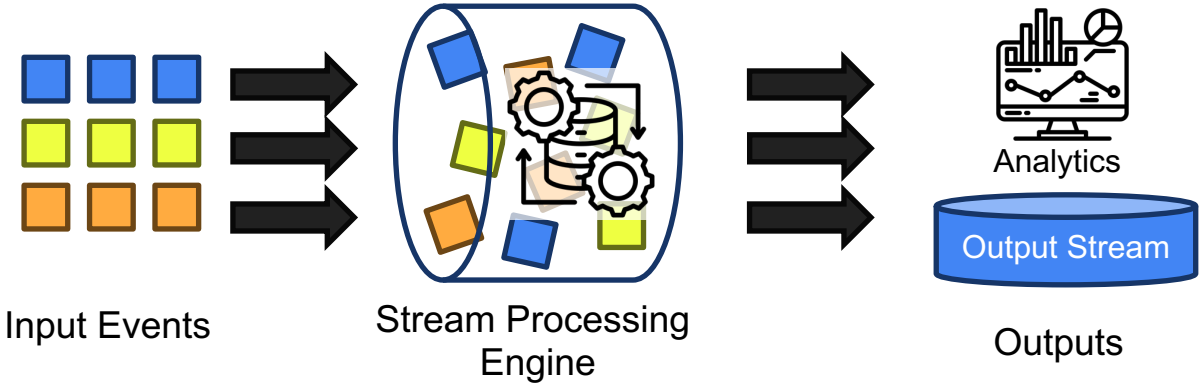
Sponge:
Fast Reactive Scaling for Stream Processing with Serverless Frameworks

Won Wook SONG, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, Byung-Gon Chun

Seoul National University, Samsung Research, Microsoft Research, UNIST, FriendliAI

Stream Processing Happens Continuously



Stream processing deals with real-time data
→ Latency-critical

Stream Processing System Requirements

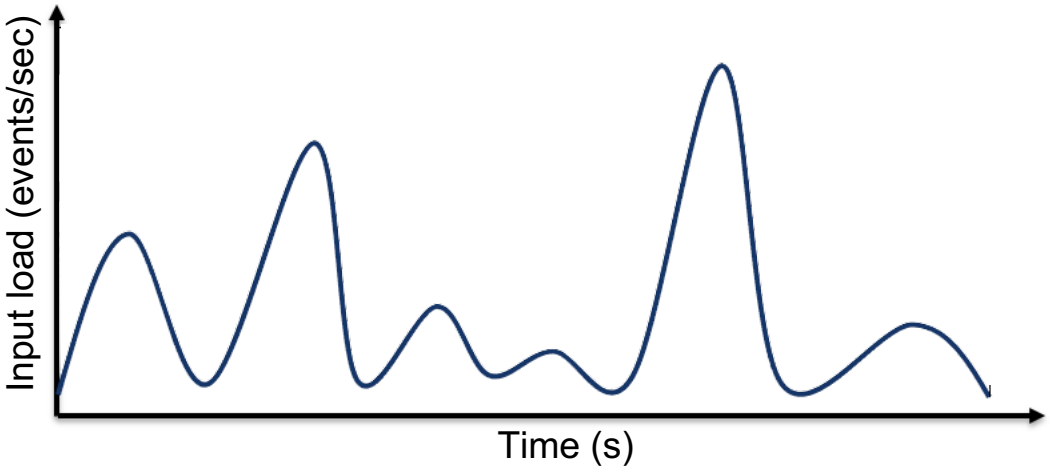
Low latency

High throughput

Correctness

Resource Efficiency

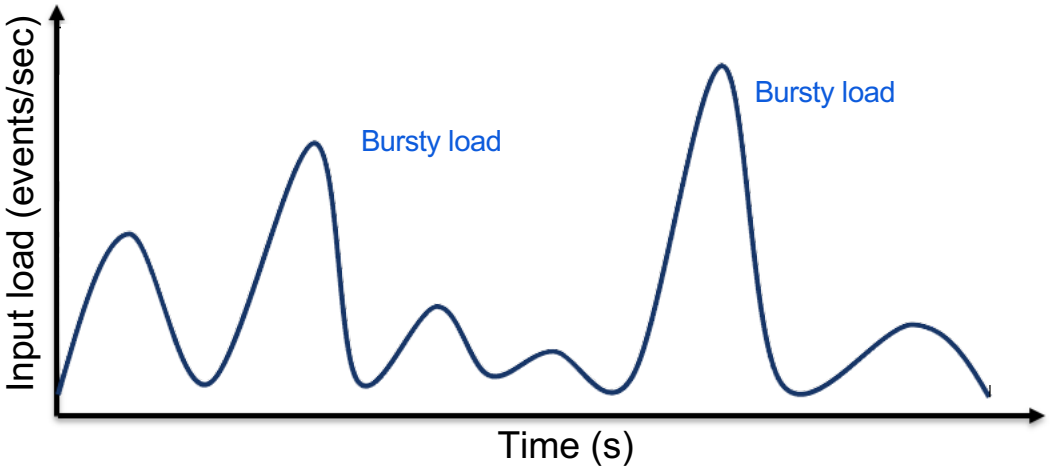
Input Patterns of Stream Workloads: Unpredictable



