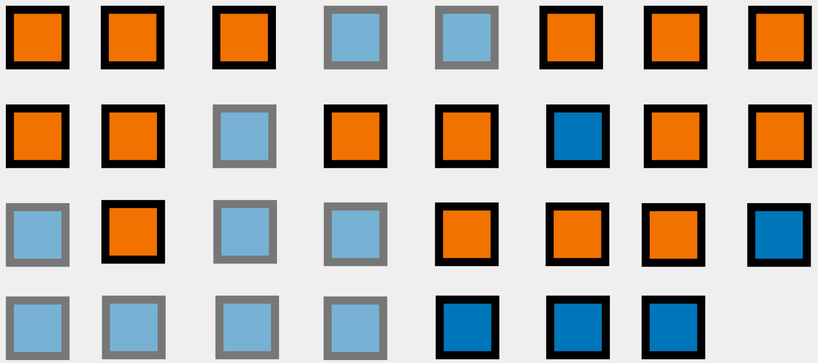


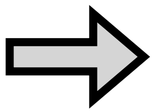
Resource Disaggregation, through Granular Programming



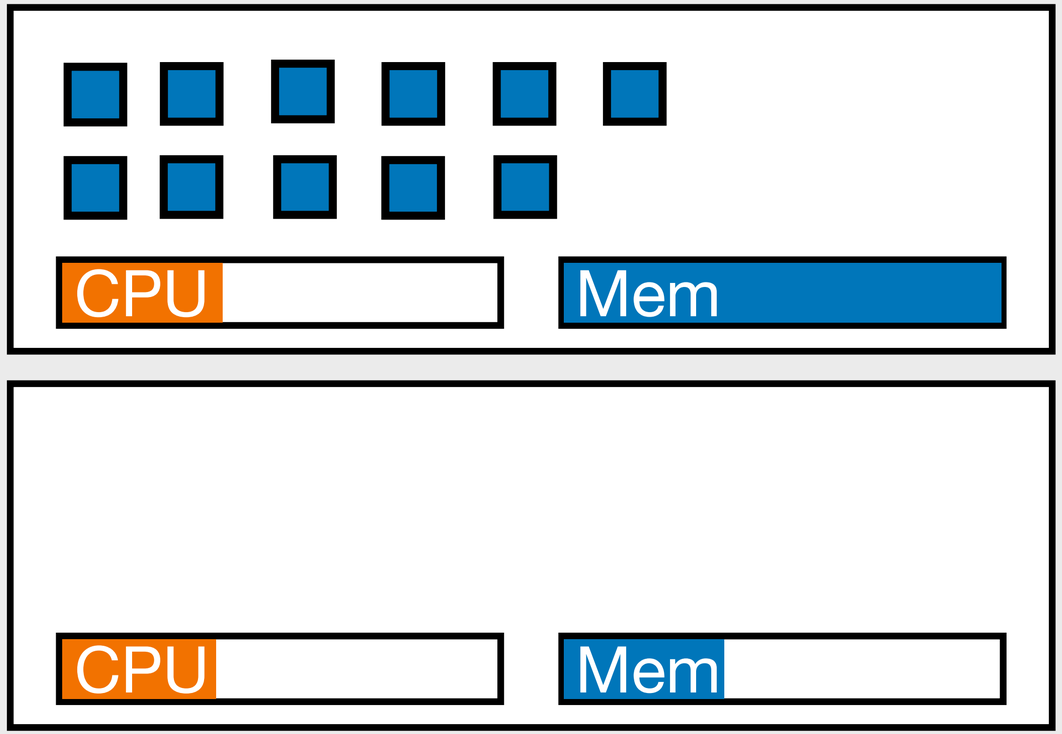
Scheduler that Leaves No Resource Stranded

Application



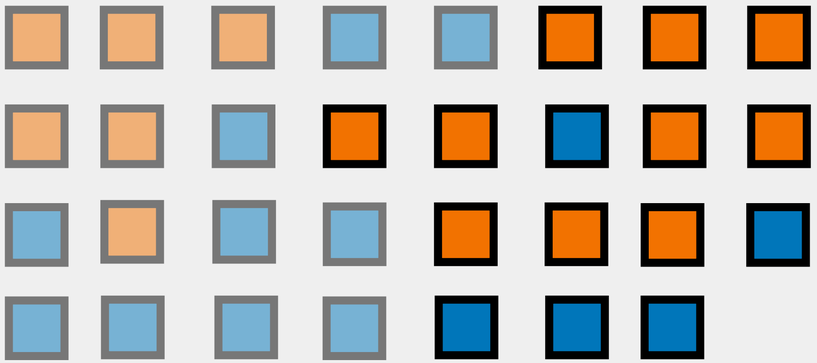


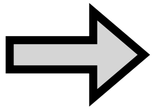
Cluster



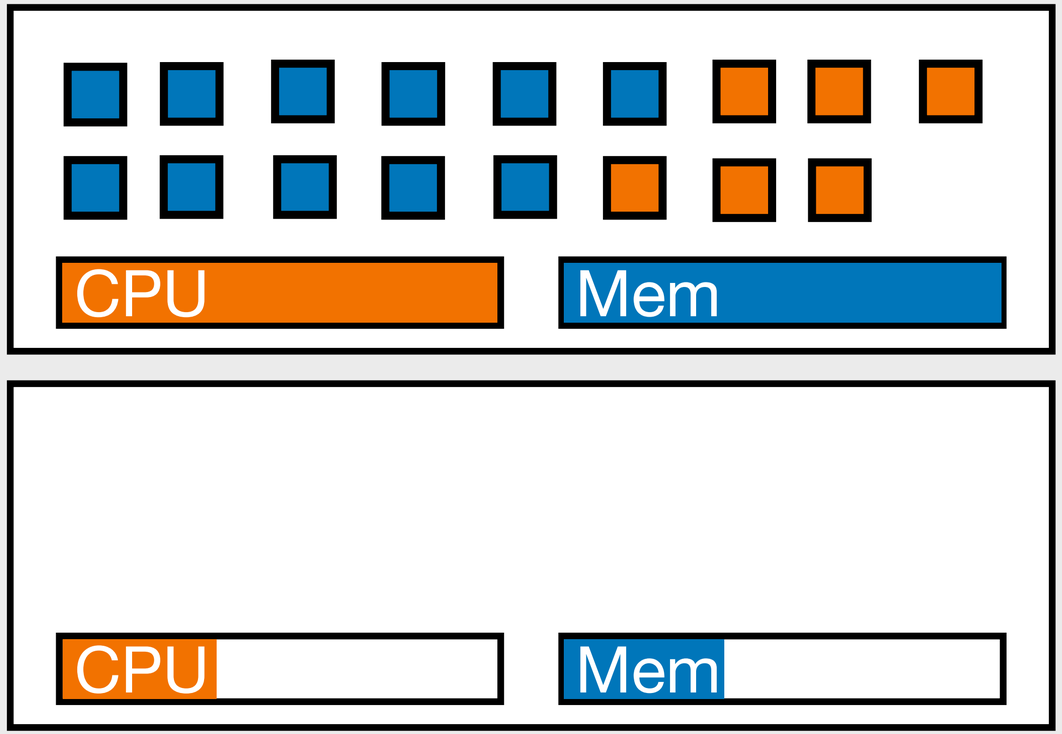
Scheduler that Leaves No Resource Stranded

Application

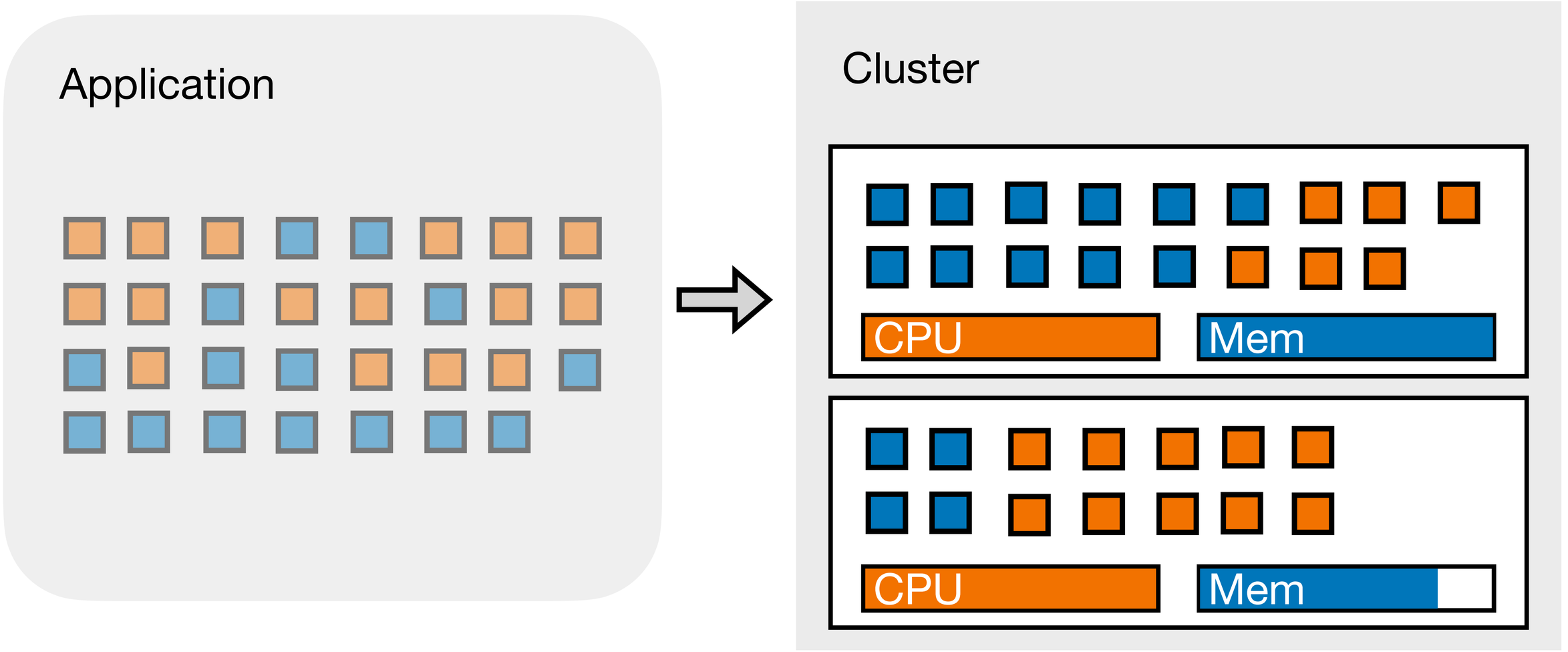




Cluster



Scheduler that Leaves No Resource Stranded



Scheduler that Leaves No Resource Stranded

Key Insight 💡

<u>Fine-grained resource decoupling</u>
unlocks stranded resources.

Our Approach

Unstrand resources without Resource Disaggregation.

Our Approach

Unstrand resources without Resource Disaggregation.

Granular Programming can disaggregate in SW.

Quicksand

A new granular programming framework that unstrands resources

Quicksand Goals

1. Use stranded resources *wherever* and *whenever* available, even briefly (<1s).
2. Support batch and latency-sensitive applications.
3. Easy to adopt and deploy on hardware today.

Challenges and Design Overview

Challenges	***Quicksand's Approach***
Compute and memory are coupled on today's computer architecture	**Resource Proclets**: granular units that *primarily* use one resource

Challenges and Design Overview

Challenges	*Quicksand's Approach*
Compute and memory are coupled on today's computer architecture	**Resource Proclets**: granular units that *primarily* use one resource
App developers think in high-level logic, not in terms of resource decoupling	**High-level frameworks** that *automatically* decompose into resource proclets

Challenges and Design Overview

Challenges	*Quicksand's Approach*
Compute and memory are coupled on today's computer architecture	**Resource Proclets**: granular units that *primarily* use one resource
App developers think in high-level logic, not in terms of resource decoupling	**High-level frameworks** that *automatically* decompose into resource proclets
Maintaining resource proclets granularity	**Split / merge** resource proclets

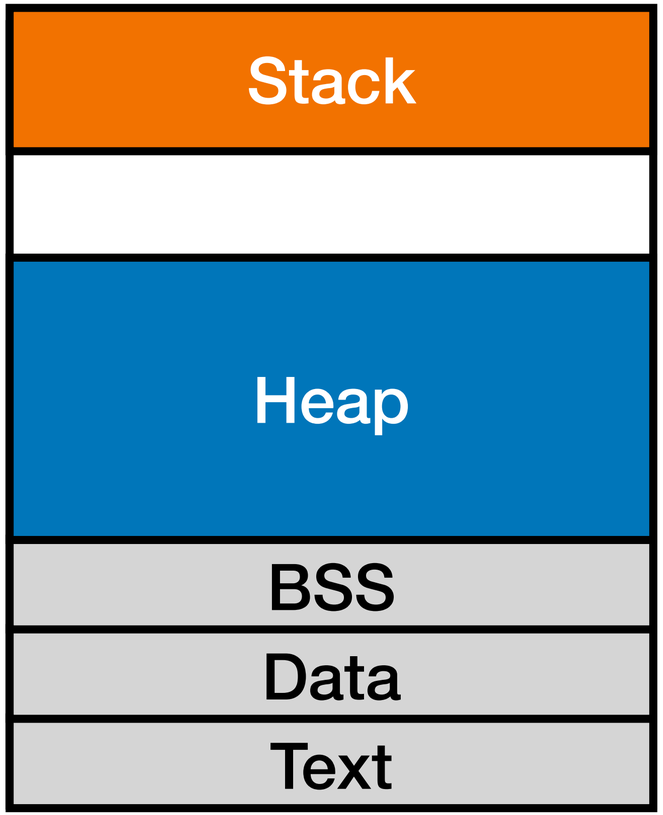
Resource Proclet Background

Decomposing Unix Processes into Proclets

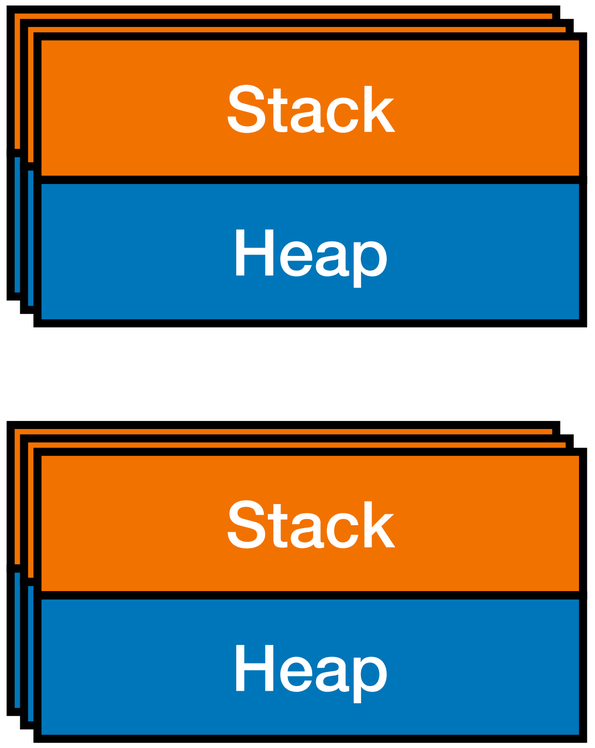
Resource Proclet Background

Decomposing Unix Processes into Proclets

Unix Process

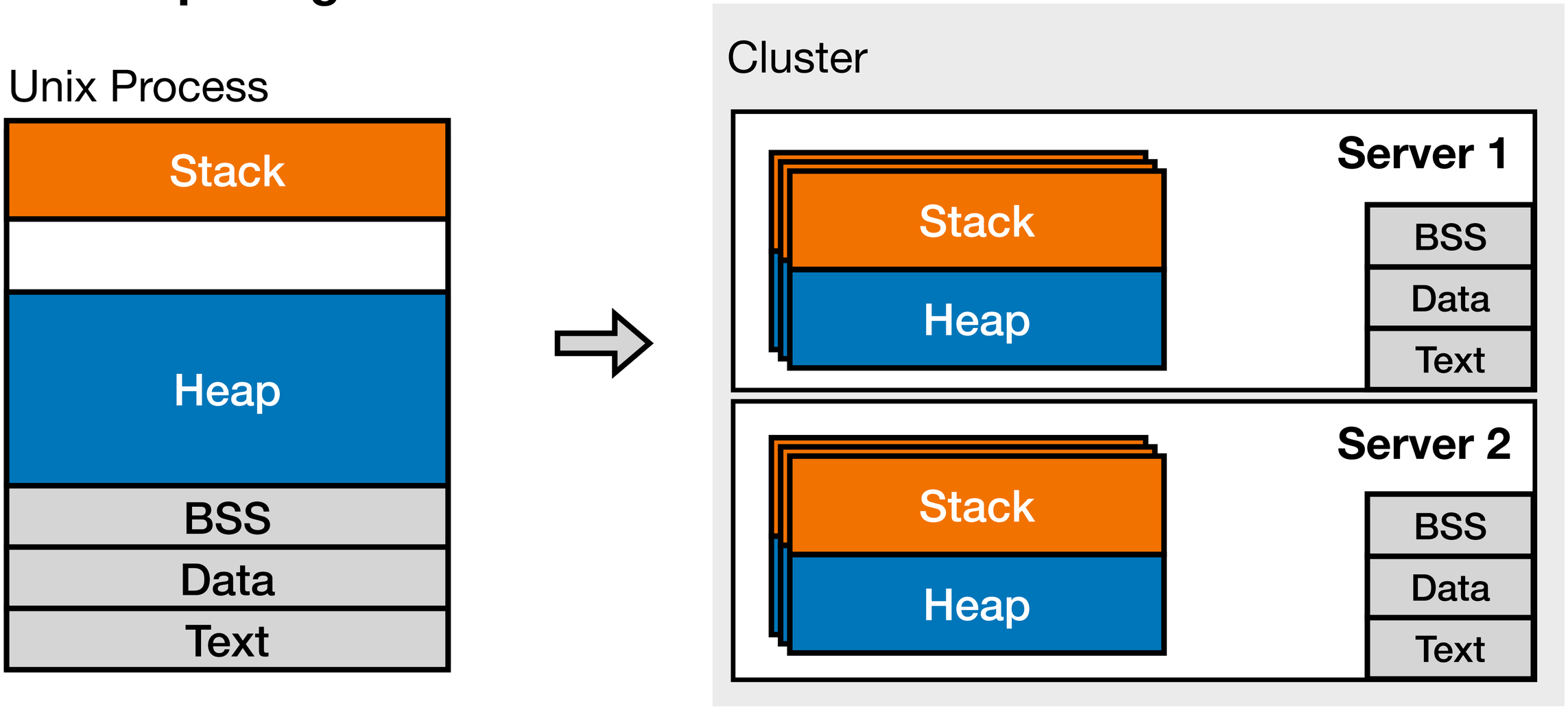


Nu Proclets [NSDI ’23]



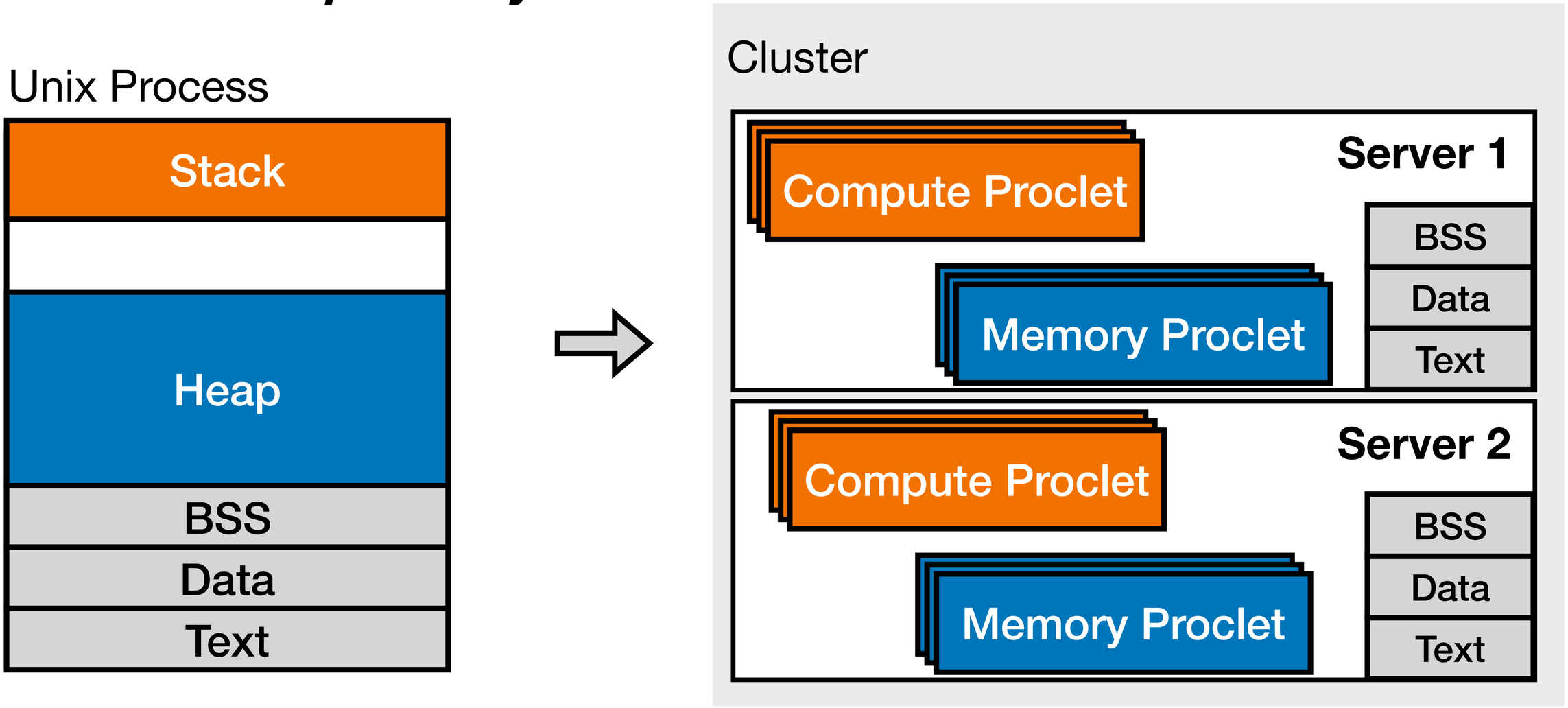
Resource Proclet Background

Decomposing Unix Processes into Proclets



Resource Proclets

Proclets that *primarily* use one resource



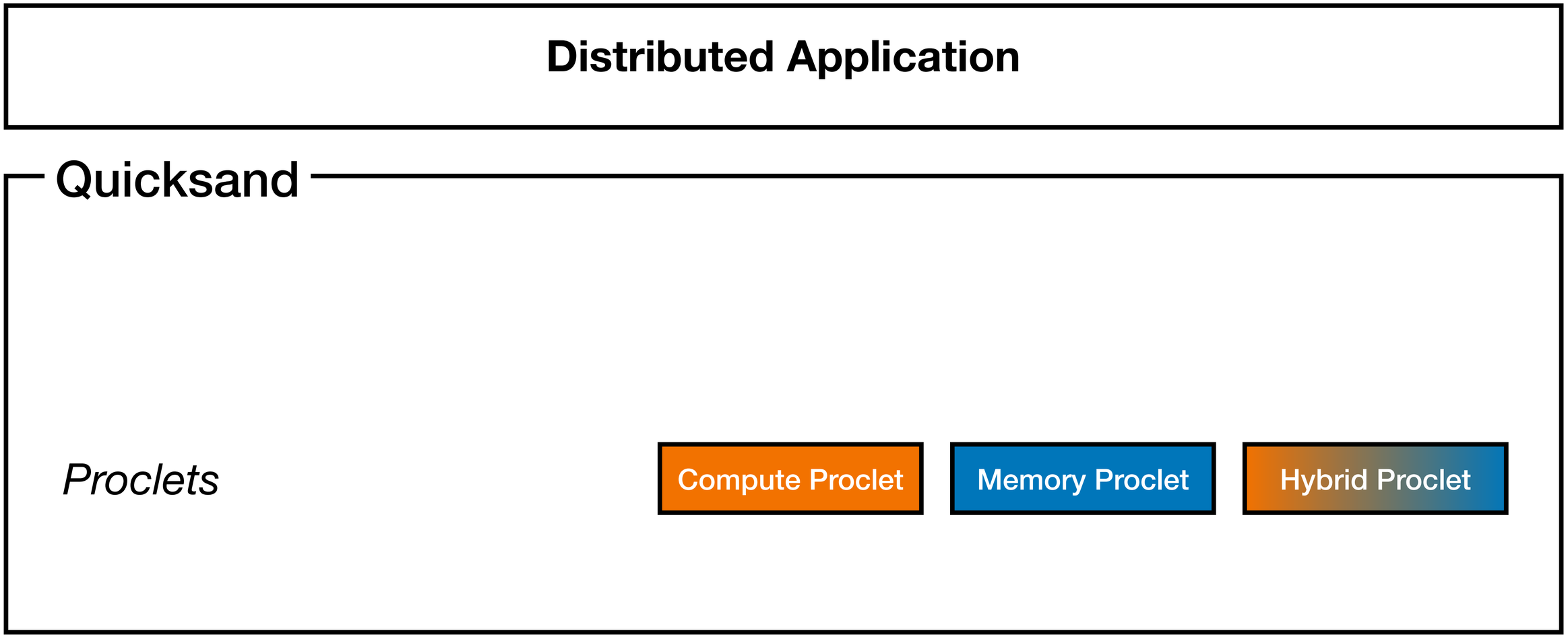
Resource Proclet Resource Usage

Resource Proclets	Compute Proclet	Memory Proclet
CPU	**1 core.** Ephemeral or long-running compute.	Cheap memory operations only.
Memory	Thread stack. Transient allocations (≈ KBs).	**Small heap regions (MBs).**

Quicksand Proclet Types

- **Resource Proclets (RP)**: Units that *primarily* use one resource.
- **Hybrid Proclet**: pair compute and memory proclets temporarily.
  - For memory access-intensive logic.
  - Re-introduces stranding.