Stream data are generated in real-time,

which are irregular and unpredictable, due to unforeseen events

Input Patterns of Stream Workloads: Bursty

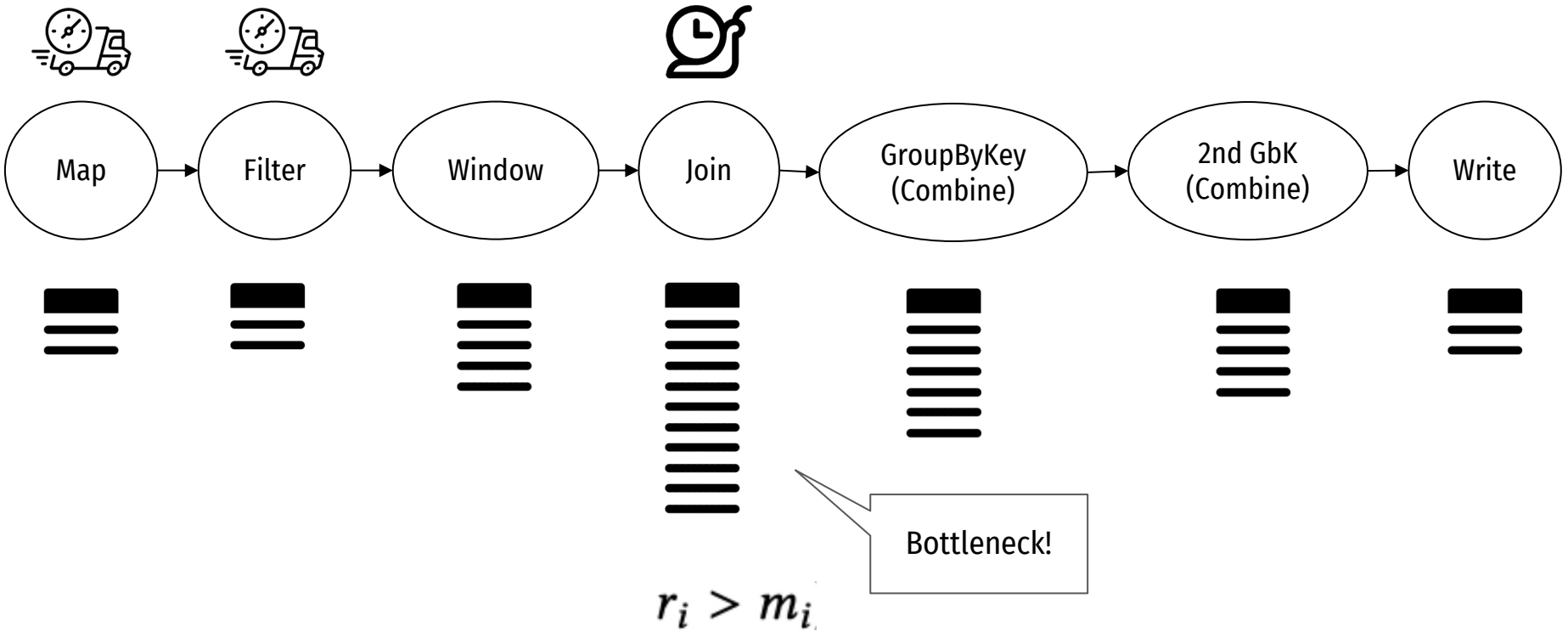


Real-time data can occur in sporadic bursts, due to random events (e.g., influencer tweets, breaking news, natural disasters)*

*Rastegar et. al., Rule caching in sdn-enabled base stations supporting massive iot devices with bursty traffic. (IEEE IoT Journal ‘20)

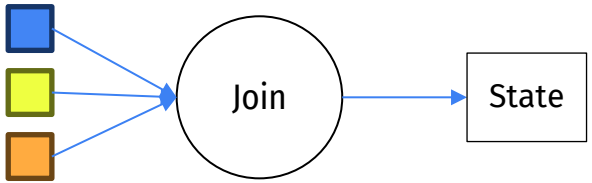
*Robinson et. al., A sensitive twitter earthquake detector. (WWW Companion ‘13)