Quicksand Architecture



Quicksand Architecture

Distributed Application

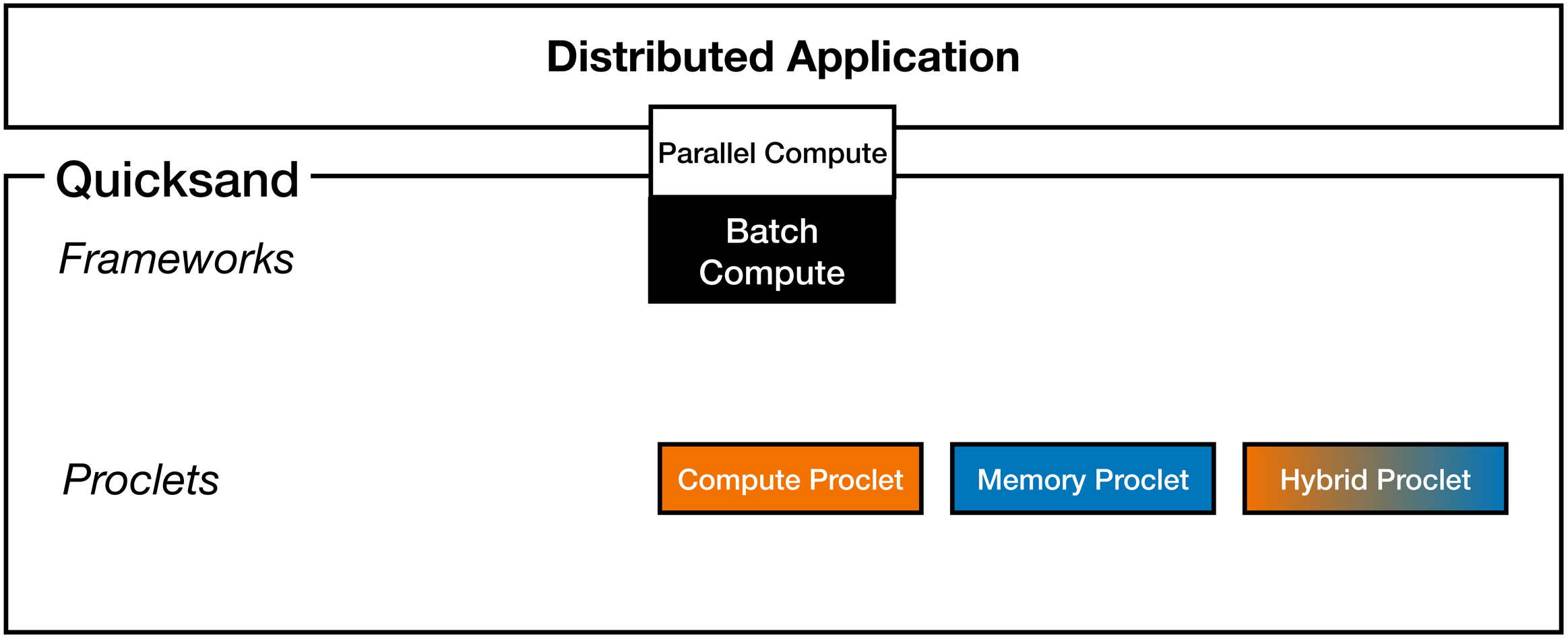
Quicksand

Frameworks

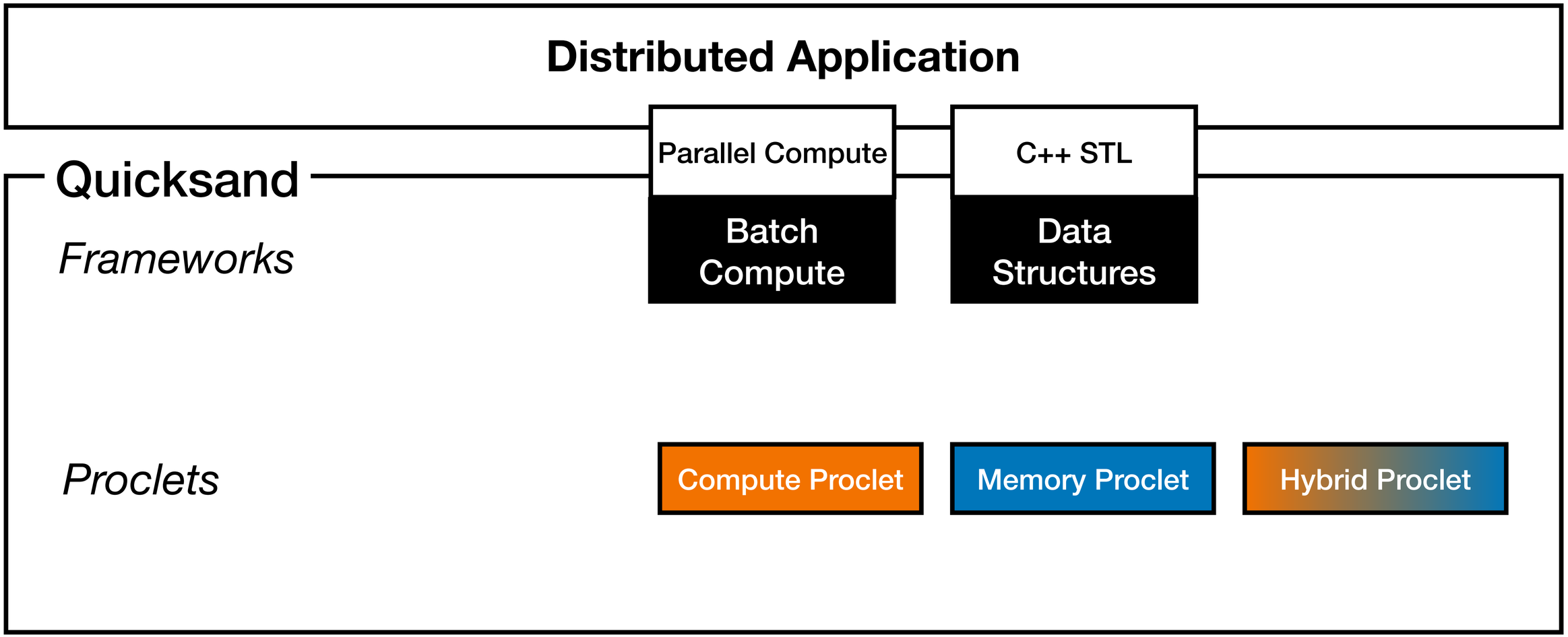
Proclets

Compute Proclet | Memory Proclet | Hybrid Proclet

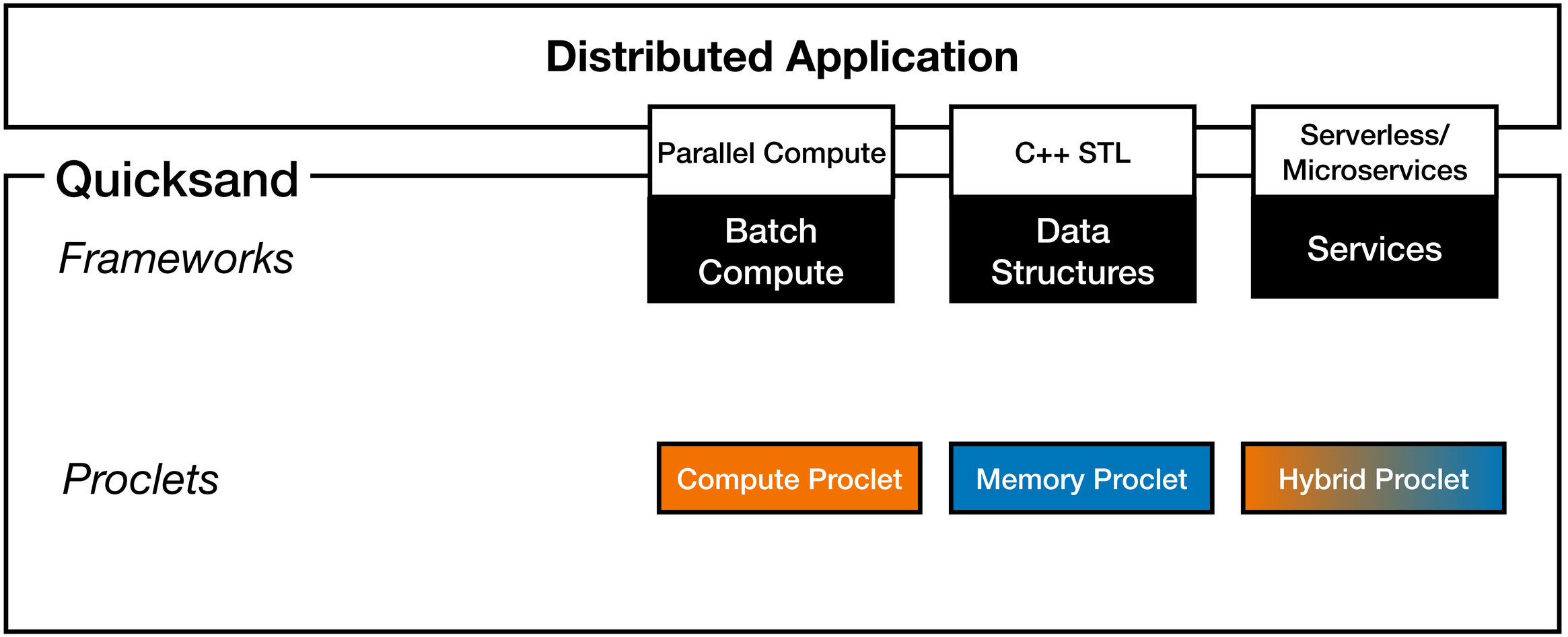
Quicksand Architecture



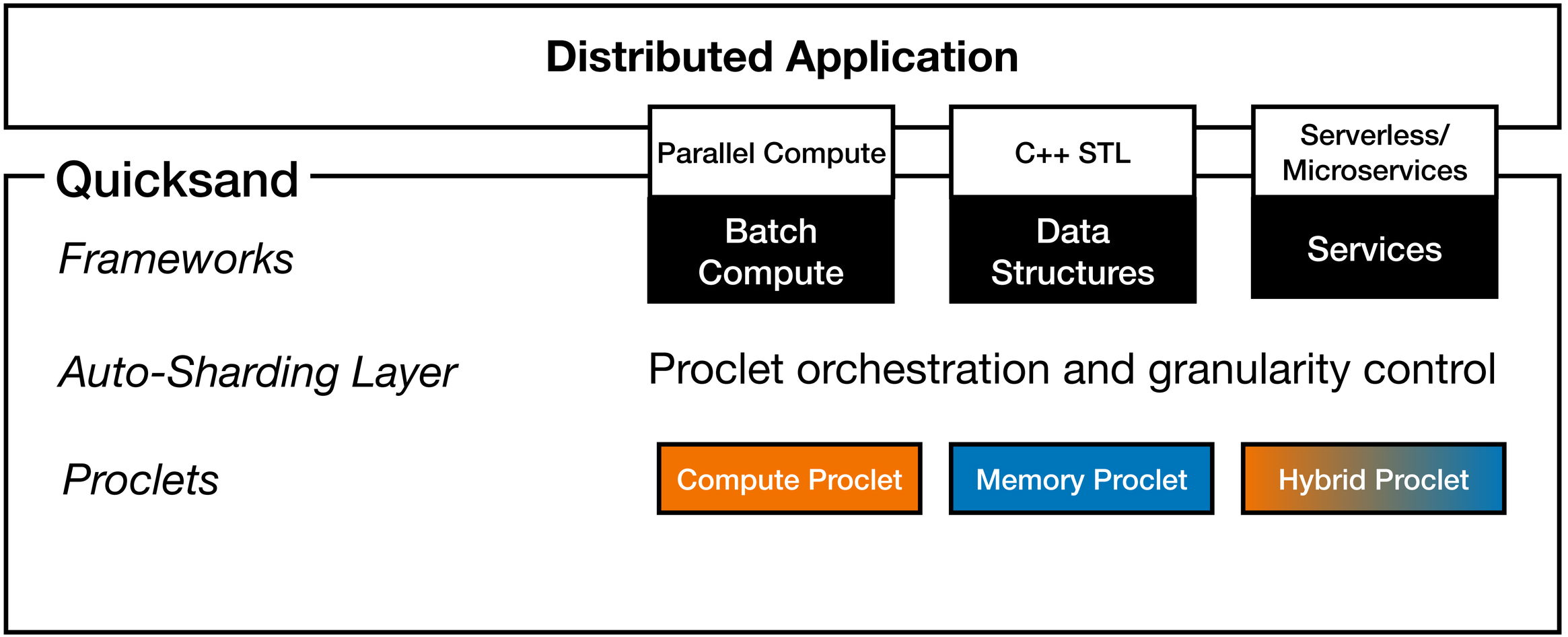
Quicksand Architecture



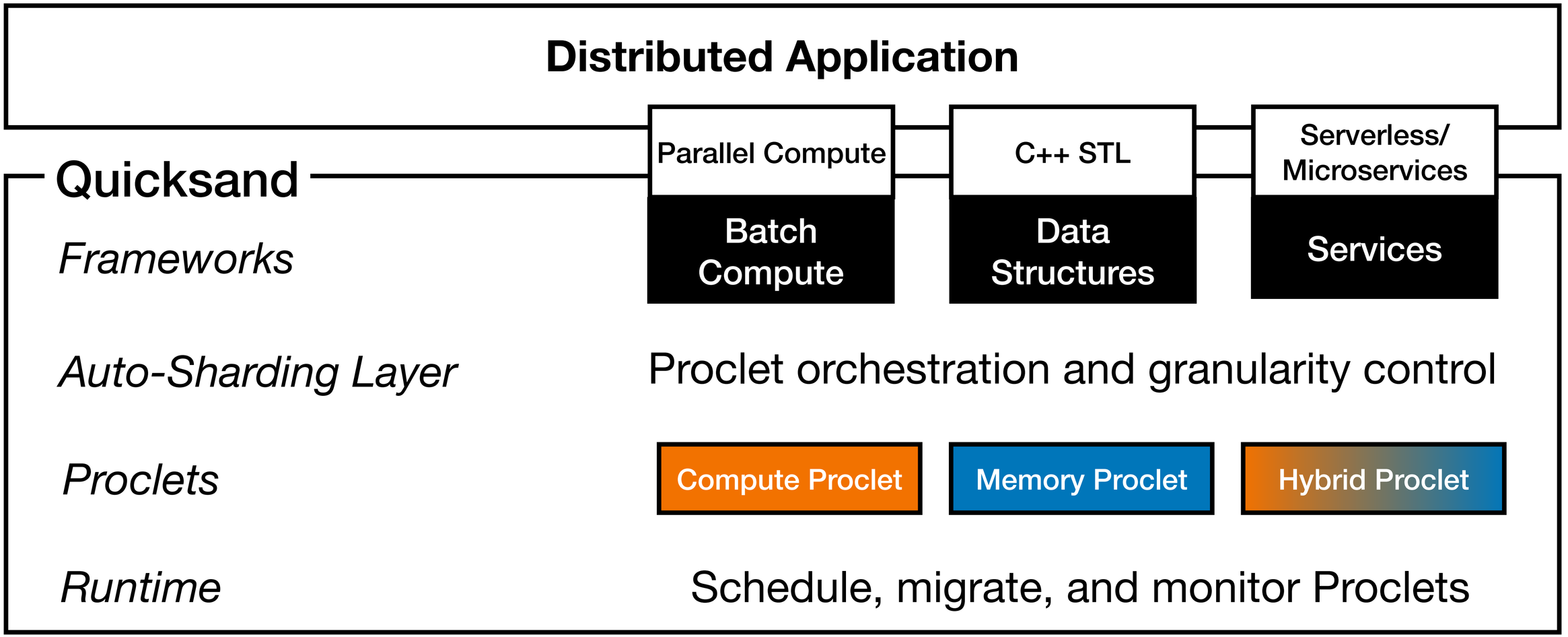
Quicksand Architecture



Quicksand Architecture



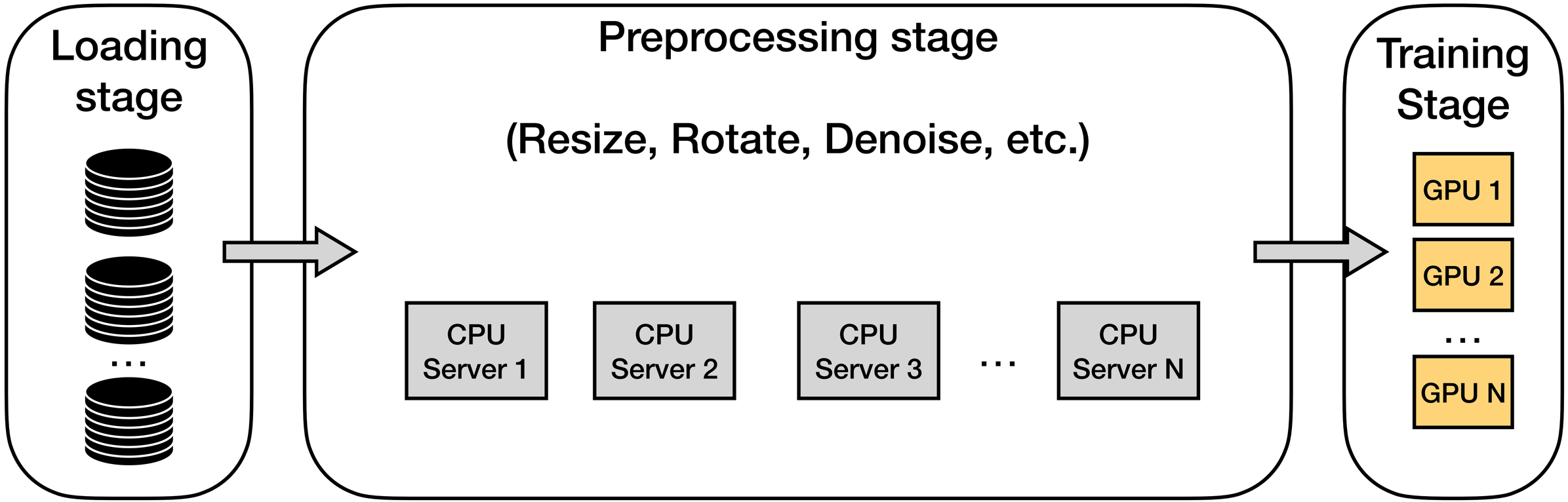
Quicksand Architecture



Quicksand in Action:
Training Data Pre-processing

Pipeline Overview

Example: training data pre-processing



Quicksand Code

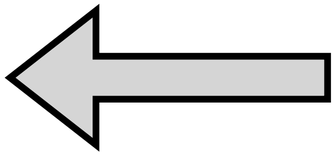
Expressing data pre-processing with high-level frameworks

```
qs::ShardedVector<Img> imgs = load_images();
qs::ShardedQueue<Img> queue;
qs::ForAll(qs::seal(imgs), [queue](Img img){
  Img processed_img = process(img);
  queue.push(processed_img);
});
```

Quicksand Code

Expressing data pre-processing with high-level frameworks

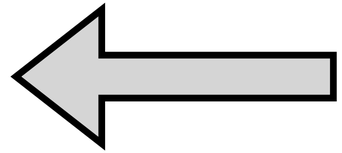
```
qs::ShardedVector<Img> imgs = load_images();
qs::ShardedQueue<Img> queue;
qs::ForAll(qs::seal(imgs), [queue](Img img){
  Img processed_img = process(img);
  queue.push(processed_img);
});
```



Quicksand Code

Expressing data pre-processing with high-level frameworks

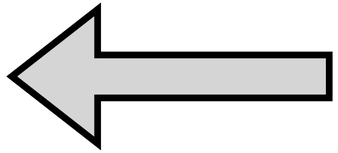
```
qs::ShardedVector<Img> imgs = load_images();
qs::ShardedQueue<Img> queue;
qs::ForAll(qs::seal(imgs), [queue](Img img){
  Img processed_img = process(img);
  queue.push(processed_img);
});
```



Quicksand Code

Expressing data pre-processing with high-level frameworks

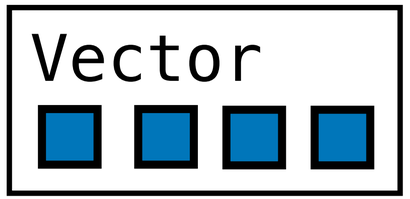
```
qs::ShardedVector<Img> imgs = load_images();
qs::ShardedQueue<Img> queue;
qs::ForAll(qs::seal(imgs), [queue](Img img){
  Img processed_img = process(img);
  queue.push(processed_img);
});
```

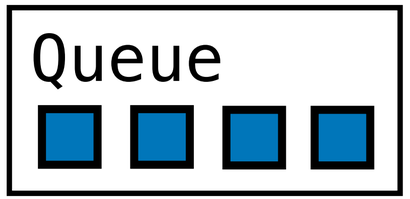


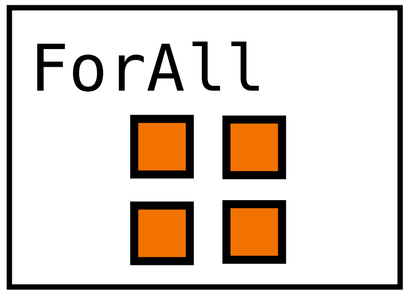
Quicksand Code

Expressing data pre-processing with high-level frameworks

```
qs::ShardedVector<Img> imgs = load_images();
qs::ShardedQueue<Img> queue;
qs::ForAll(qs::seal(imgs), [queue](Img img){
  Img processed_img = process(img);
  queue.push(processed_img);
});
```

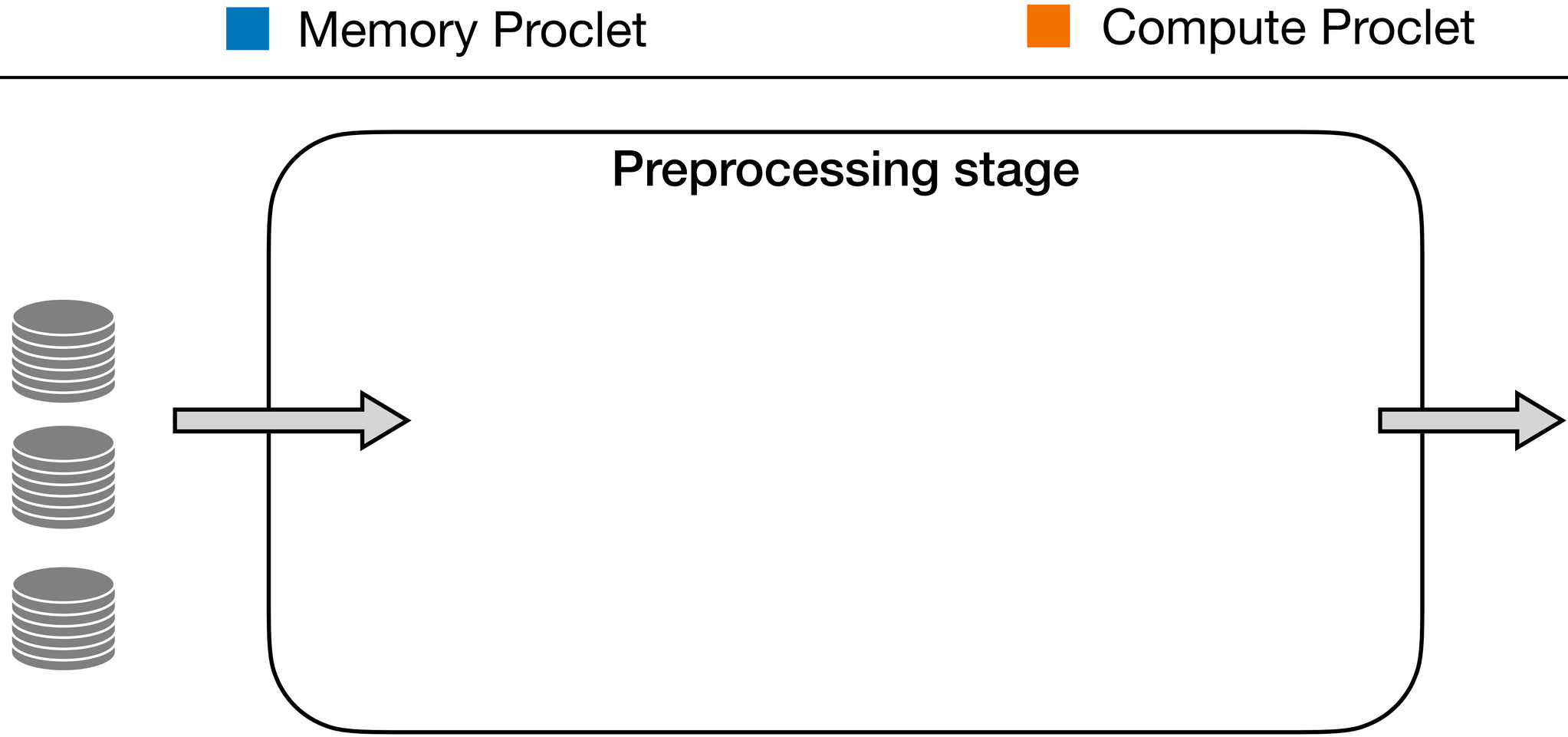






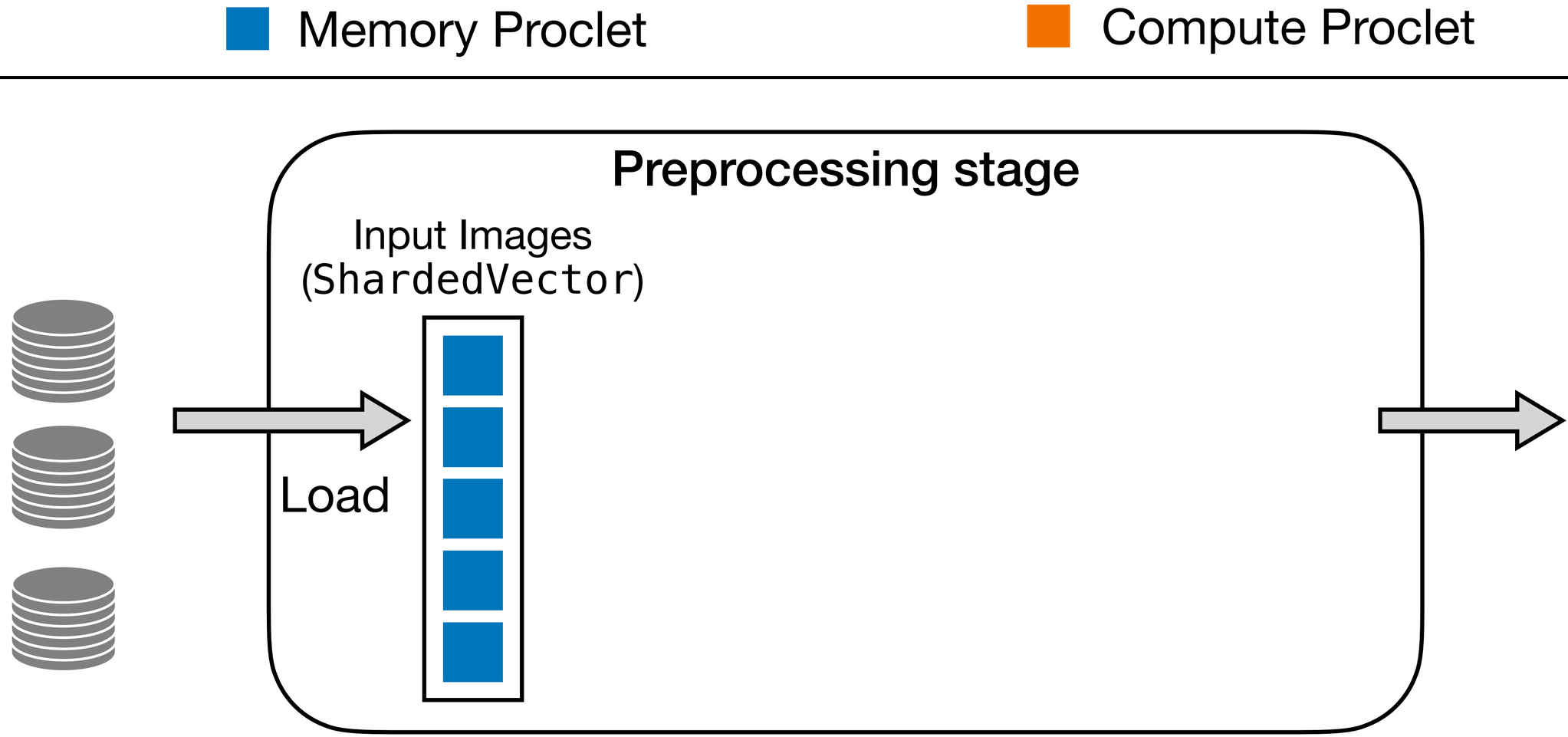
Decompose Pipeline into Resource Proclets

Example: training data pre-processing



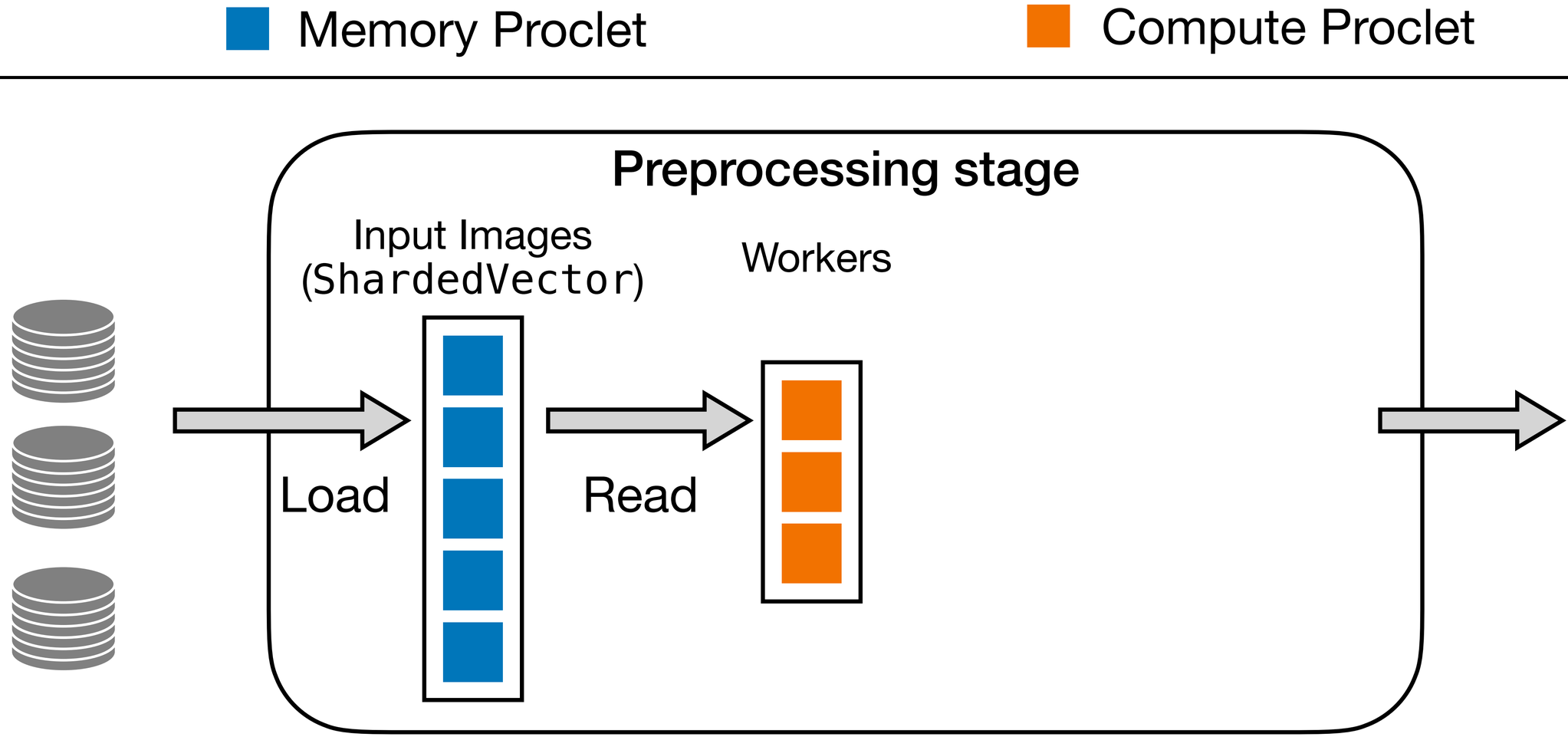
Decompose Pipeline into Resource Proclets

Example: training data pre-processing



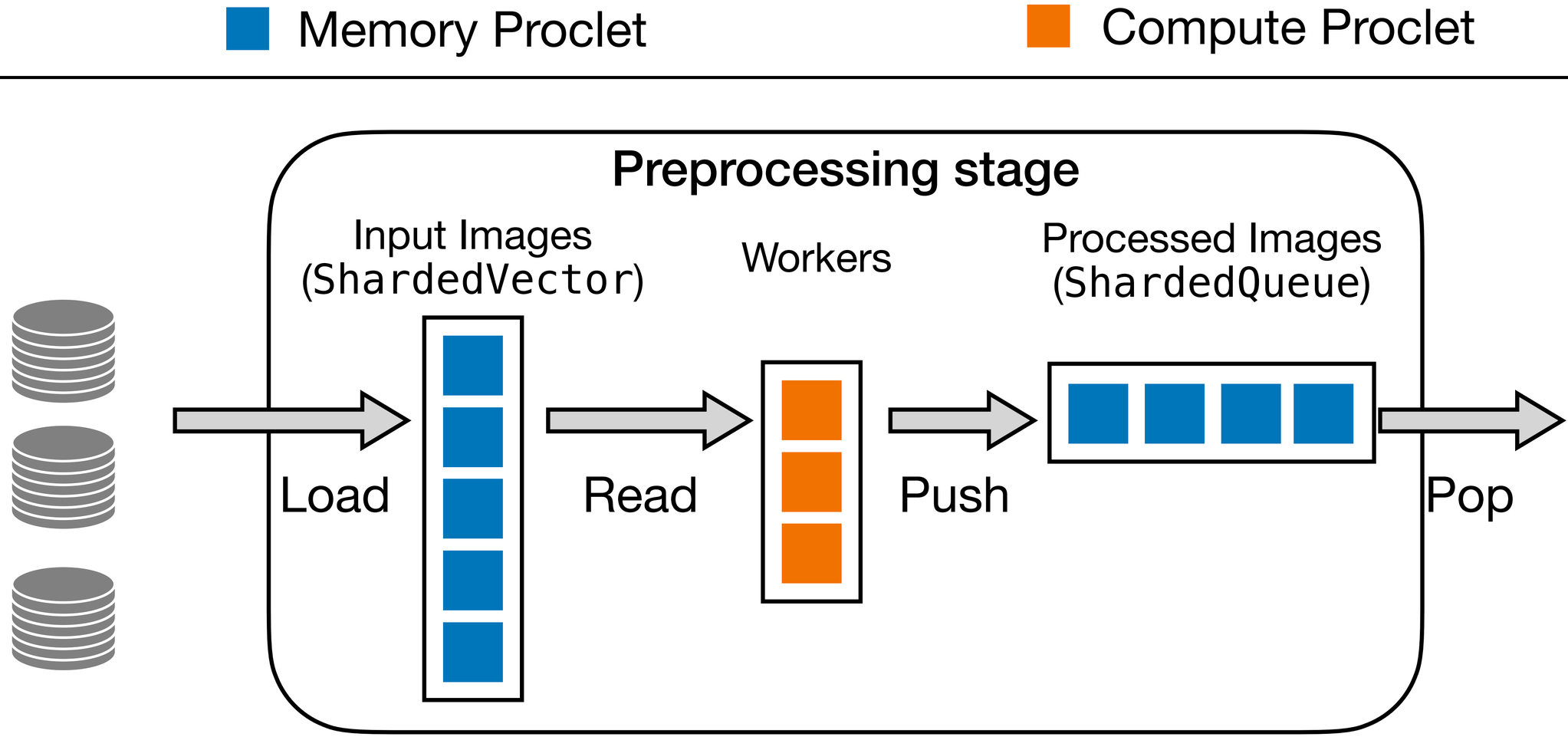
Decompose Pipeline into Resource Proclets

Example: training data pre-processing



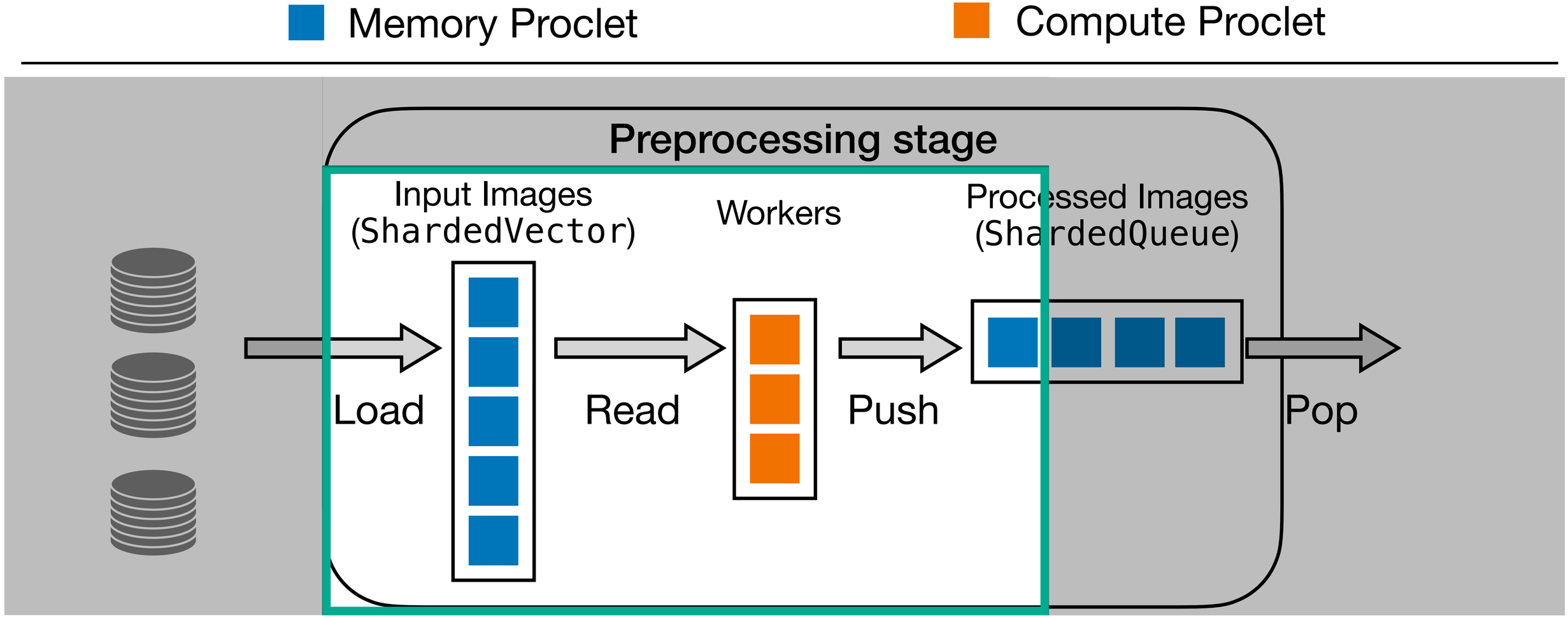
Decompose Pipeline into Resource Proclets

Example: training data pre-processing



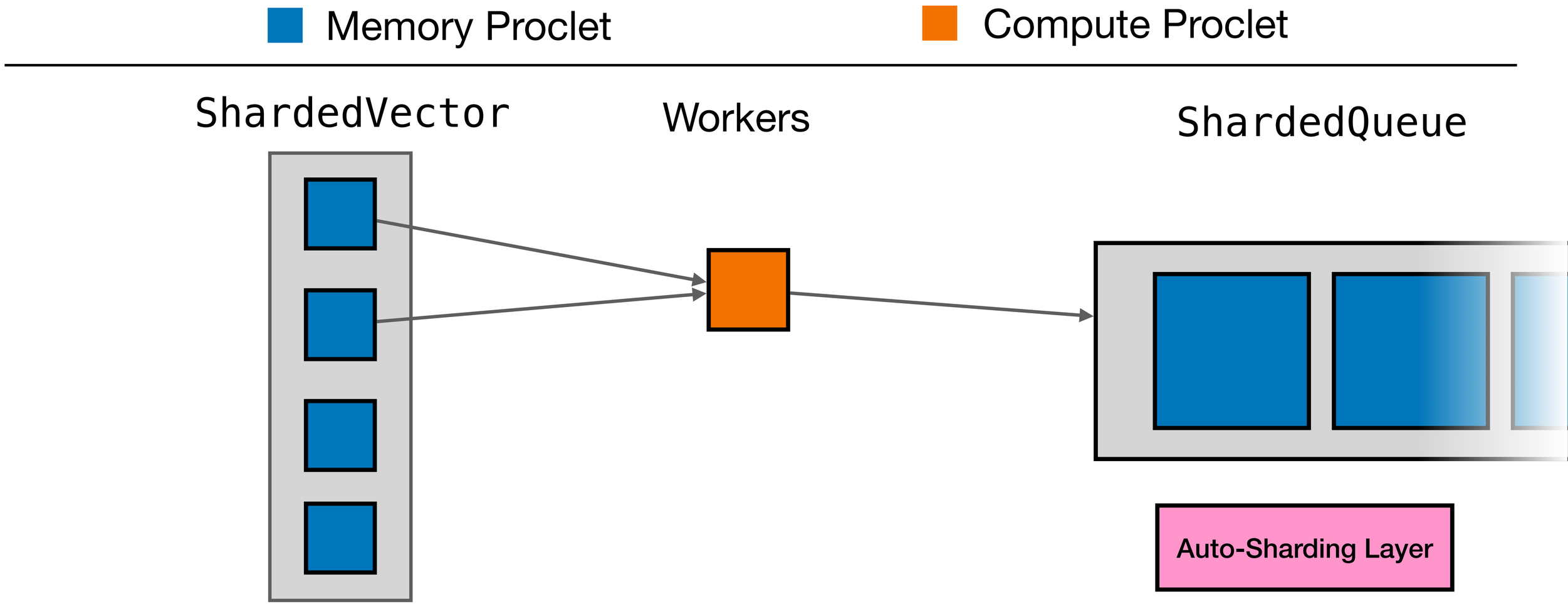
Decompose Pipeline into Resource Proclets