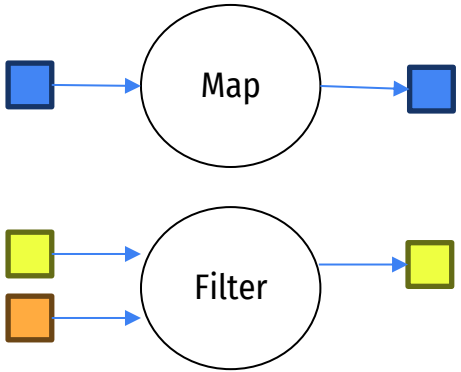
Bursty Input Data Builds Up and Clogs the Pipeline



Input rate > Max throughput

Stream Operators: Stateful vs. Stateless

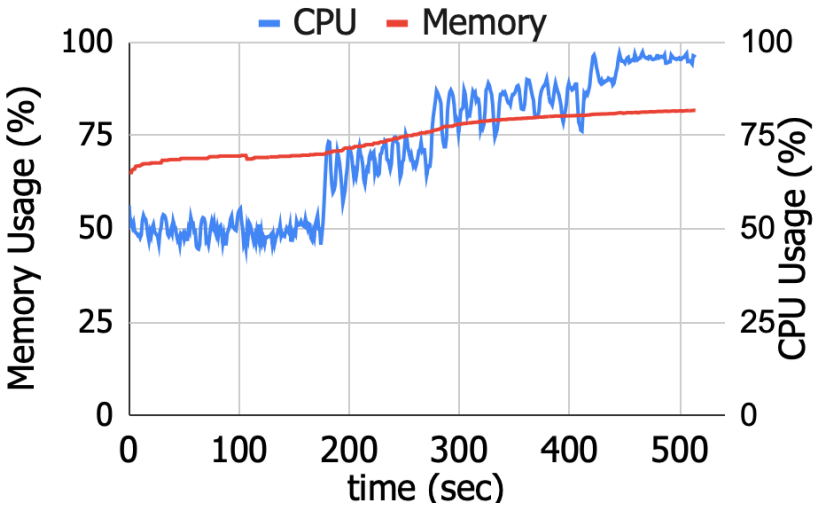




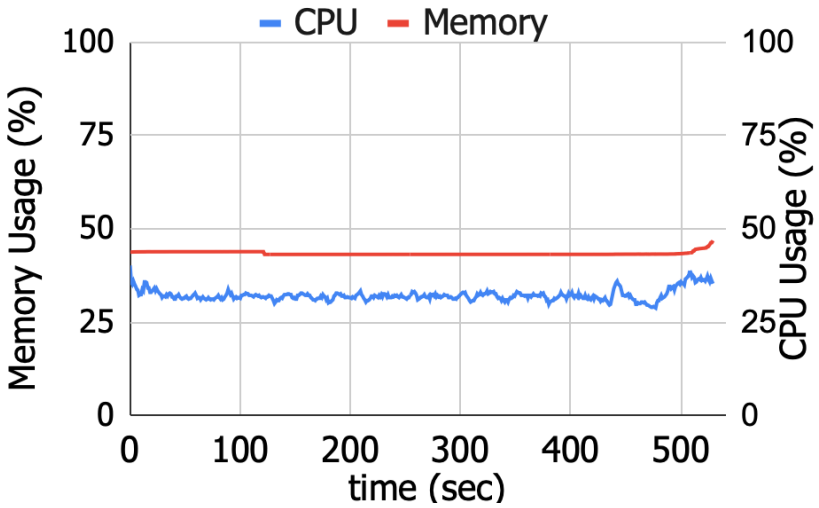
CPU/Memory trace of a stateful join operator
Stateful operators are more tricky to handle due to state handling (e.g., migration)

CPU/Memory trace of a stateless map operator
Stateless operators can easily scale-out