Example: training data pre-processing



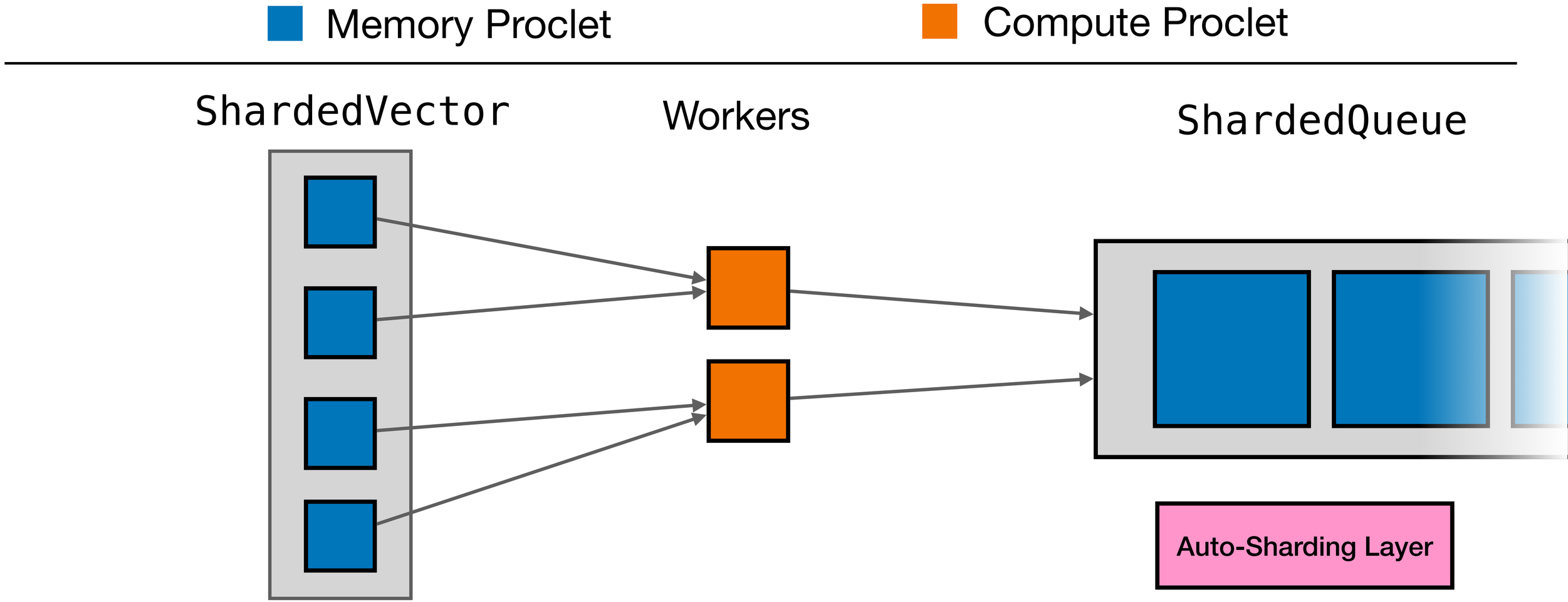
Spawning Compute Proclets

```
qs::ForAll(data, func)
```

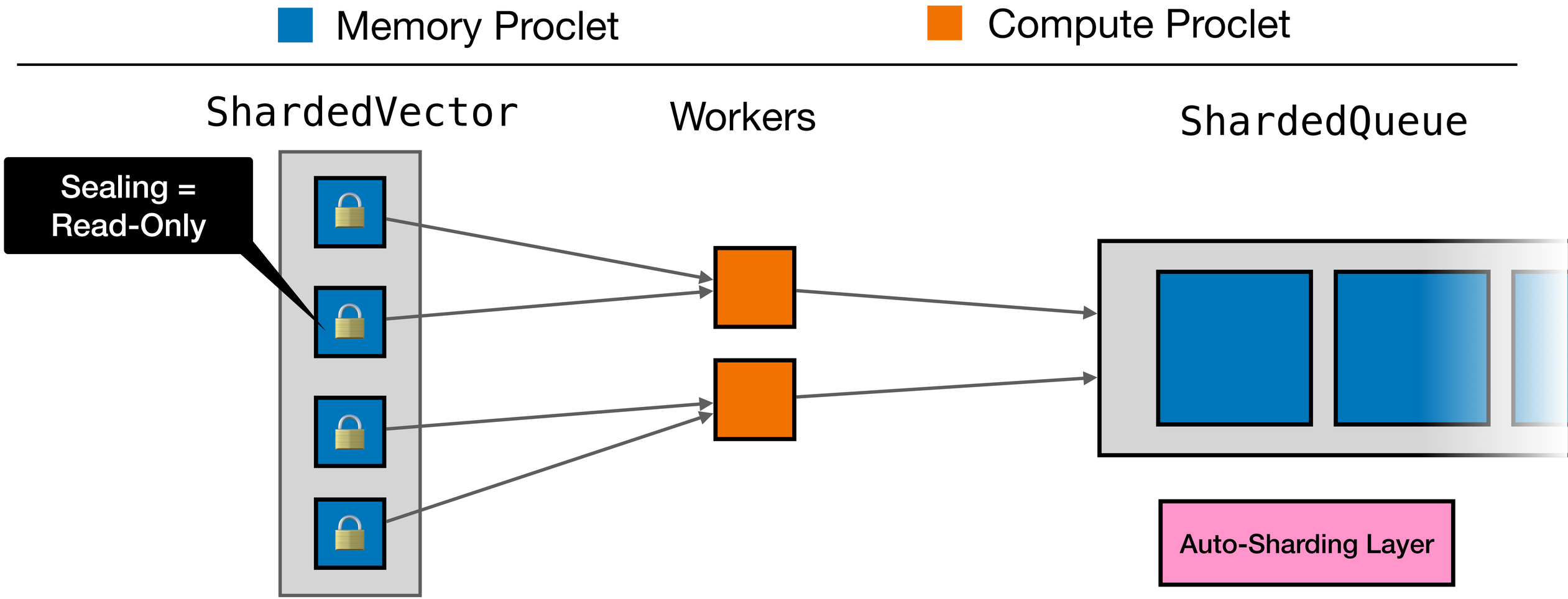


Spawning Compute Proclets

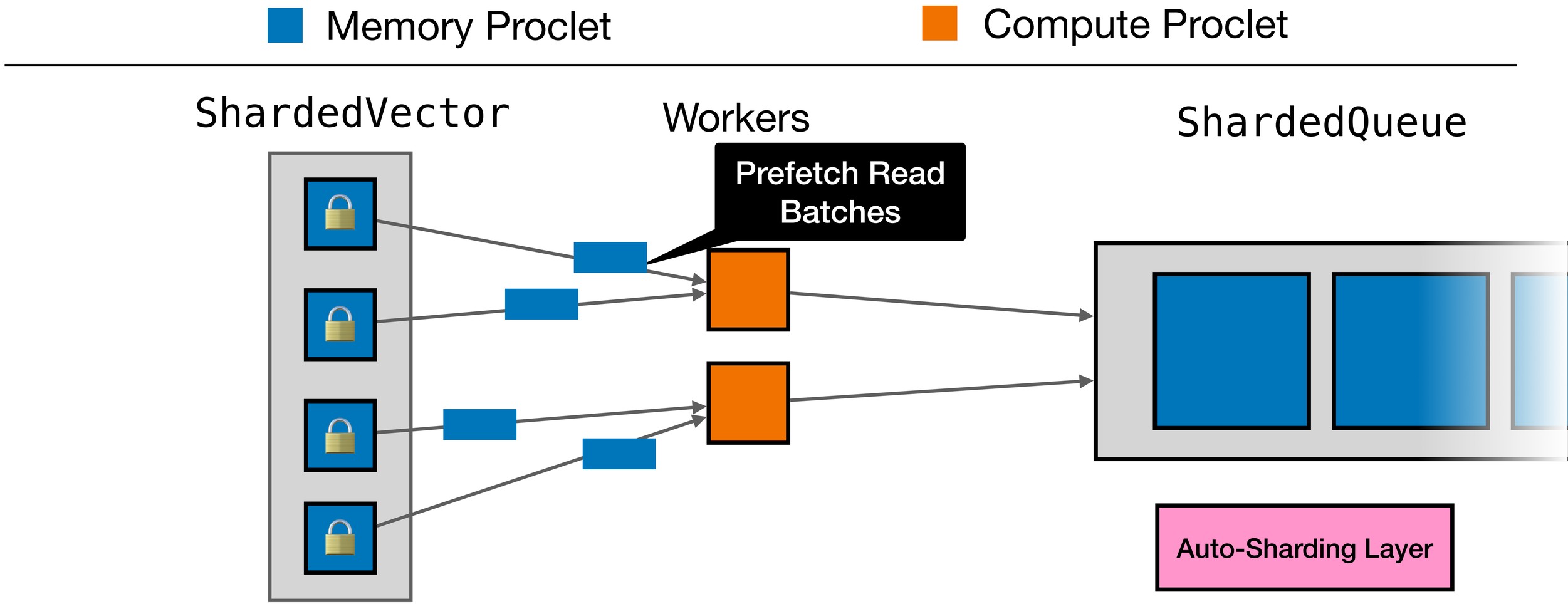
To use all available compute



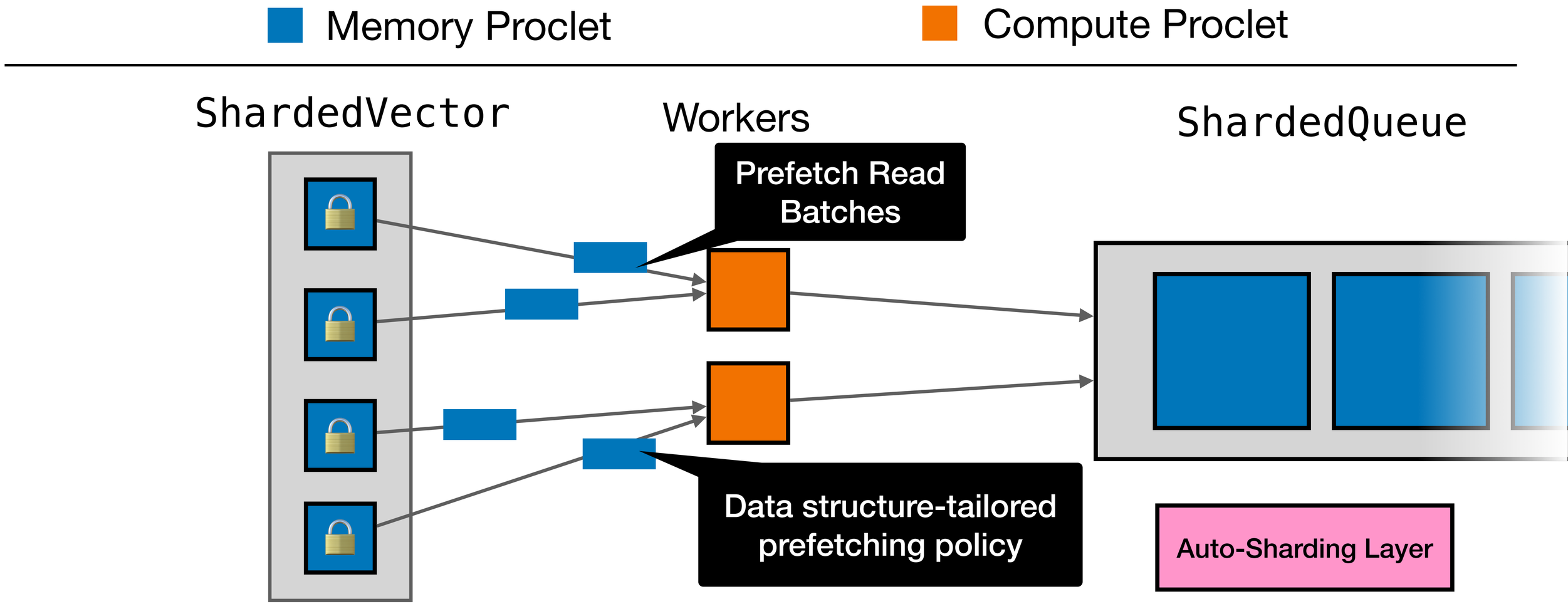
Sealing Memory Proclets



Latency Hiding by Prefetching

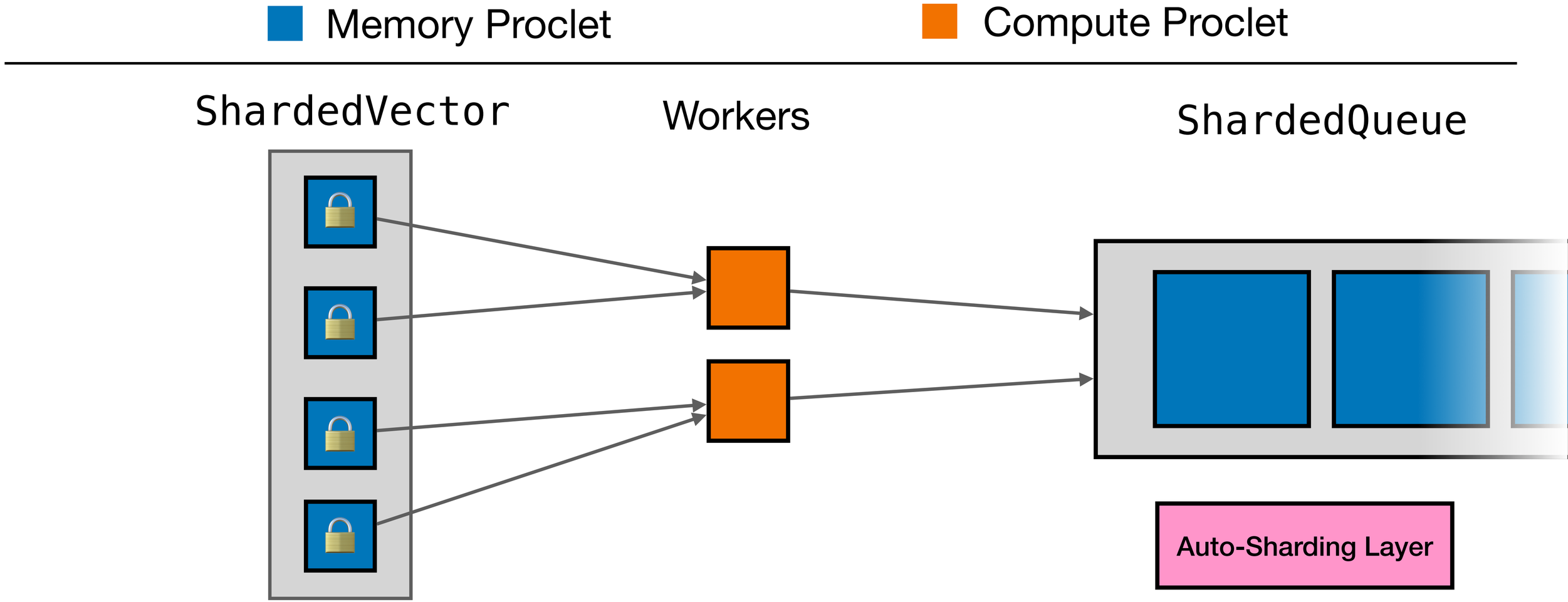


SW-Defined, Semantics-Informed Prefetching



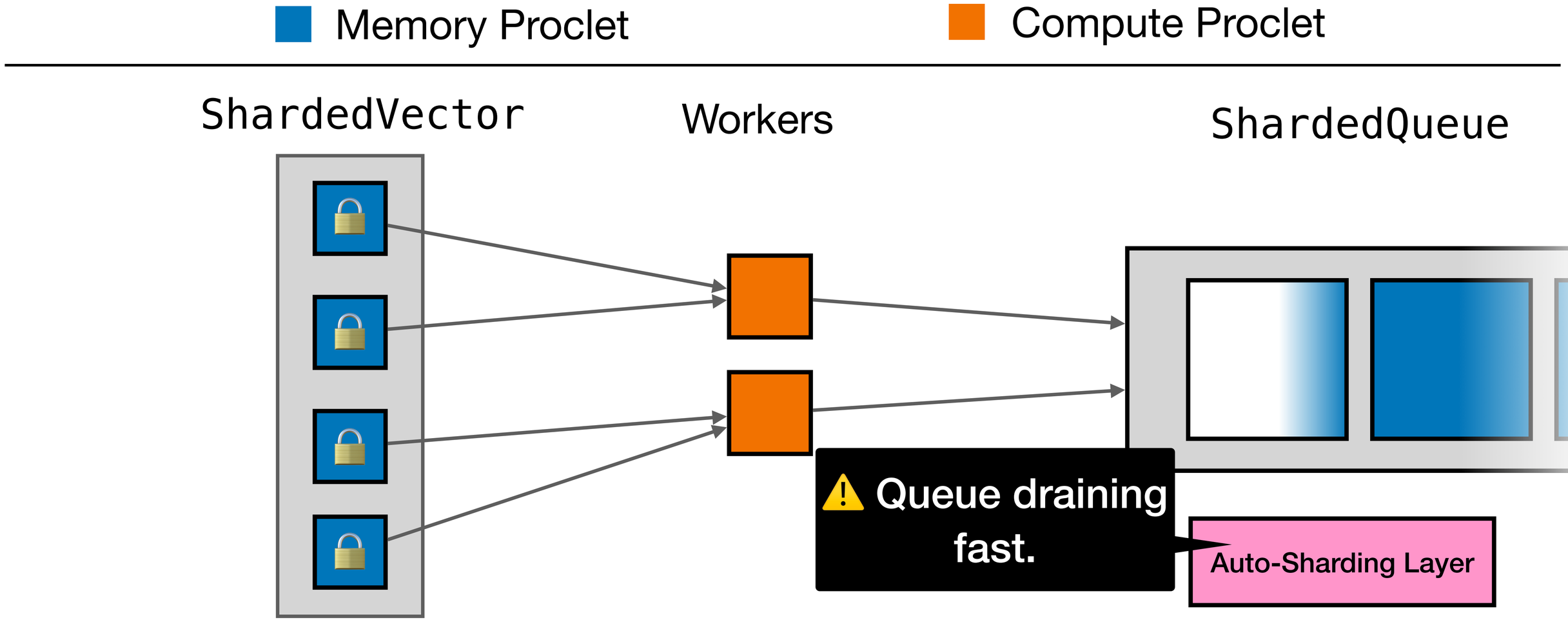
Adapting to Load Changes

Split / Merge Compute Proclets



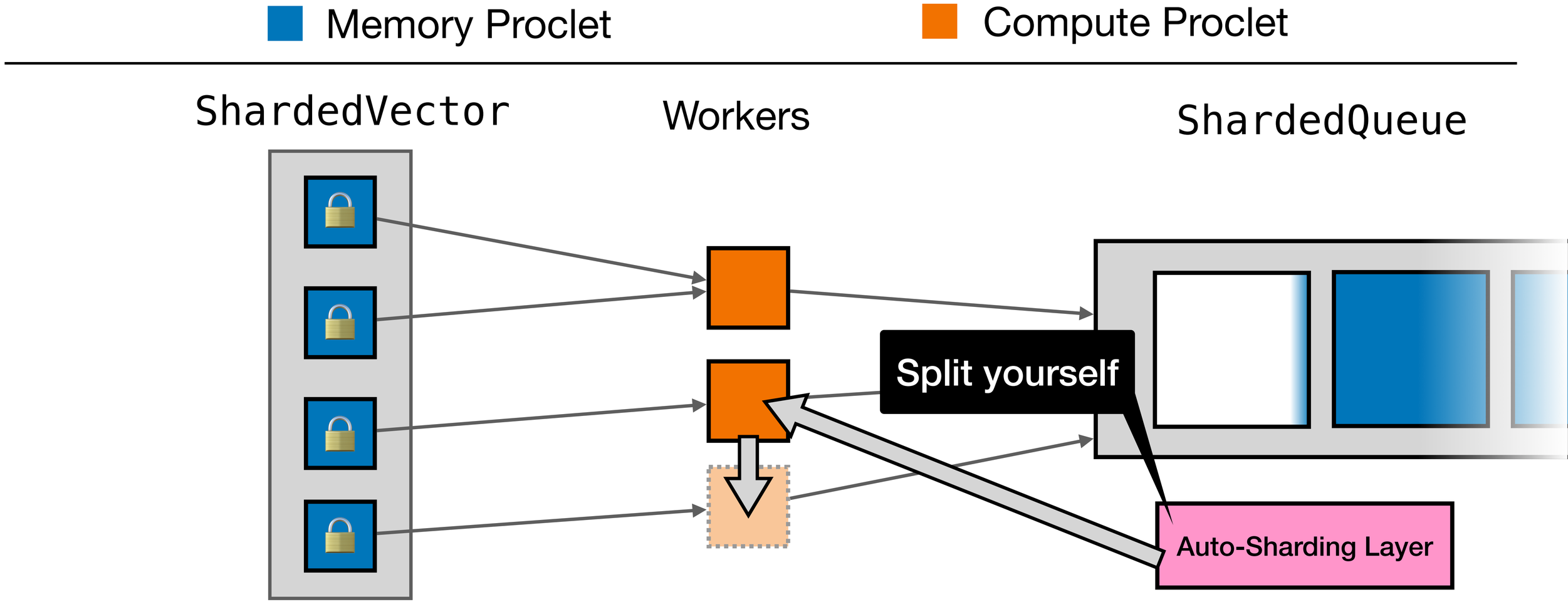
Quicksand Can Monitor App-Level Signals

Split / Merge Compute Proclets



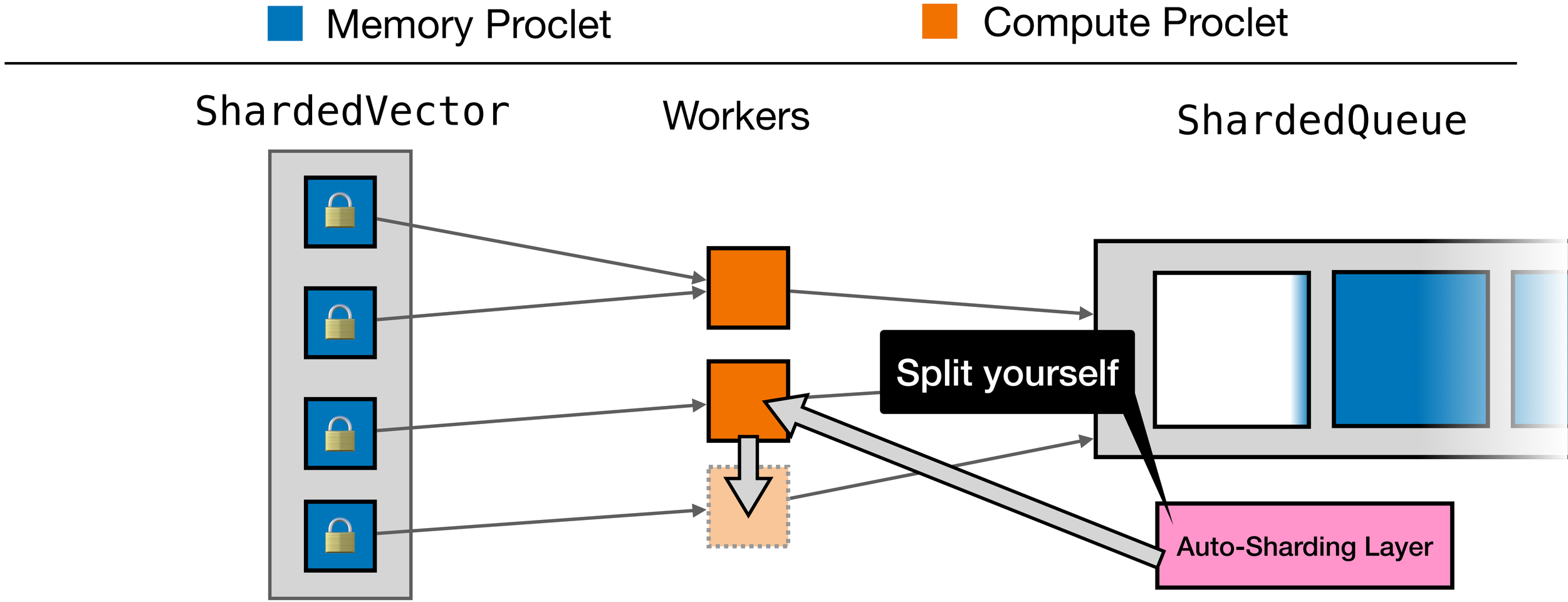
Split Compute Proclet to Increase Parallelism

Split / Merge Compute Proclets



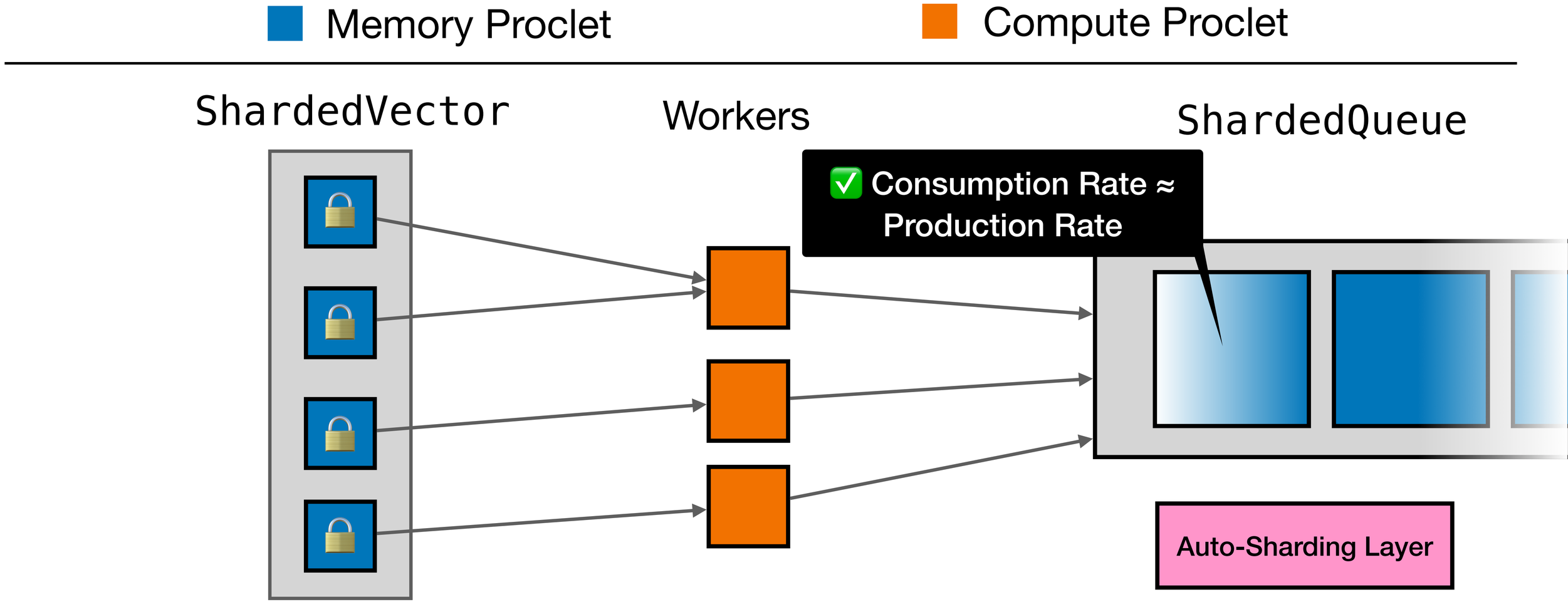
Fine-Grained Work Assignment Enables Splitting