Stream Operators: Stateful vs. Stateless

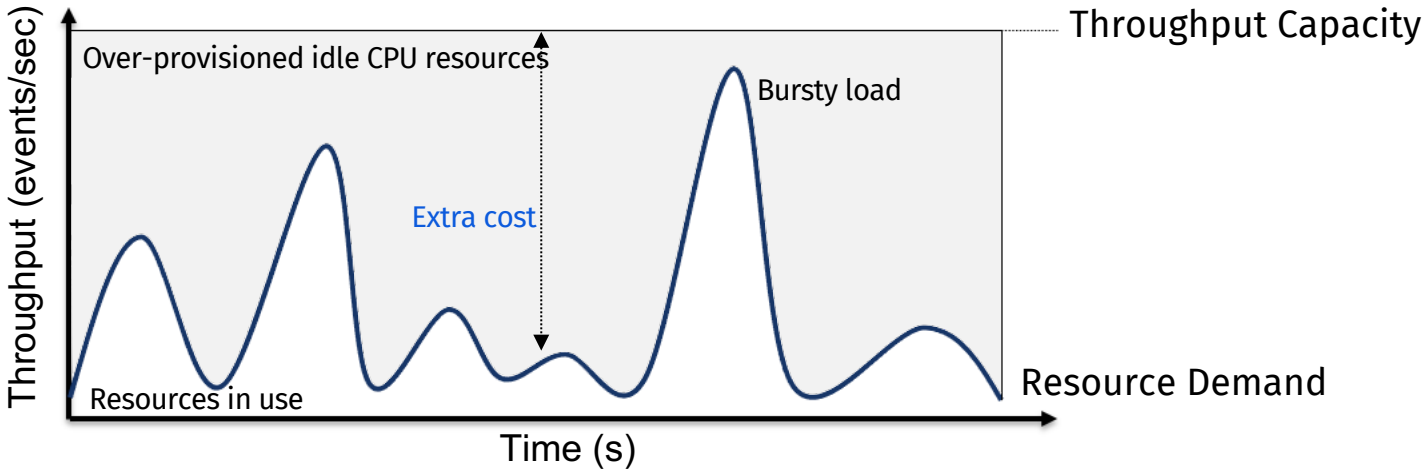


CPU/Memory trace of a stateful join operator
Stateful operators are more tricky to handle due to state handling (e.g., migration)



CPU/Memory trace of a stateless map operator
Stateless operators can easily scale-out

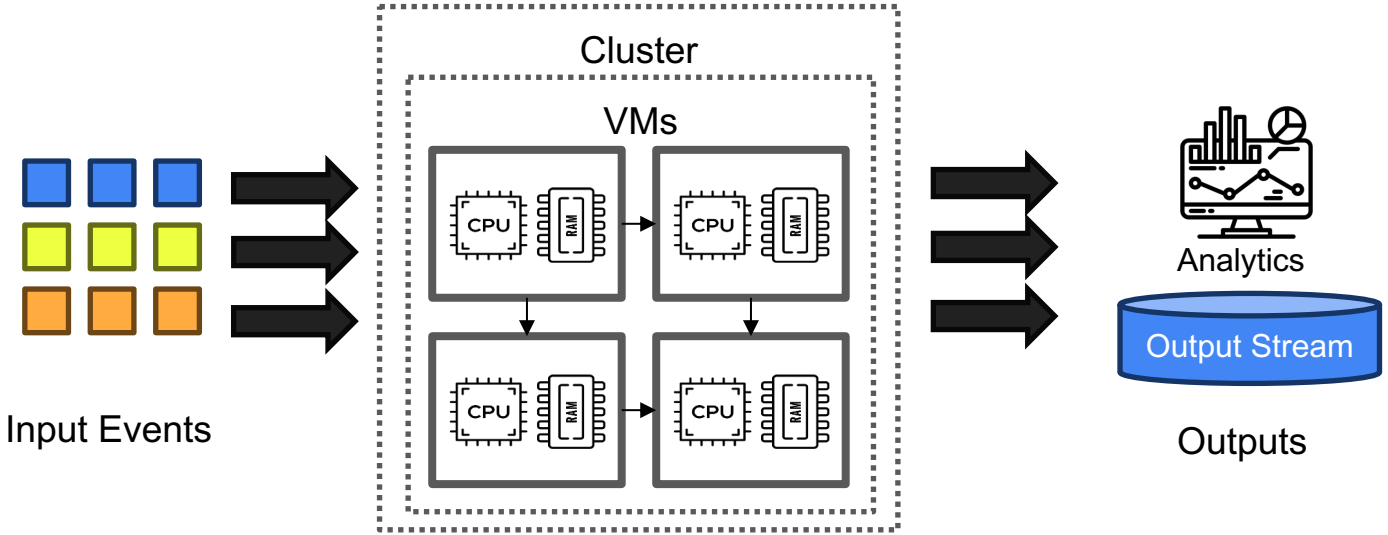
Preventing Latency with Over-provisioned Resources



Simplest solution, but bursty loads are unpredictable

→ Must reserve 5-10x resources at all times = *costly*

Scaling with On-Demand Virtual Machines (VMs)