Split / Merge Compute Proclets



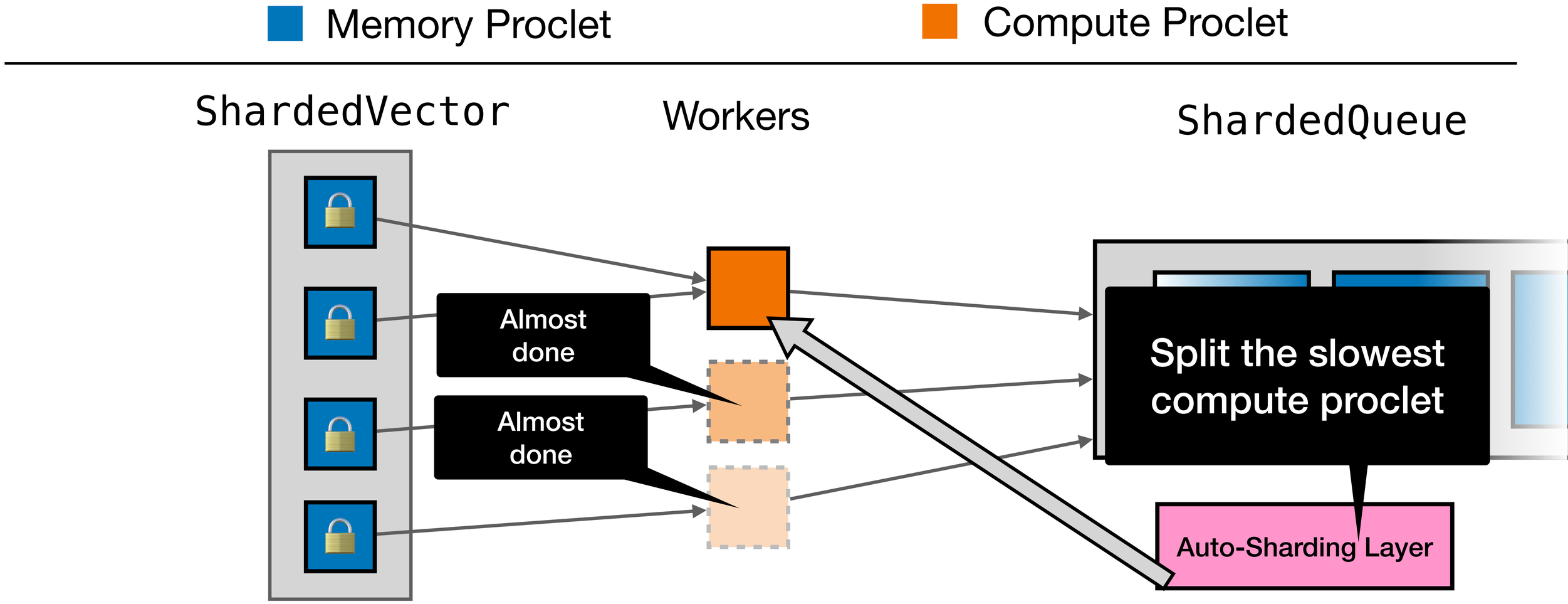
Control Loop for Signal Monitoring

Split / Merge Compute Proclets



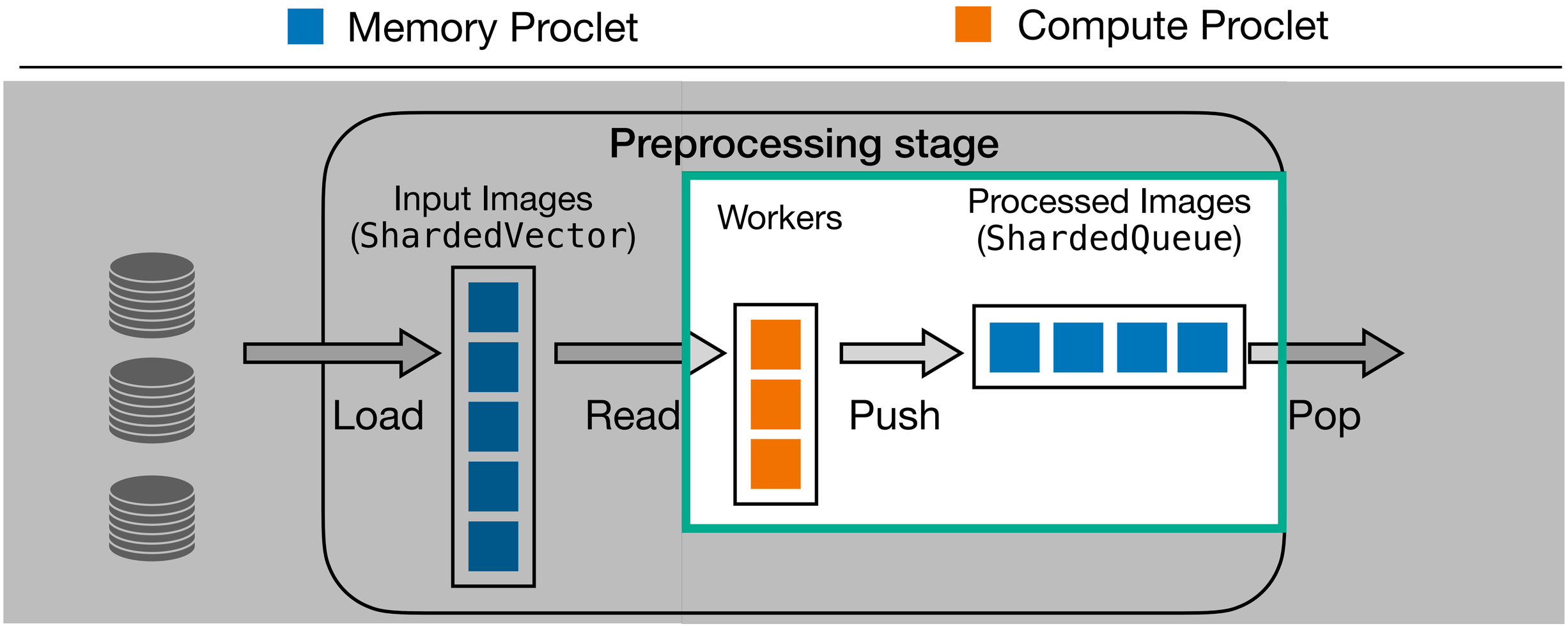
Straggler Mitigation via Splitting

Split / Merge Compute Proclets



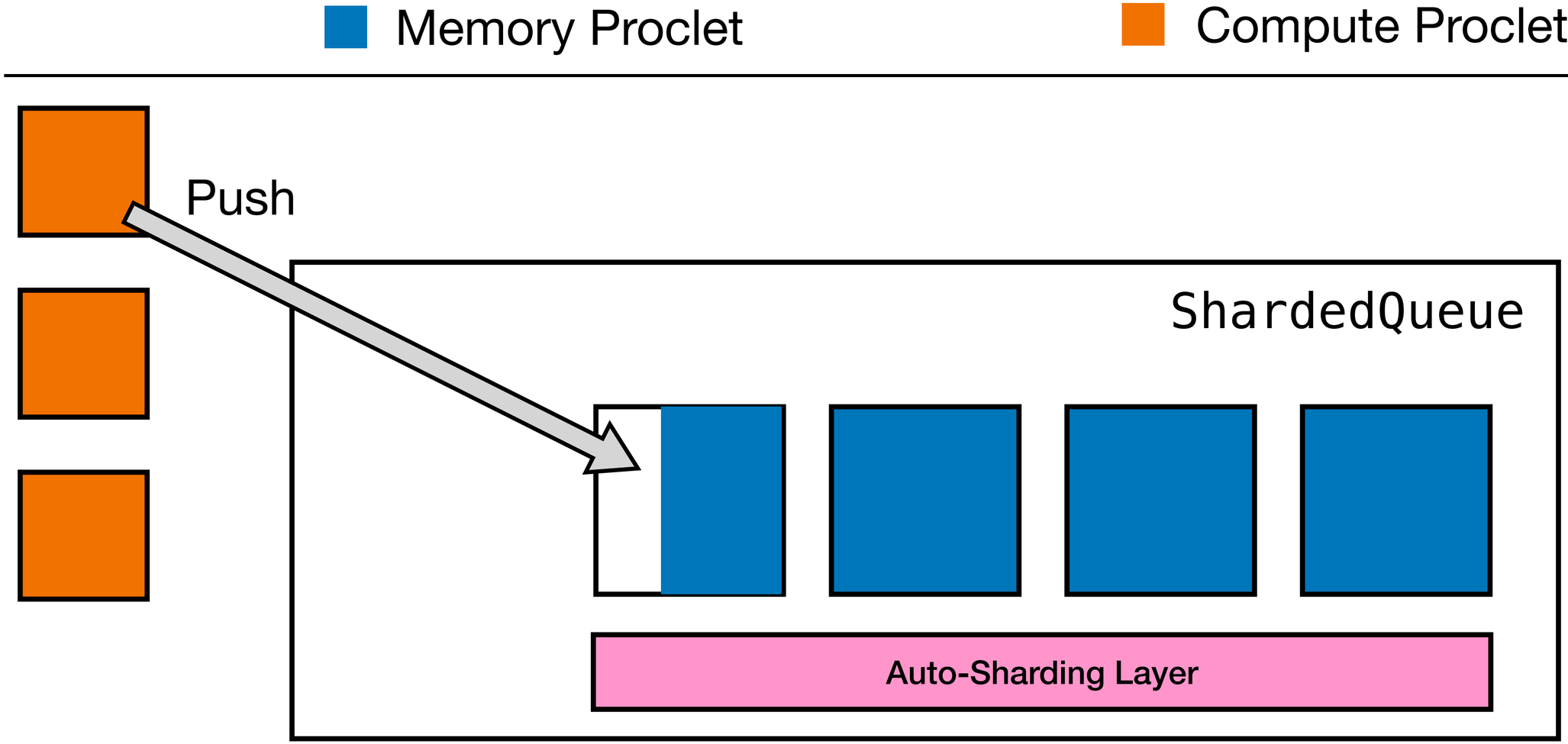
Pipeline Phase Two

Example: training data pre-processing



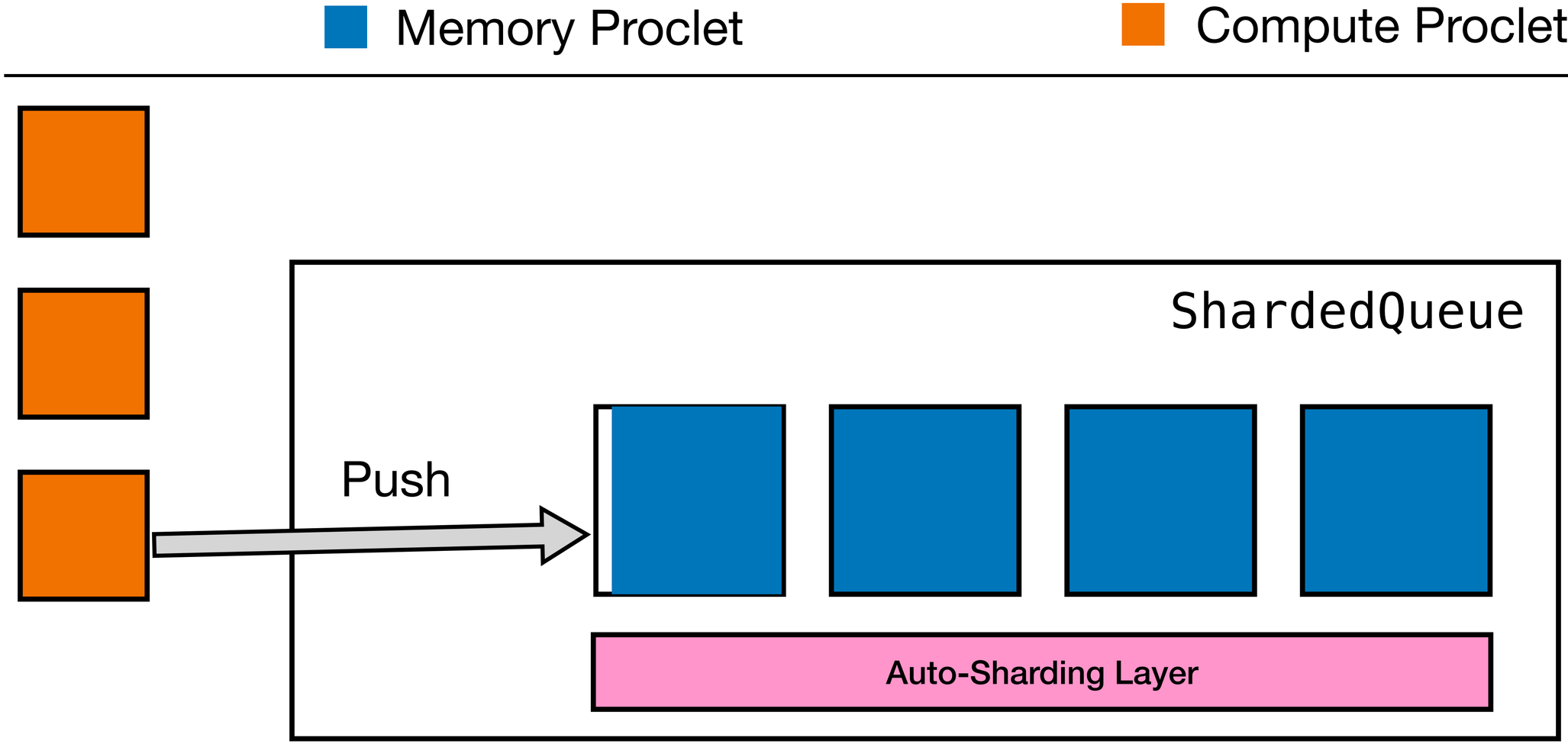
Proclet Memory Usage Varies Over Time

Split / Merge Memory Proclets



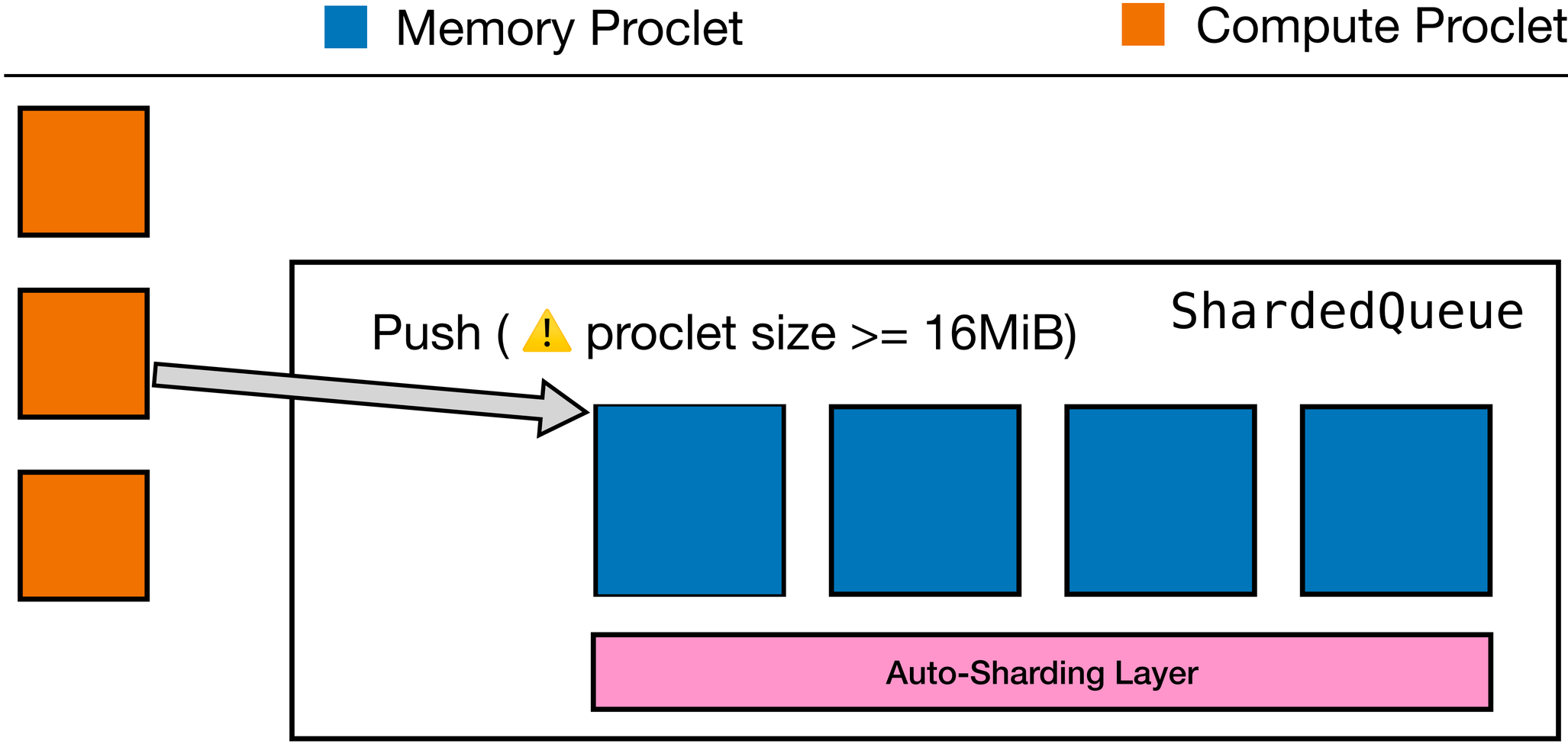
Proclet Memory Usage Varies Over Time

Split / Merge Memory Proclets



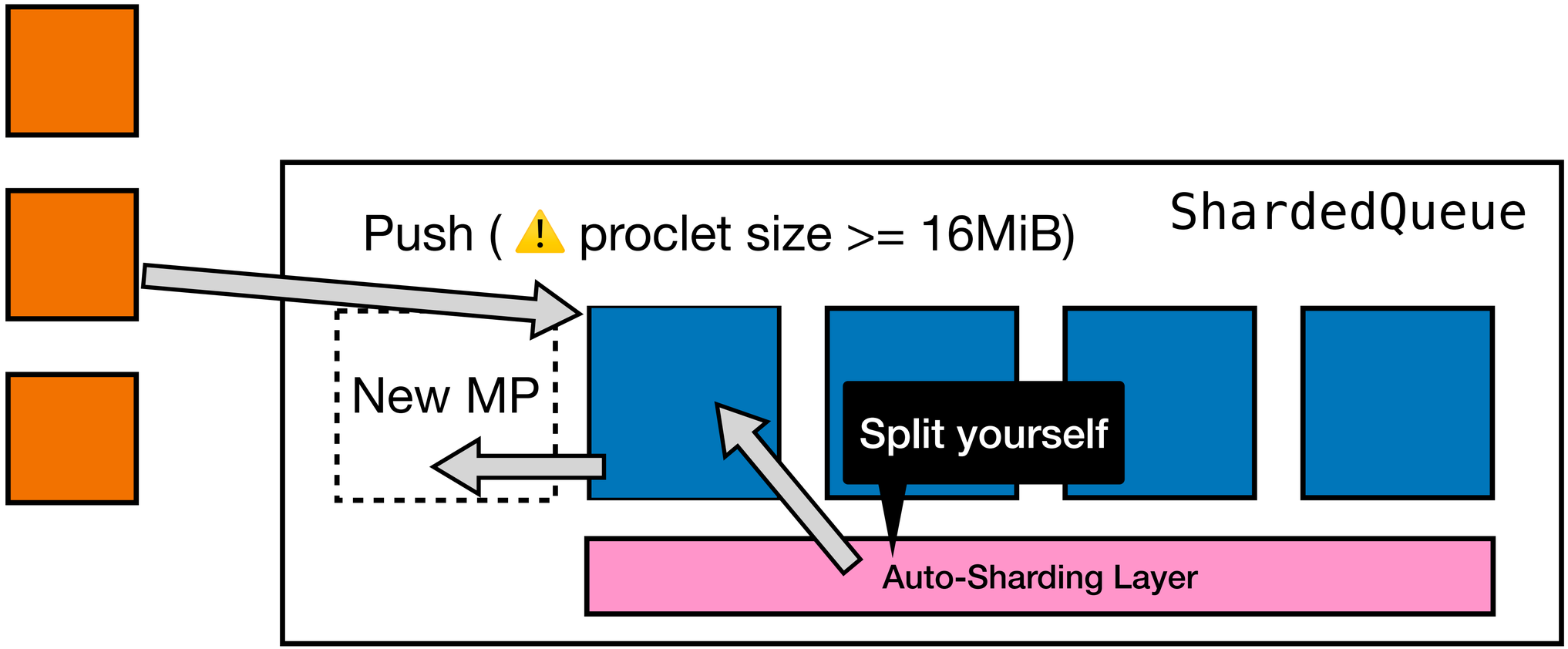
Quicksand Enforces Proclet Memory Usage

Split / Merge Memory Proclets



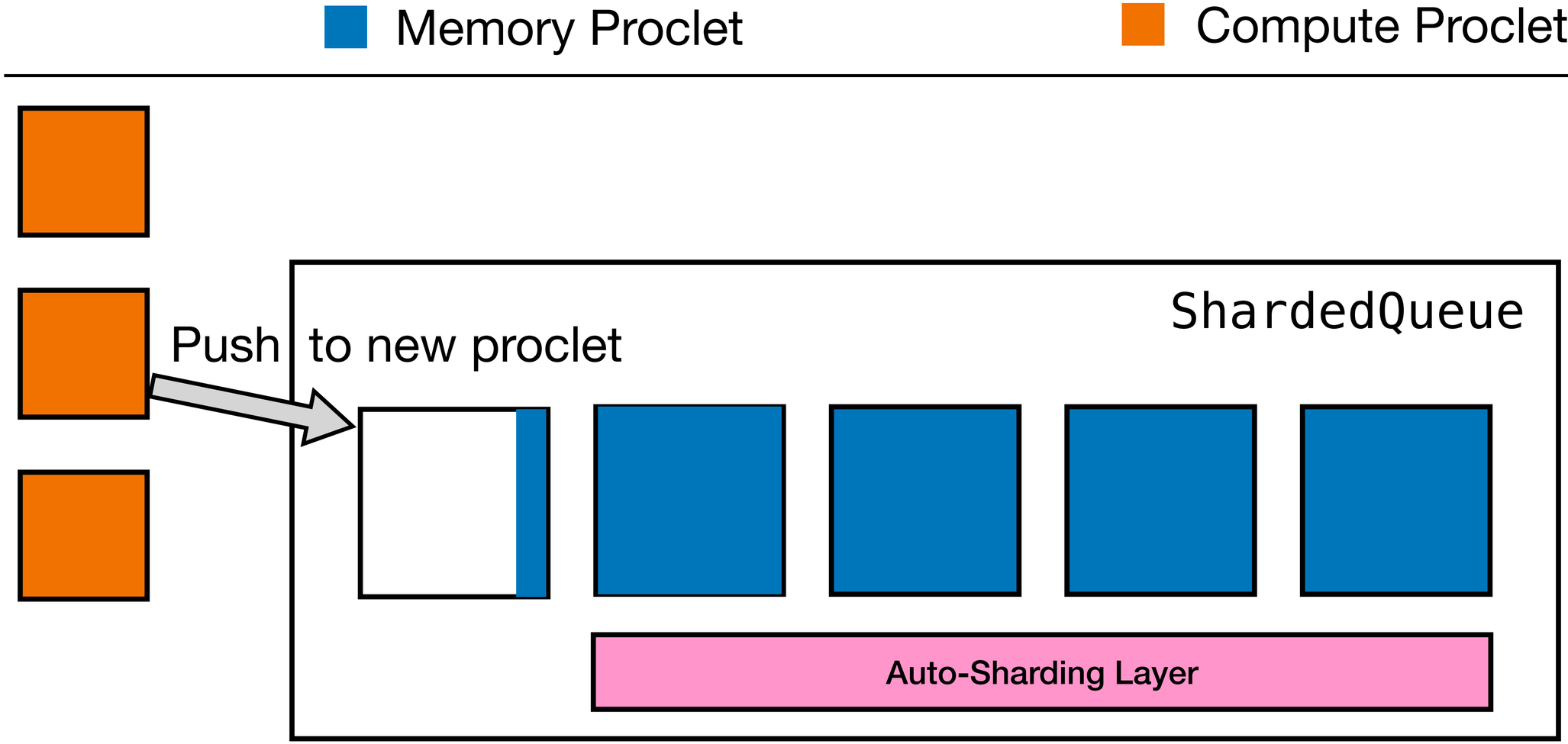
Split Memory Proclets to Maintain Granularity

Split / Merge Memory Proclets

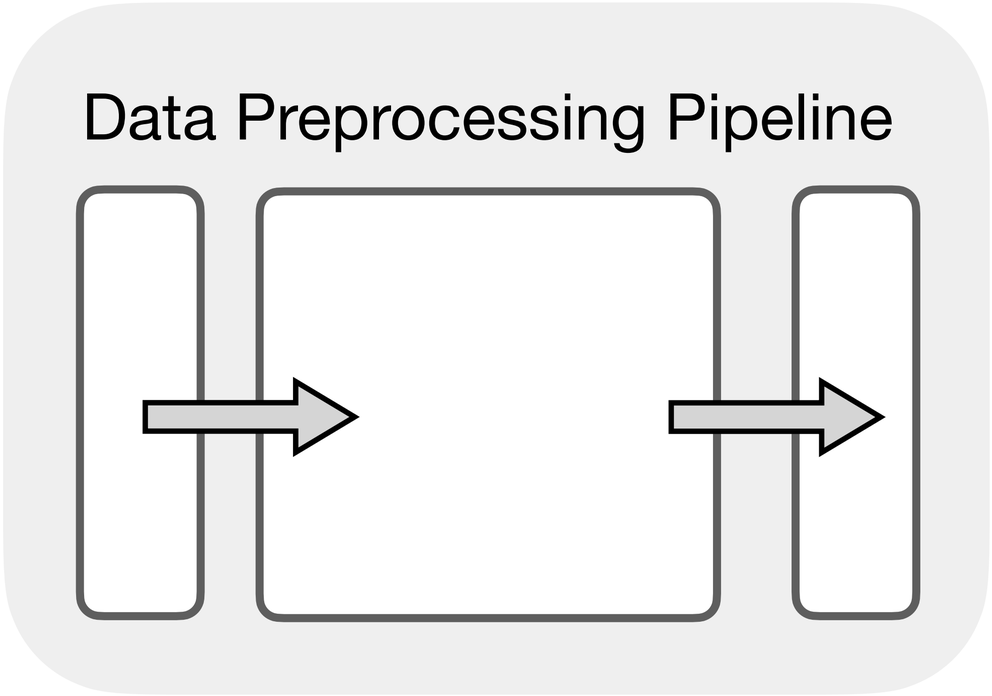


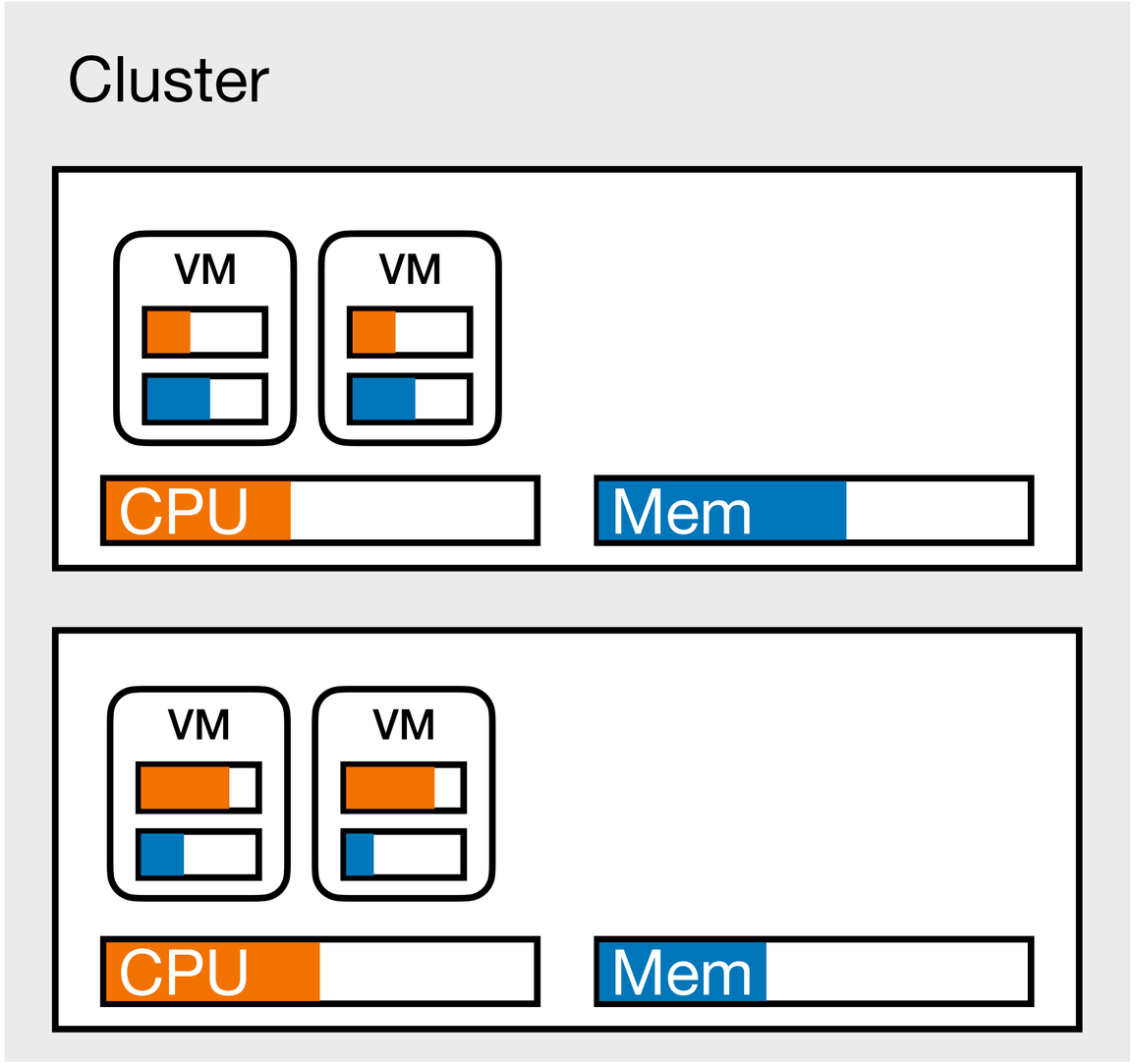
Split Memory Proclets to Maintain Granularity