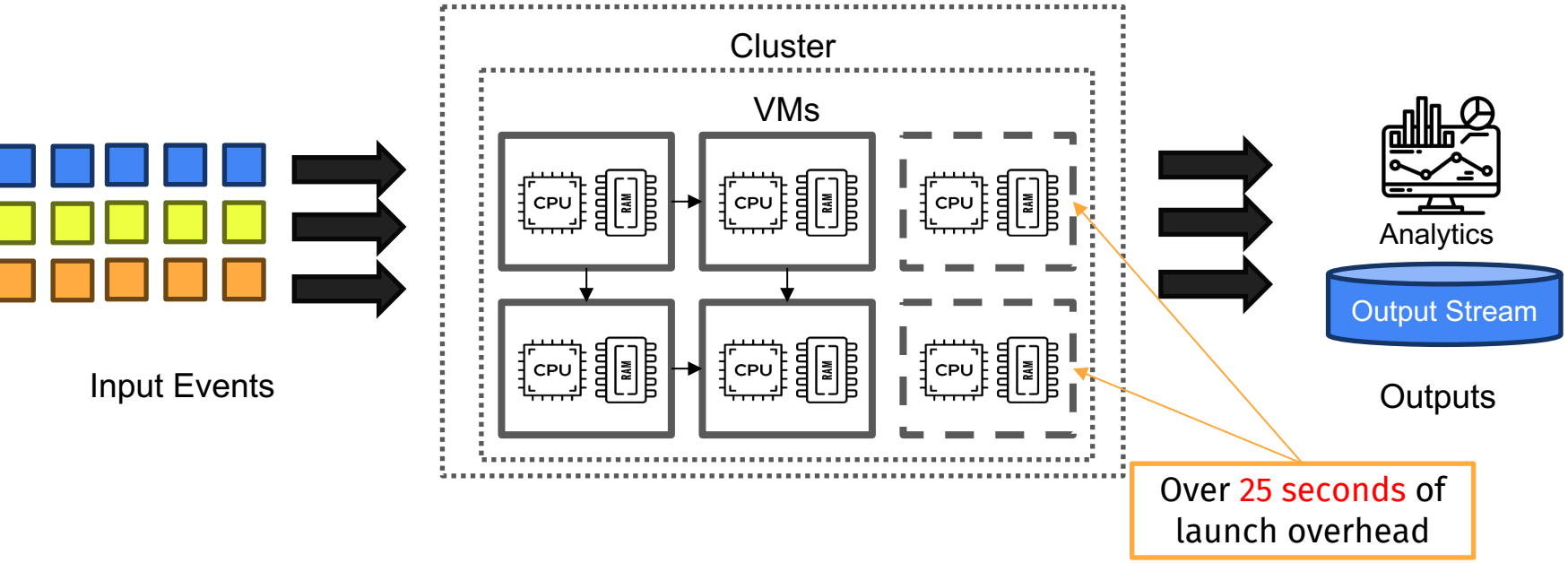
Machine-isolated by bare-metal hypervisors

Fixed specification of CPU and memory

10Gbps network

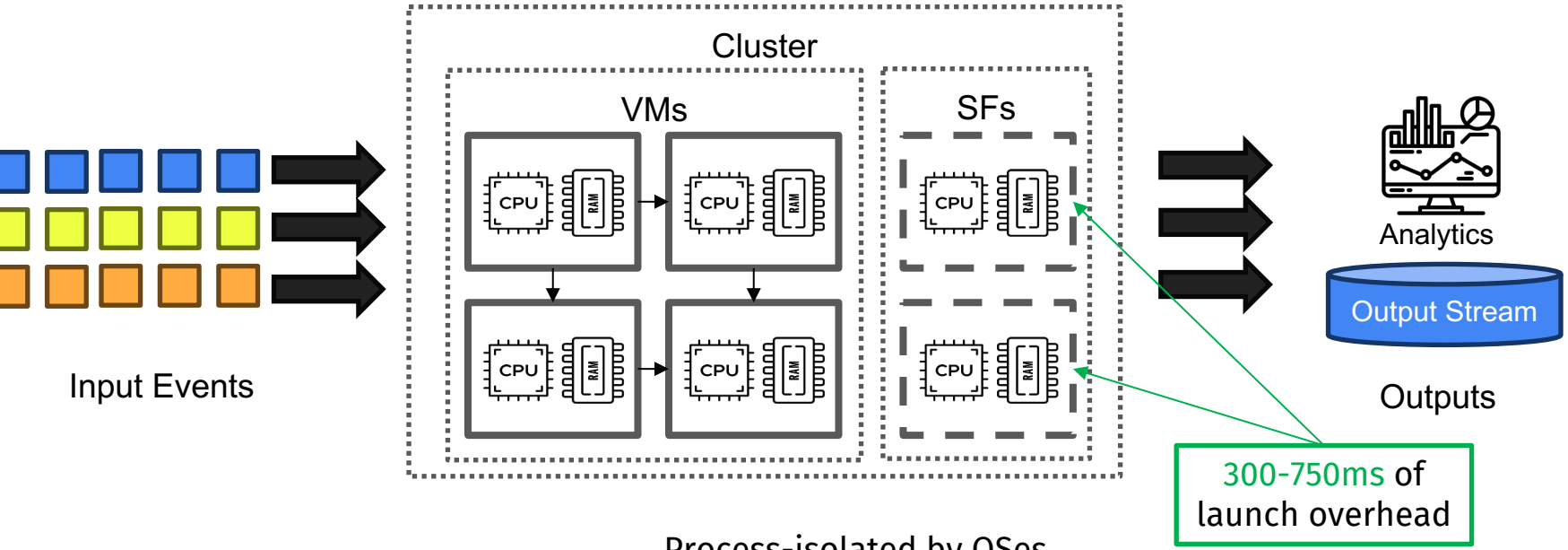
Stable and powerful

Scaling with On-Demand Virtual Machines (VMs)



VM Start-up Time is Too Slow (25-30s)

Scaling with On-Demand Serverless Functions (SFs)



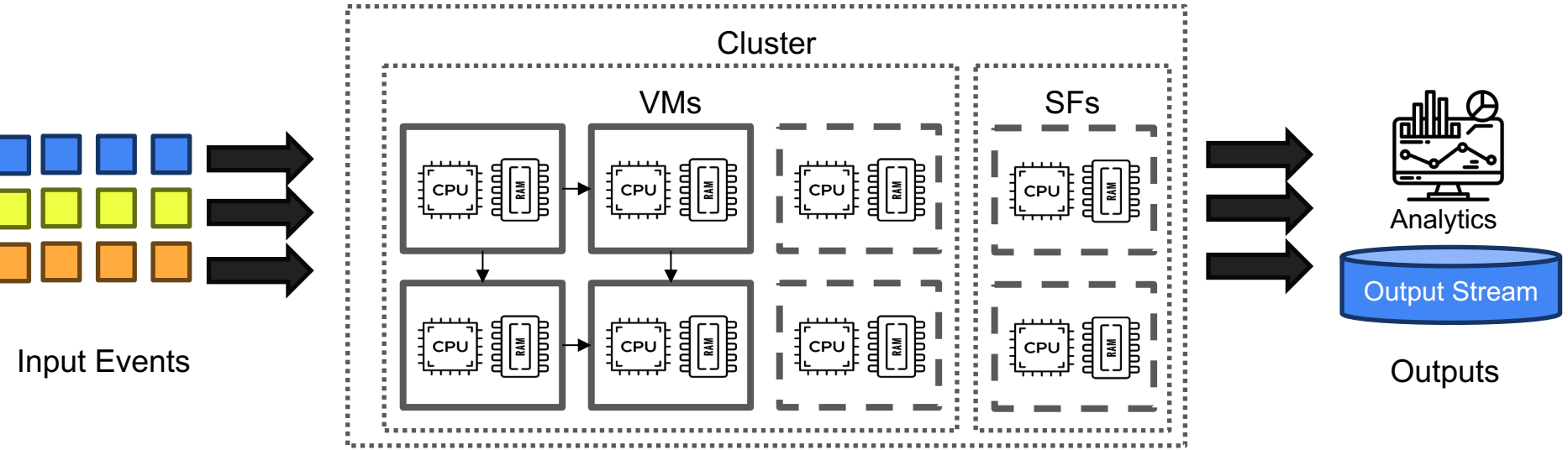
Process-isolated by OSes

Flexible allocation of CPU according to mem size

800-1200Mbps network per instance

4x more expensive than VMs

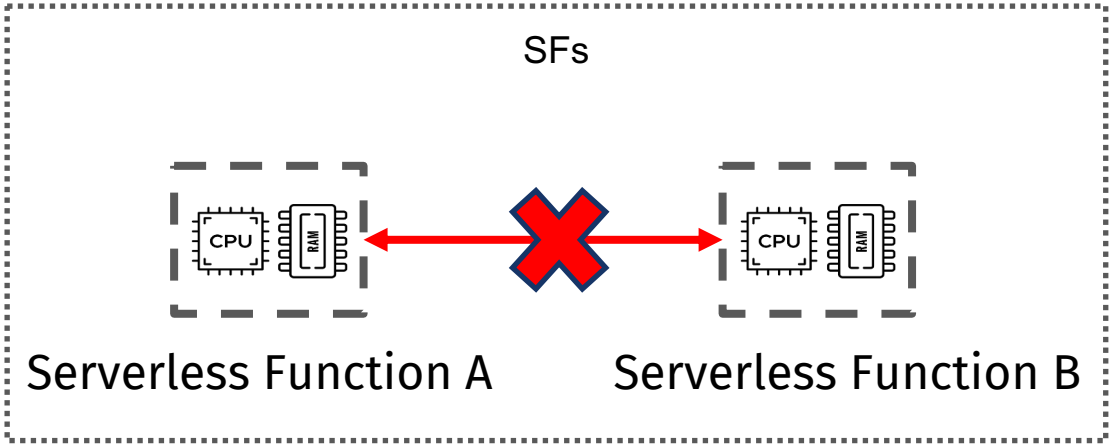
Scaling with On-Demand VMs and SFs