Split / Merge Memory Proclets



Using Resources with Quicksand

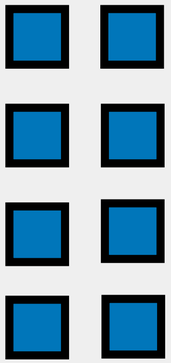


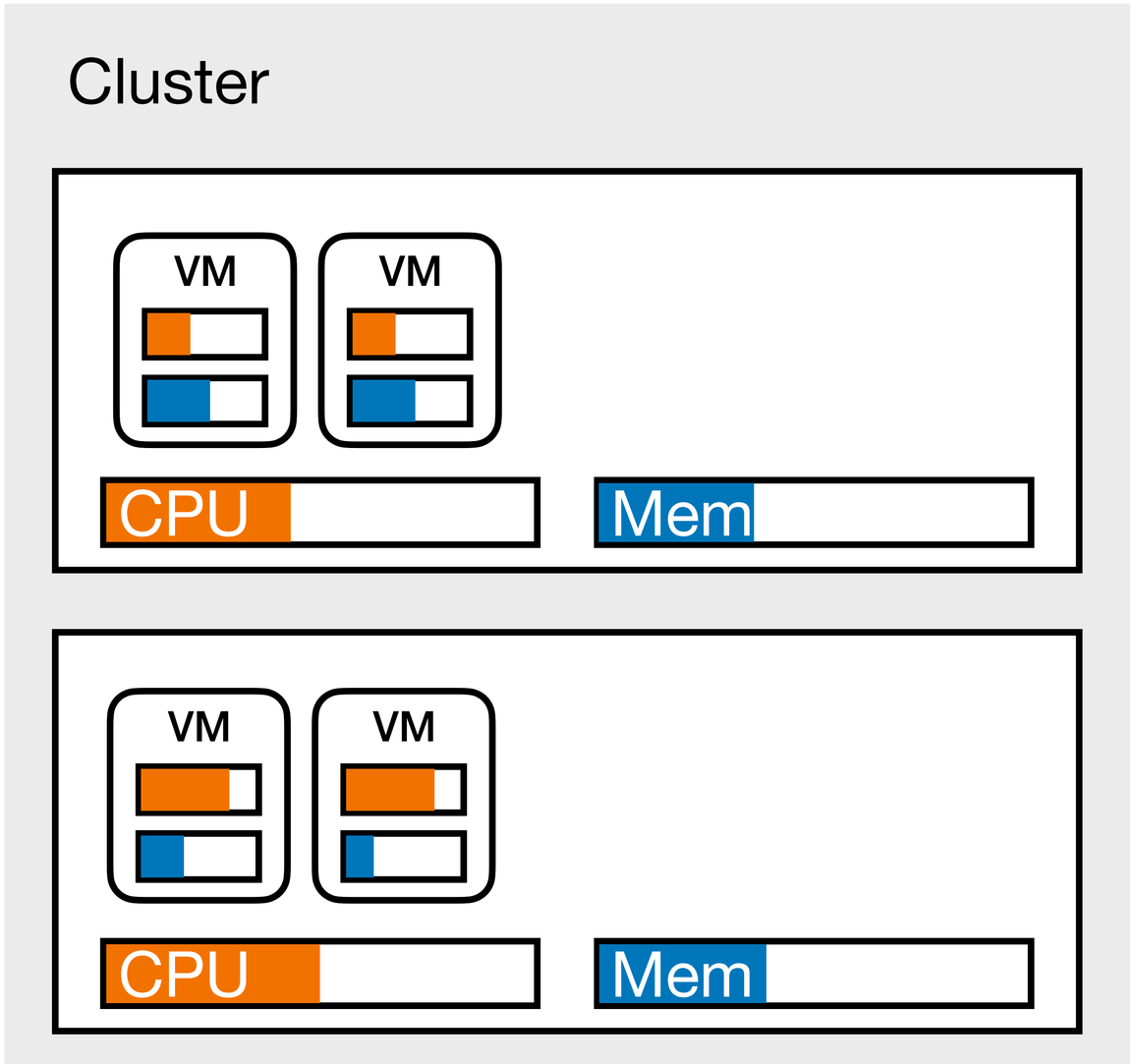


Using Resources with Quicksand

Unprocessed Data Loaded

Data Preprocessing Pipeline

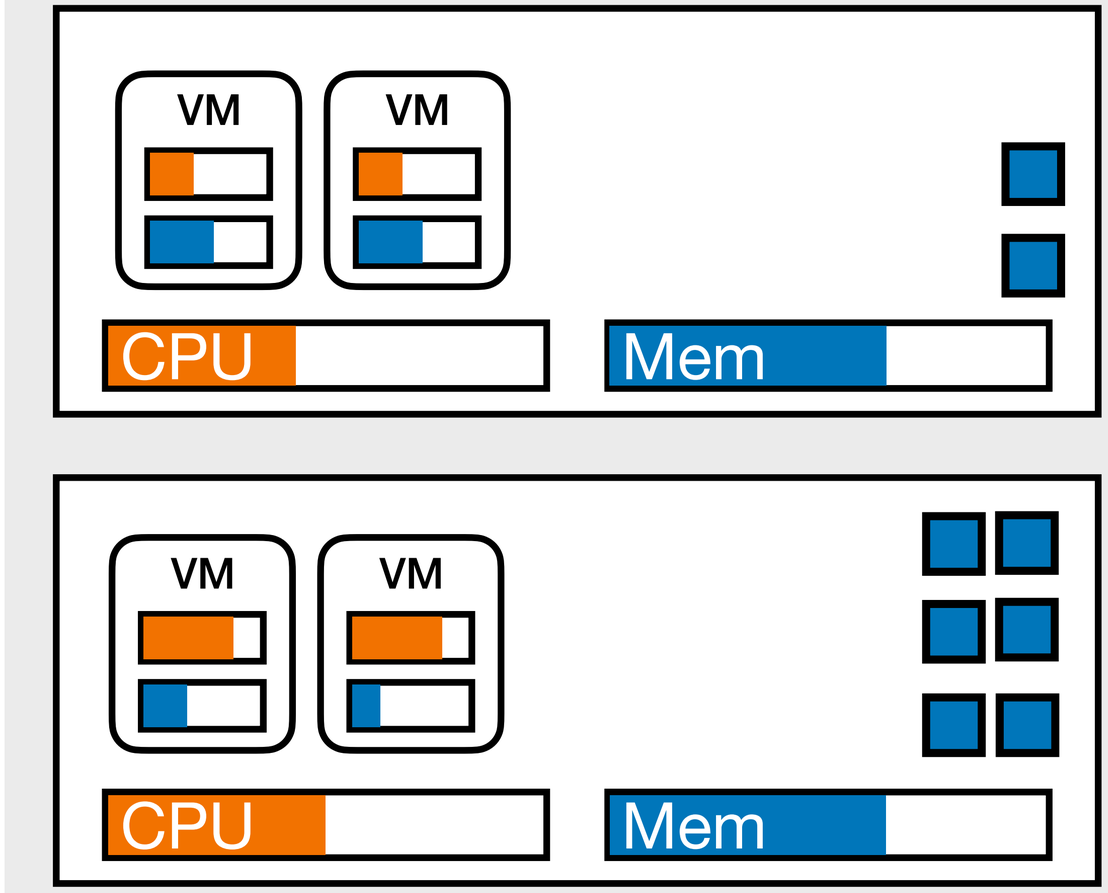




Using Resources with Quicksand

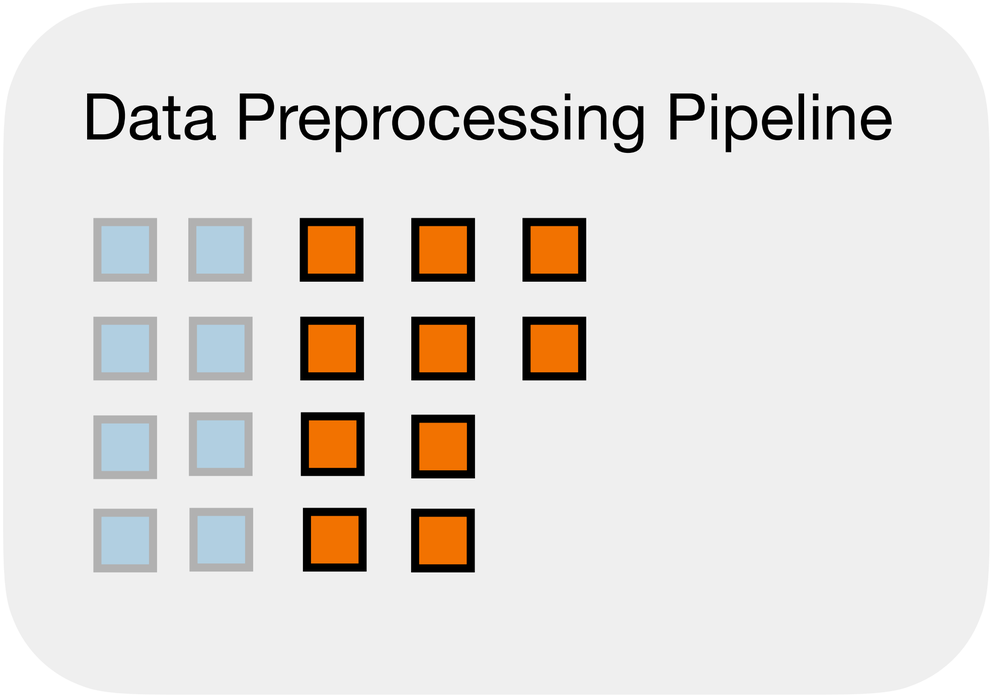
Unprocessed Data Loaded

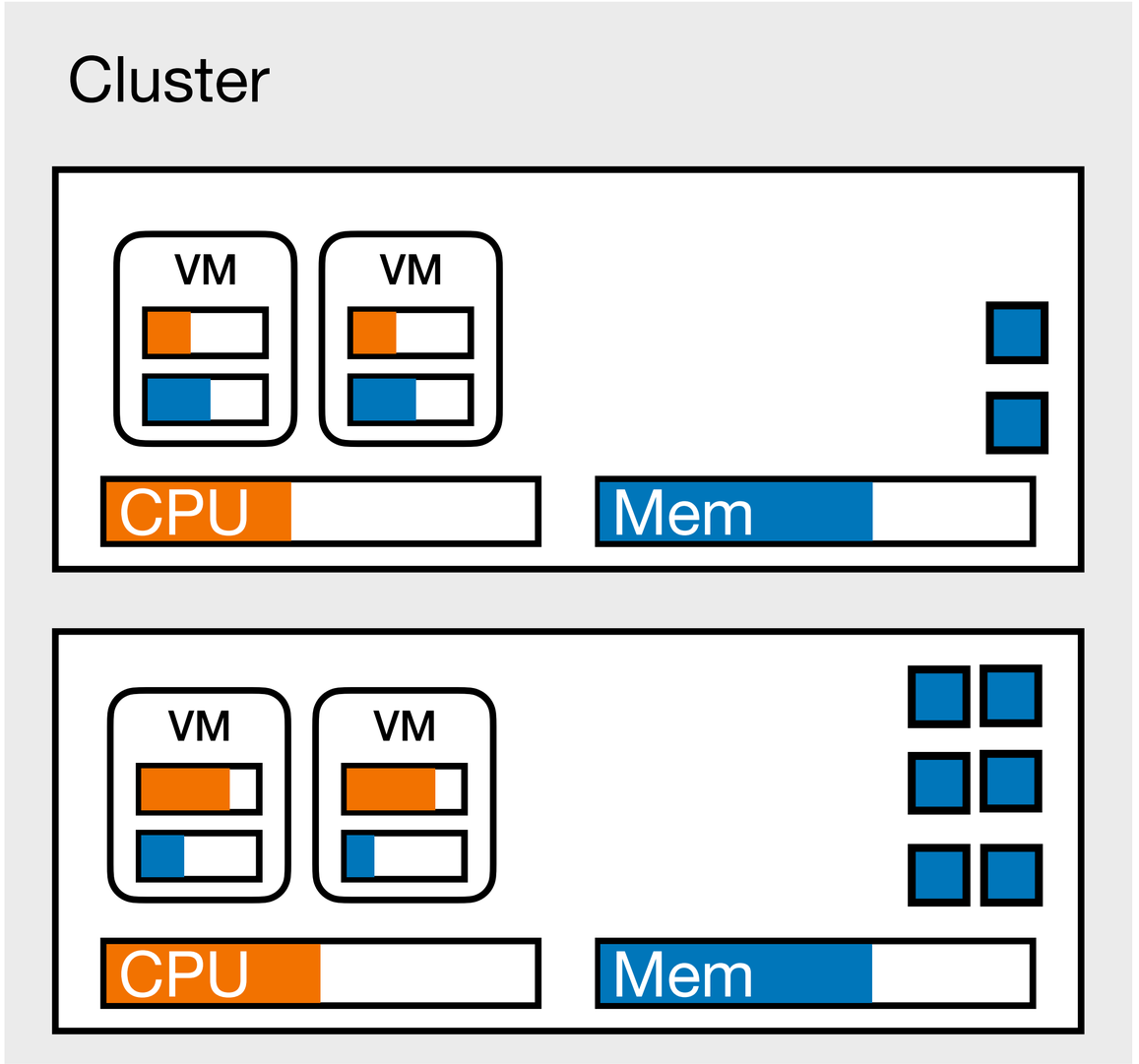
Data Preprocessing Pipeline



Using Resources with Quicksand

Start Pre-processing

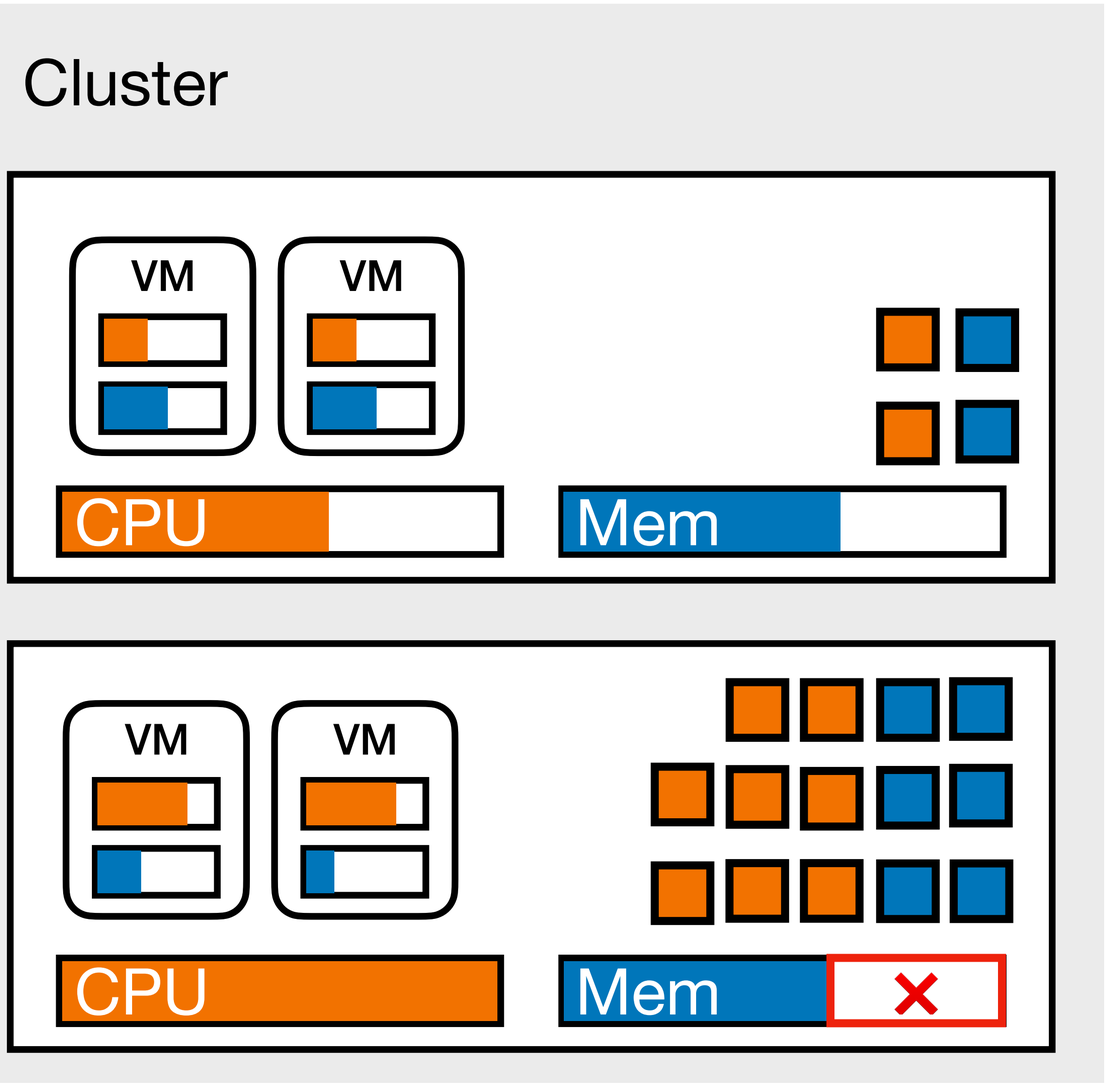




Unstranding Resources with Quicksand

Start Pre-processing

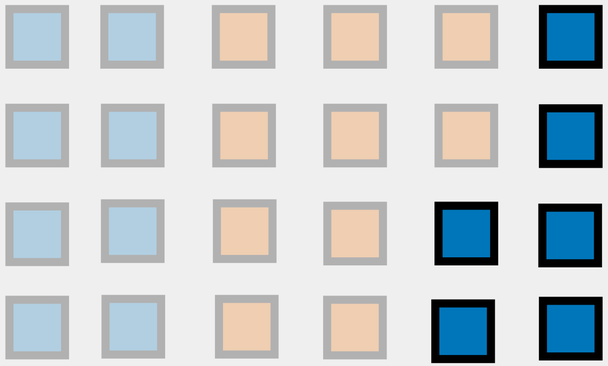
Data Preprocessing Pipeline

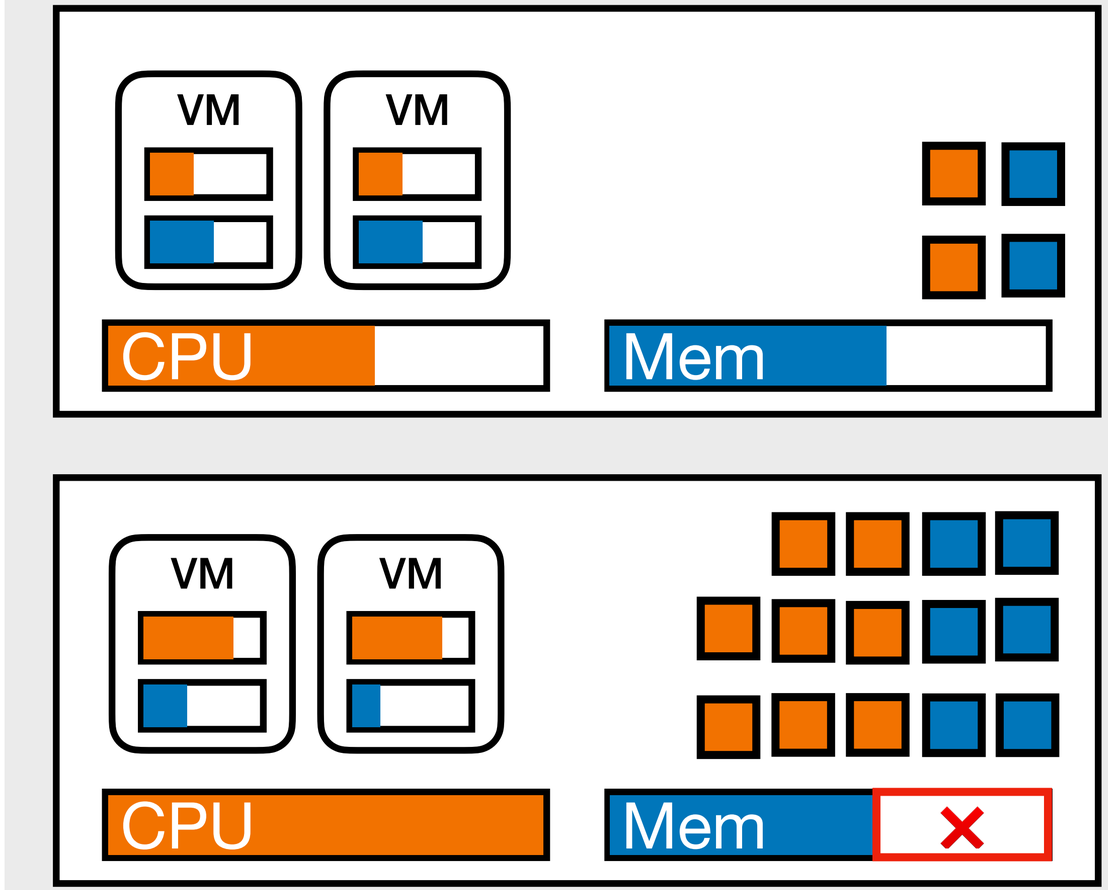


Unstranding Resources with Quicksand

Storing processed images

Data Preprocessing Pipeline

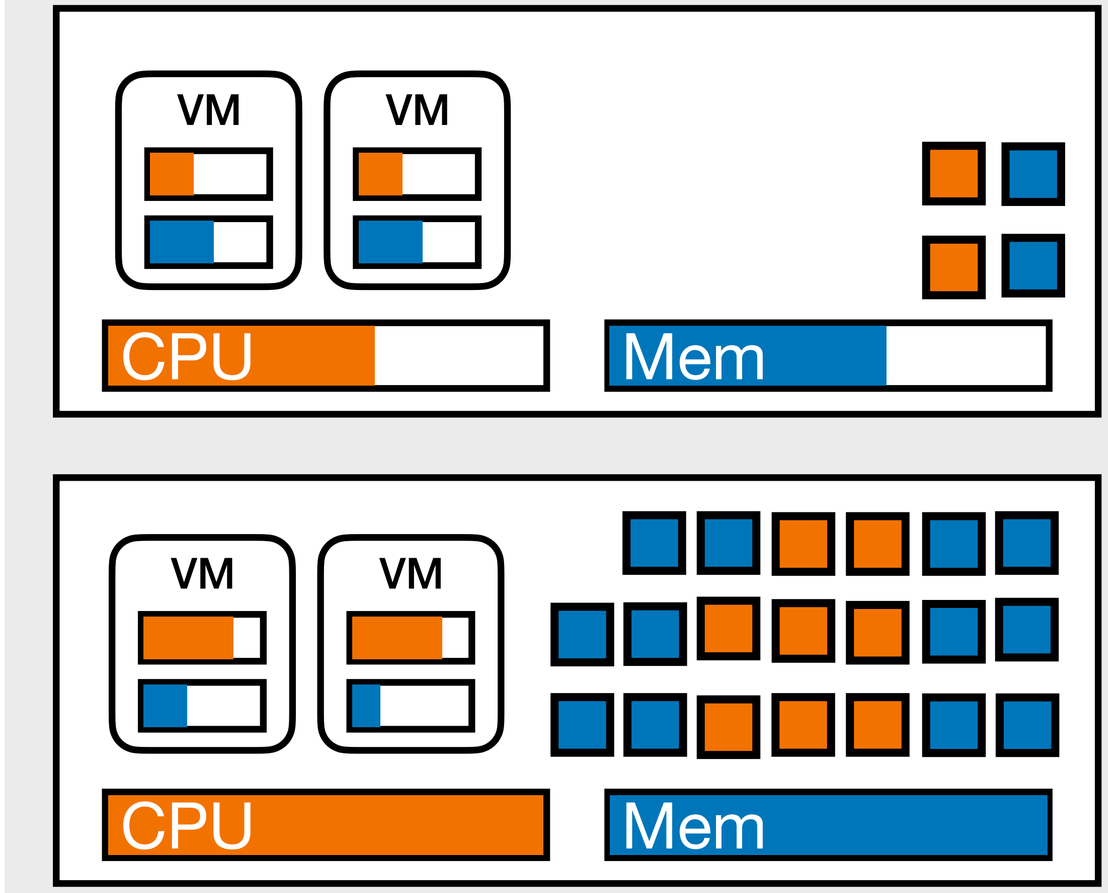




Unstranding Resources with Quicksand

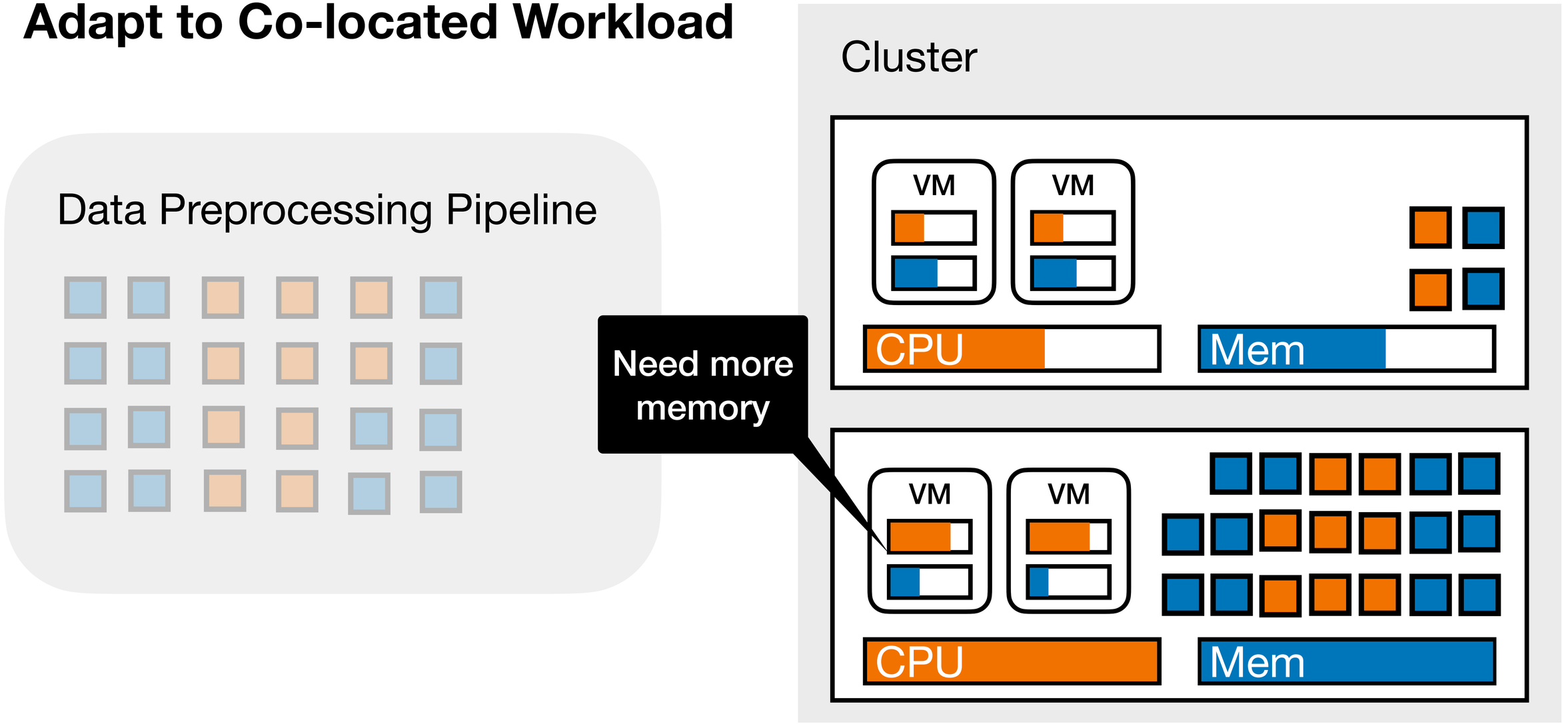
Storing processed images

Data Preprocessing Pipeline



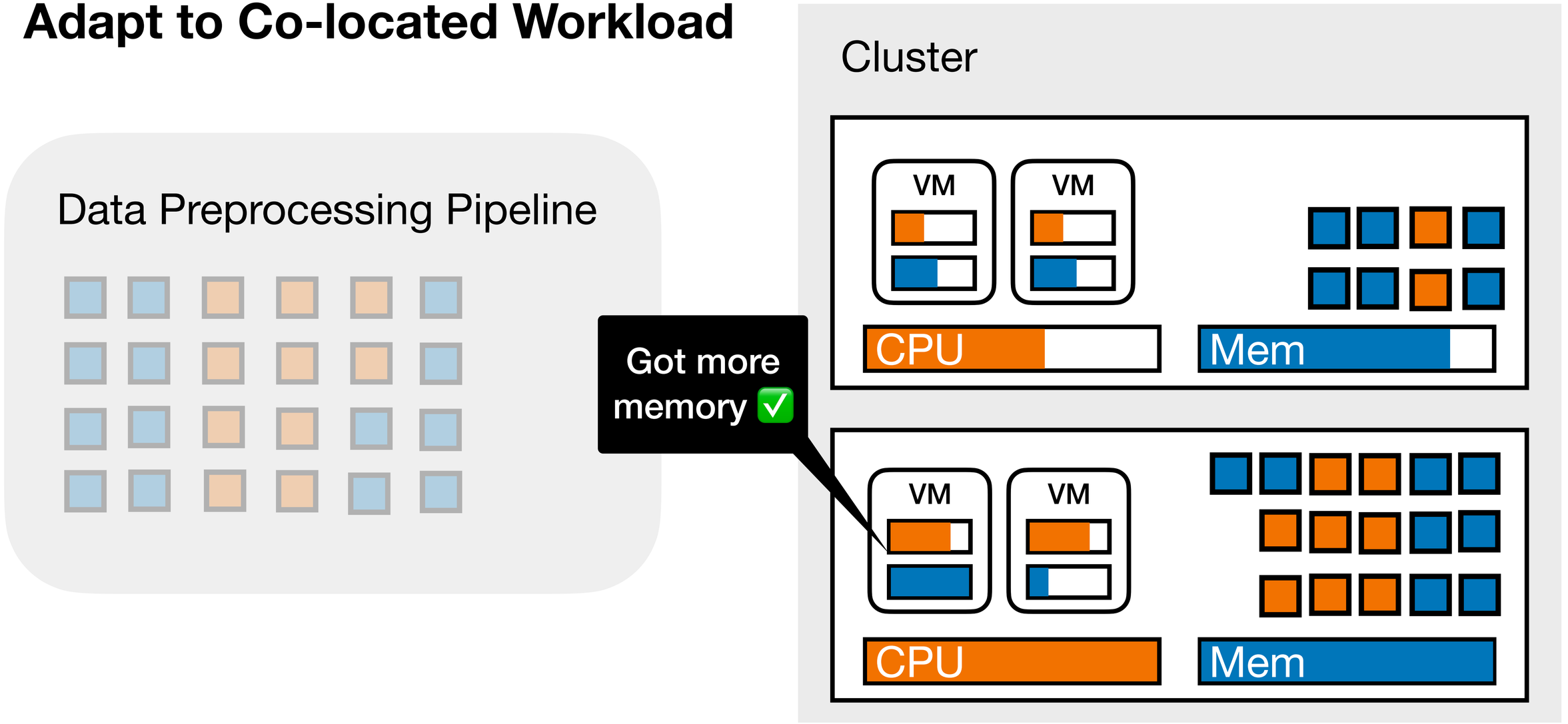
Shift Resource Usage Across Servers

Adapt to Co-located Workload

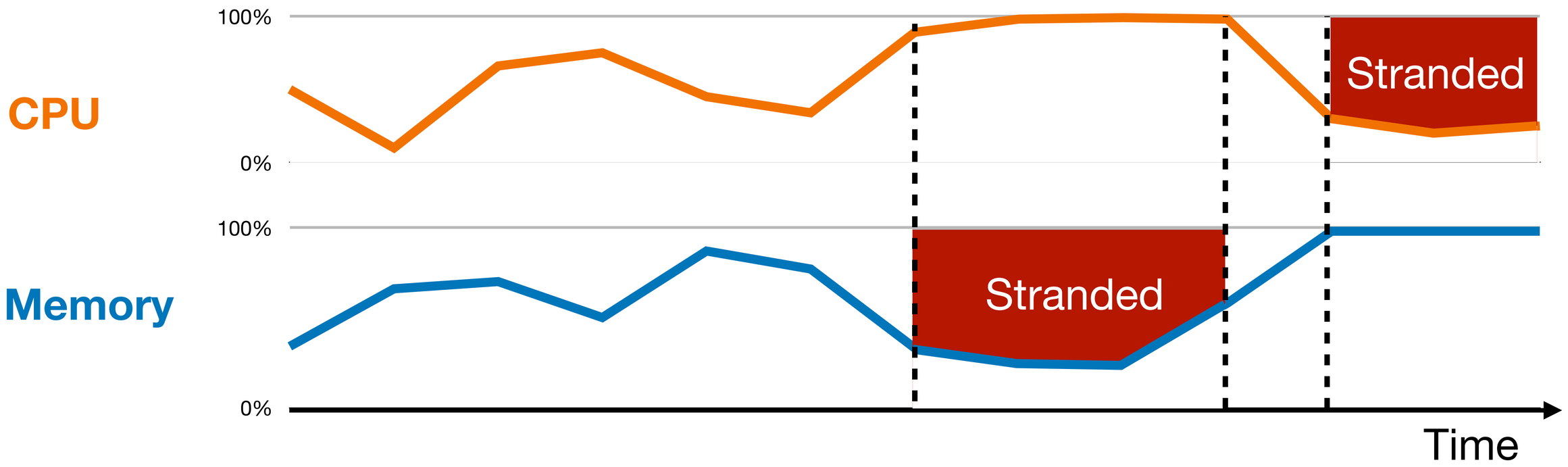


Shift Resource Usage Across Servers

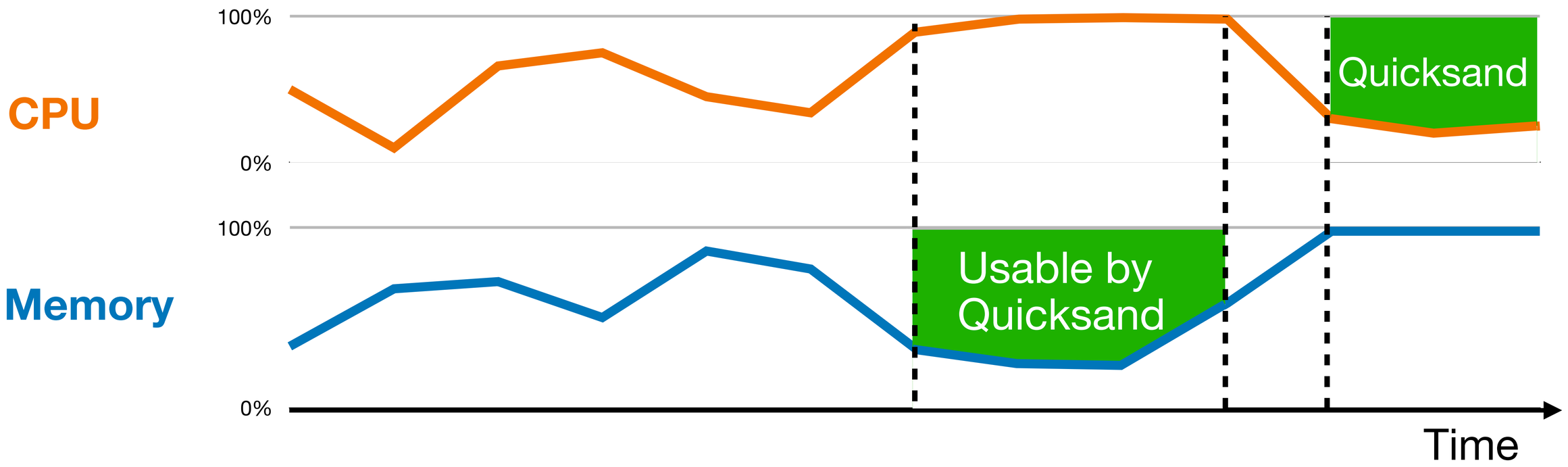
Adapt to Co-located Workload



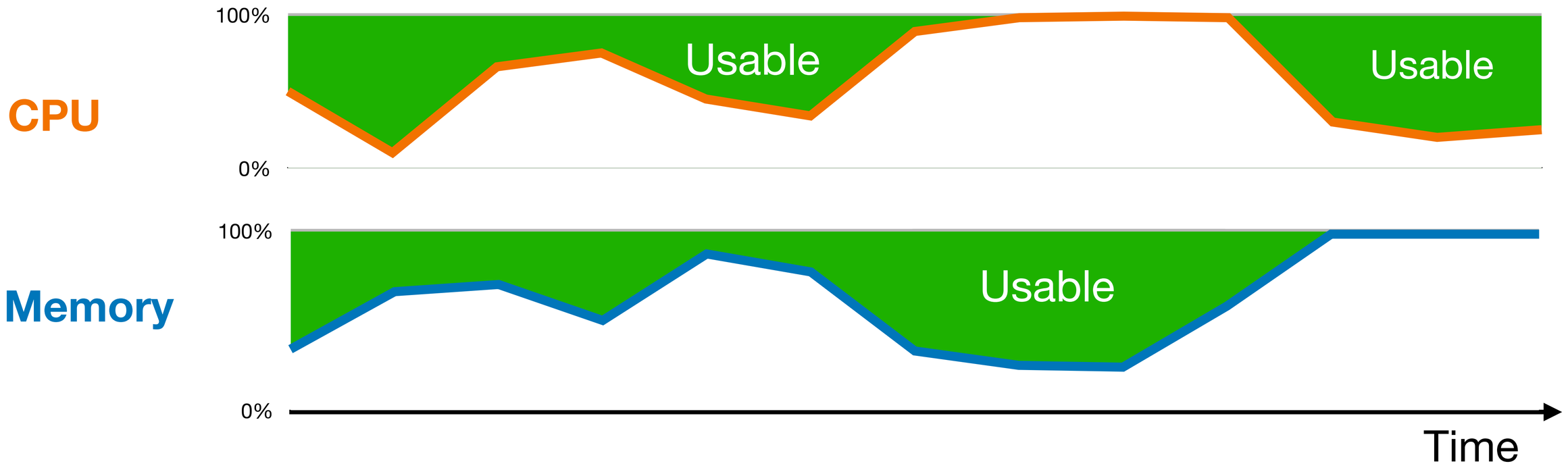
Unstrand Resources with Quicksand



Unstrand Resources with Quicksand



Quicksand Vision: Leave No Resource Idle



Quicksand Prototype and Application Porting

- Quicksand implementation: 10k C++ LoC.
- Implemented on top of Nu [NSDI ’23] and Caladan [OSDI ’20].
- Ported 4 applications:

Application	**ML Data Preprocessing**	**SocialNetwork** DeathStarBench [ASPLOS ’19]	**Distributed Sorting**	**Video Encoding** ExCamera [NSDI ’17]
Quicksand-related LoC	21	98	22	203

Evaluation Questions

1. Does Quicksand unstrand resources better than existing solutions?
2. Can Quicksand respond to changes in resource availability and demand?
3. Do resource proclets separate the use of different resources?
4. Does Quicksand's rapid scaling and fine granularity improve utilization and performance?