SFs to handle short-living bursty input loads &

VMs to handle long-living input loads

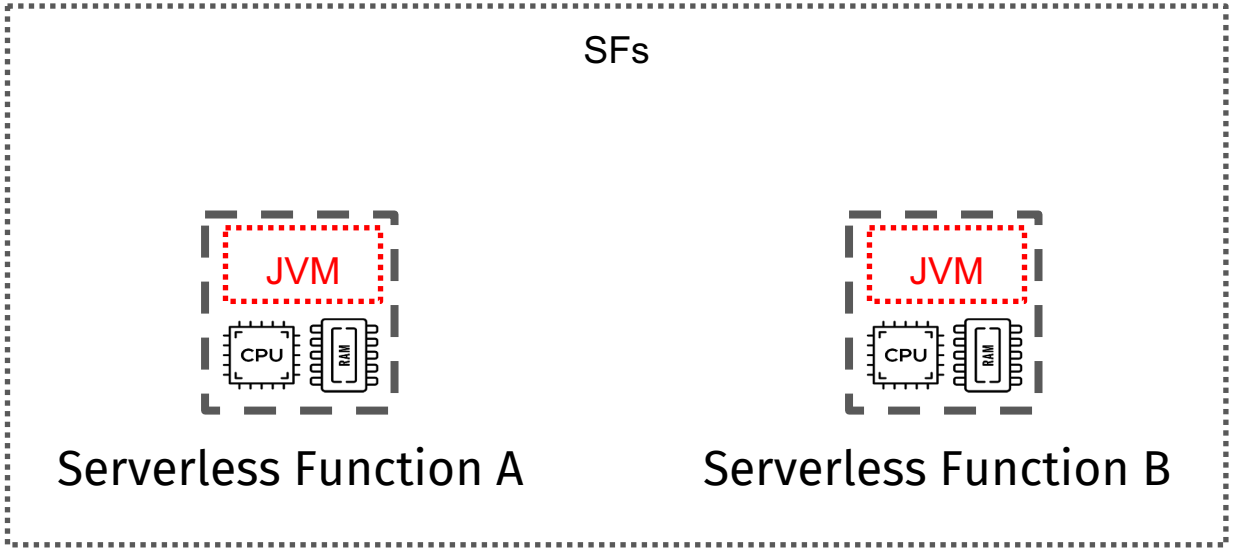
Direct Network Communications are Prohibited among SFs



Serverless instances are not designed

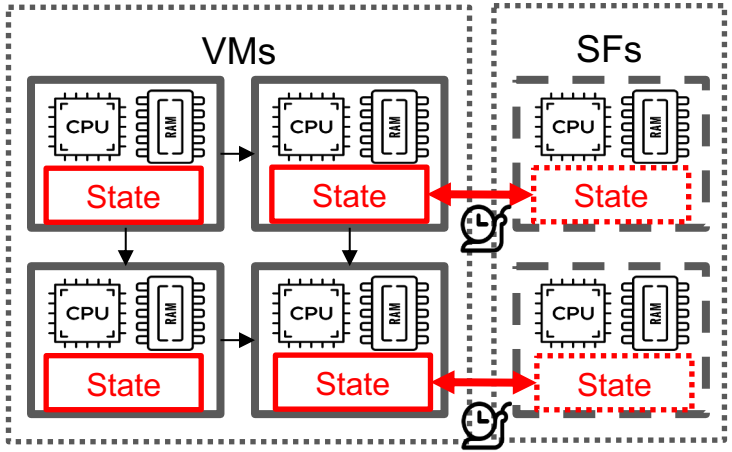
to provide stable, direct network connections