Evaluation Questions

1. Does Quicksand unstrand resources better than existing solutions?
2. Can Quicksand respond to changes in resource availability and demand?
3. Do resource proclets separate the use of different resources?
4. Does Quicksand's rapid scaling and fine granularity improve utilization and performance?

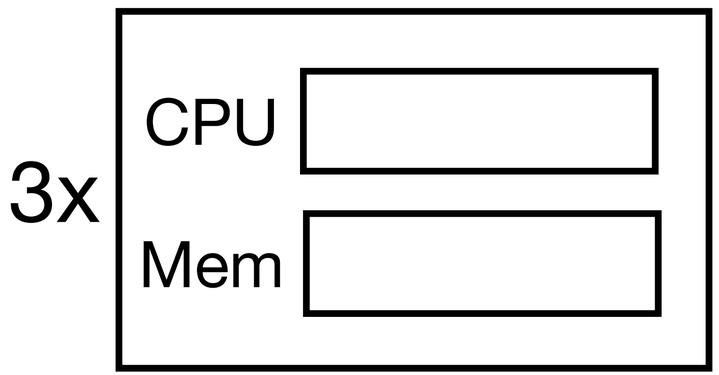
Can Quicksand unstrand resources effectively?

Experiment Setup

Can Quicksand unstrand resources effectively?

Experiment Setup

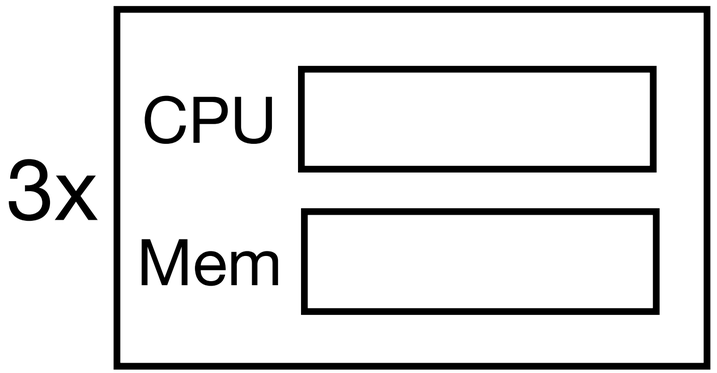
Balanced Setup



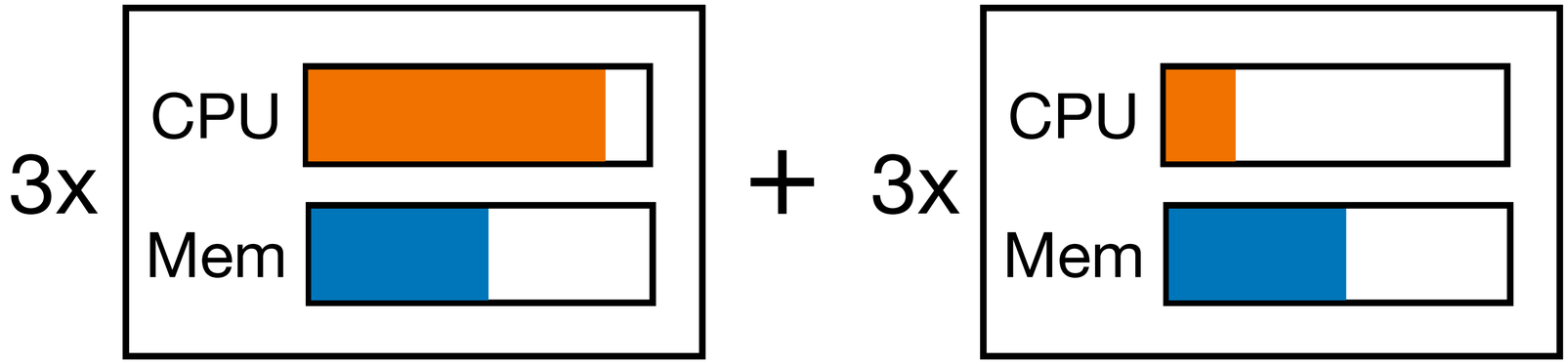
Can Quicksand unstrand resources effectively?

Same amount of idle resources across setups

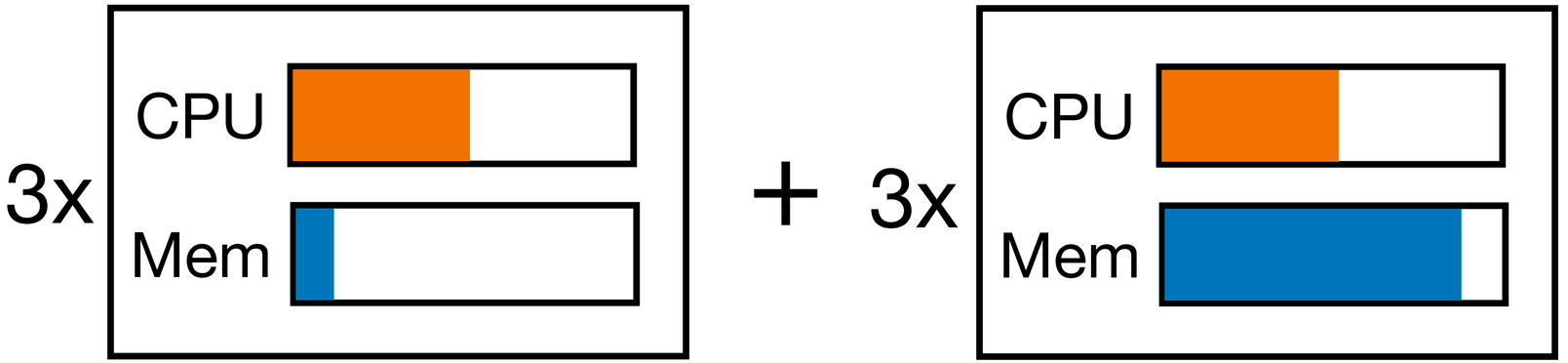
Balanced Setup



CPU-Unbalanced



Mem-Unbalanced



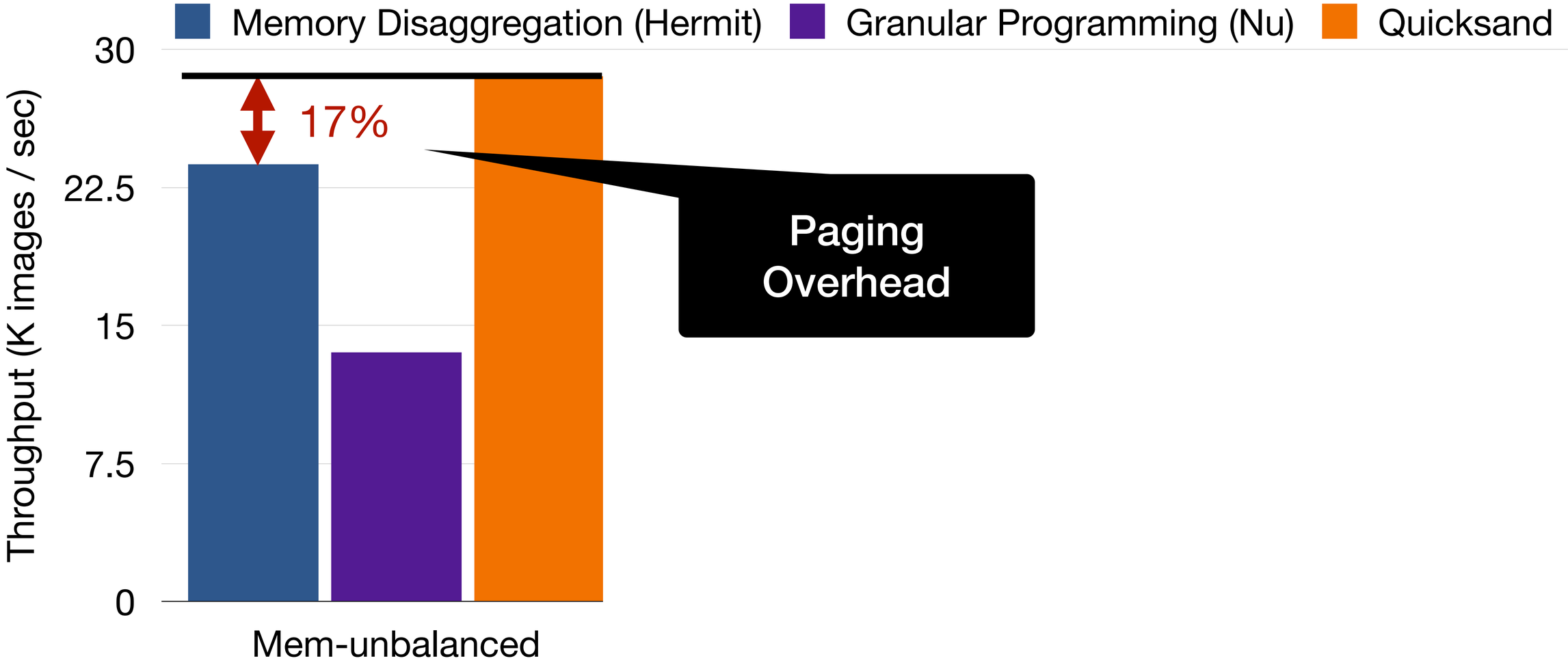
Can Quicksand unstrand resources effectively?

Baselines

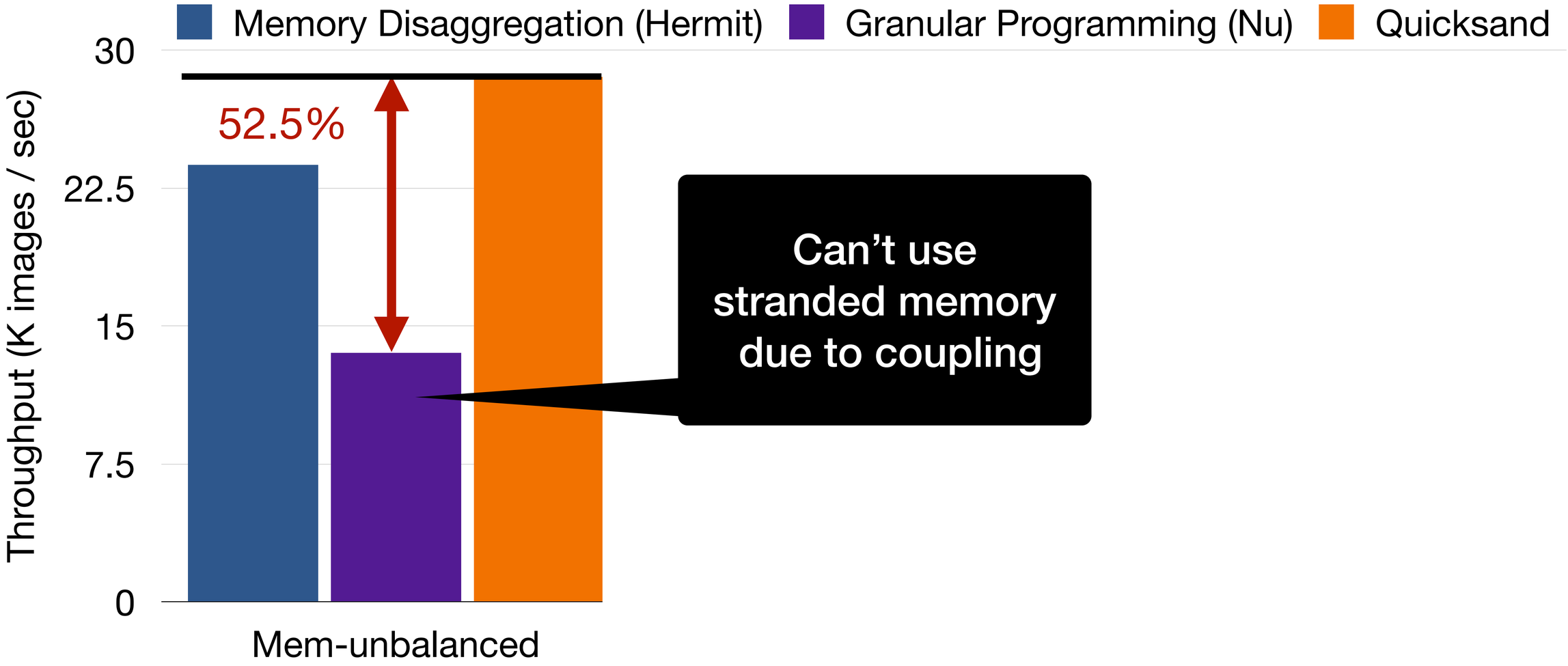
- **Memory disaggregation**: Hermit [NSDI ’23]
- **Granular programming**: Nu [NSDI ’23]

Mem Disaggregation comes with app overhead

But it can unstrand memory

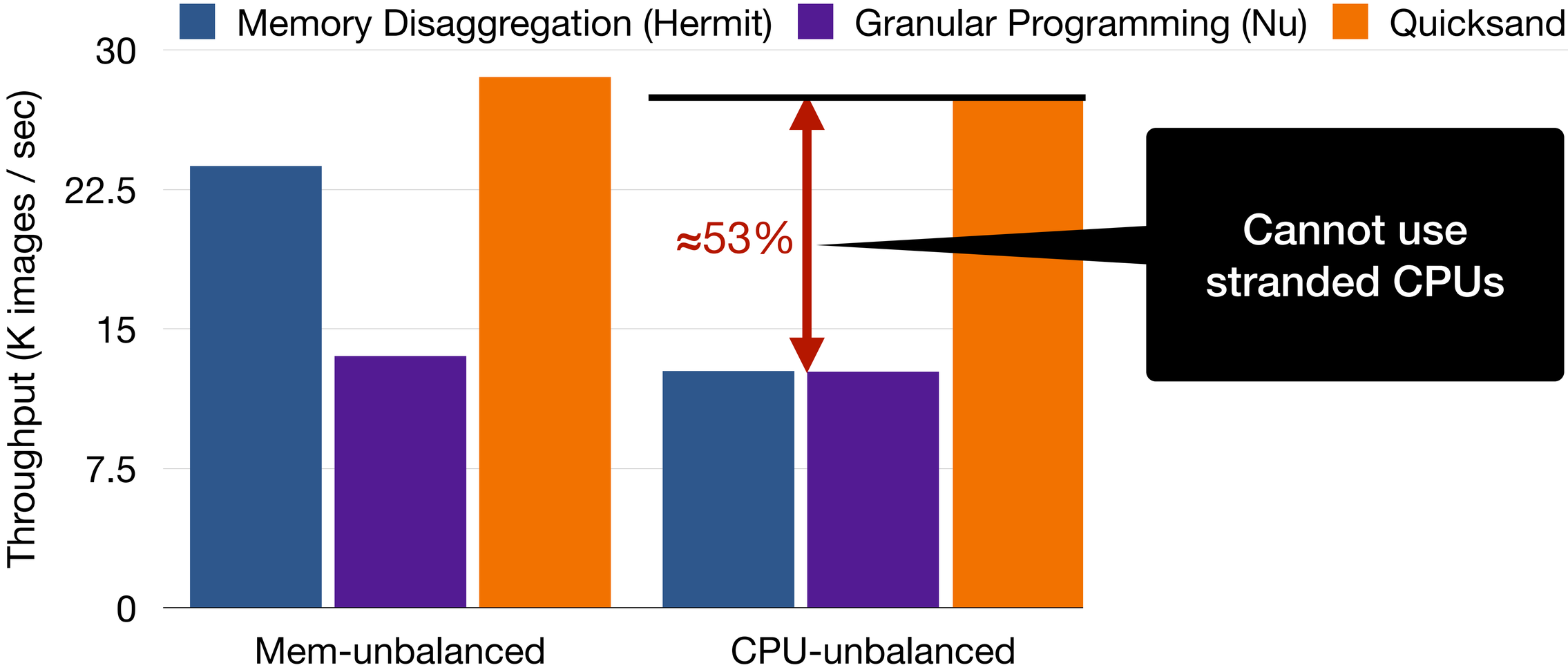


Resource coupling prevents using stranded memory

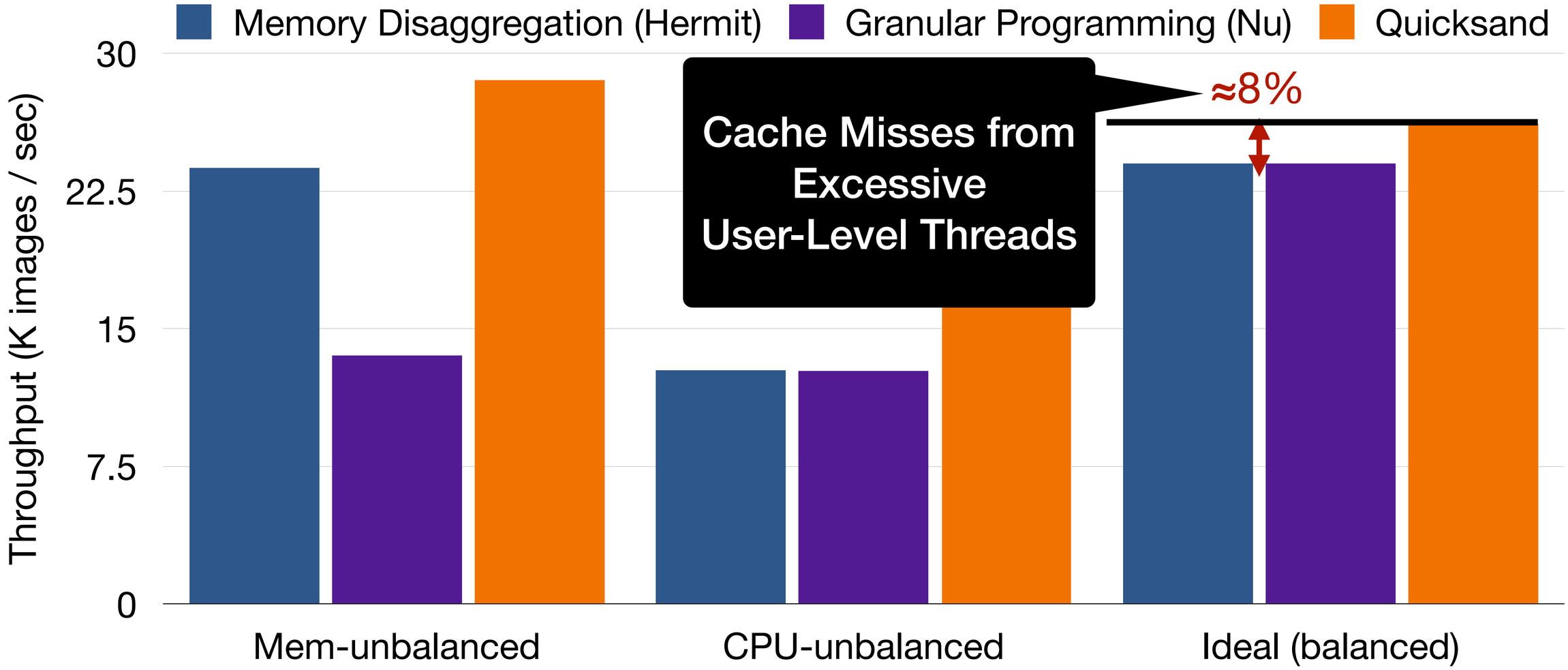


Mem disaggregation doesn’t unstrand CPUs

Nor do granular units that couple resources



Quicksand outperforms in balanced resource setup



More results in the paper

- Millisecond-level scaling up/down to:
  - Use transiently available resources.
  - Adapt to workload changes (phased behavior, load imbalance).
- Resource Proclets effectively separate compute / memory usage.
- The benefits of fine-granularity.

Quicksand

A new programming framework that unstrands datacenter resources

- **Motivation**: Resource stranding is a major inefficiency in today's datacenters.
- **Approach**: Unstranding via SW, not via HW resource disaggregation.
- **Key Insight**: Unstrand by decomposing apps into units that primarily use one resource.
- **Evaluation**: Quicksand can use stranded resources in various workloads.
- Quicksand is open source at: github.com/NSDI25-Quicksand/Quicksand