Managed Runtime Initialization Overhead



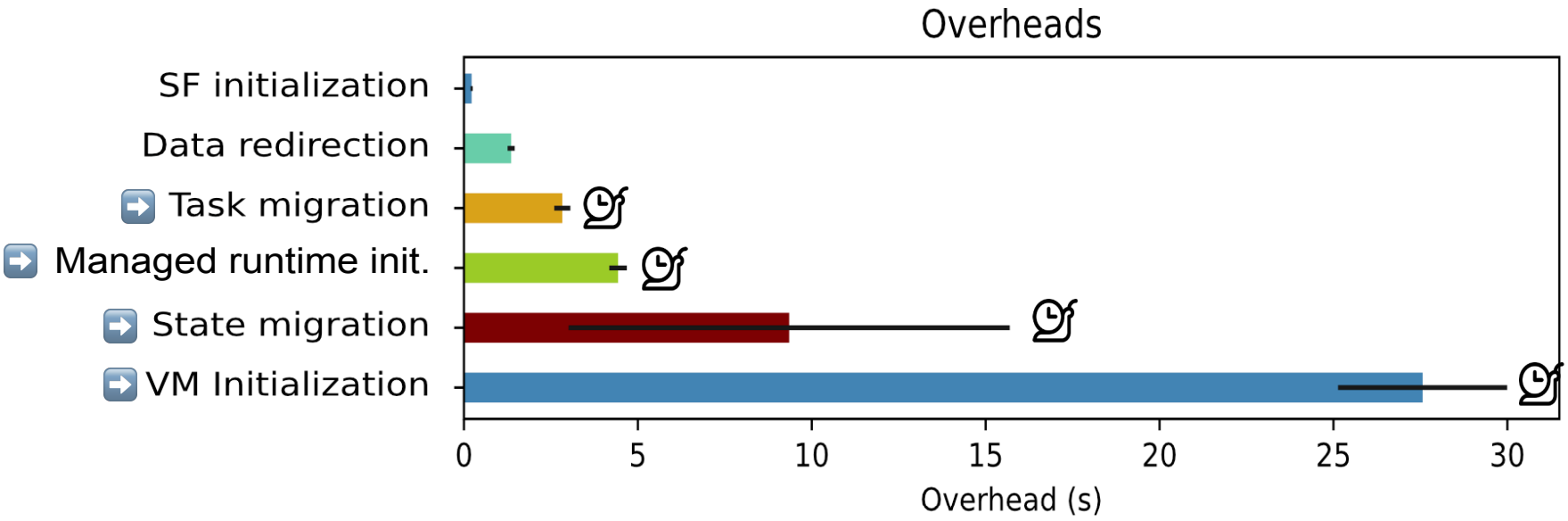
Managed runtimes (e.g., JVM) incur launch overheads (~4 seconds)

State & Task Migration Overhead



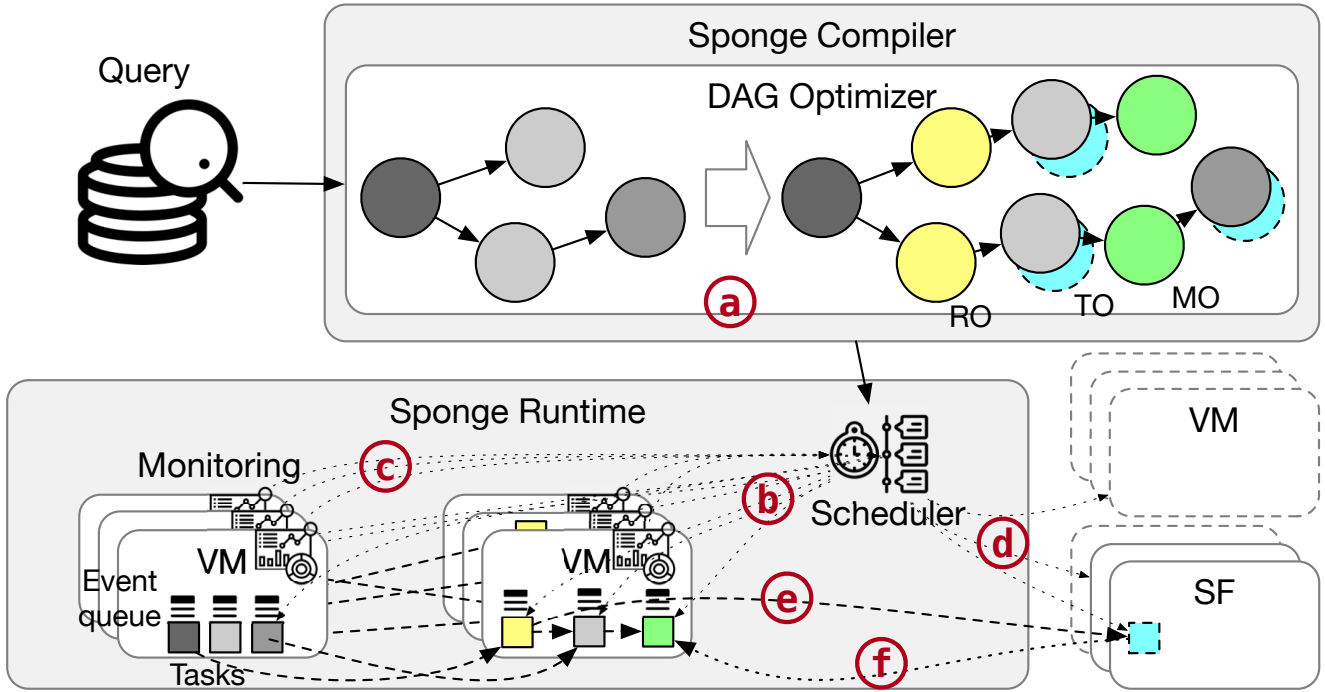
State & task migration overheads are not negligible due to smaller network bandwidths

Challenges and Overheads to Overcome



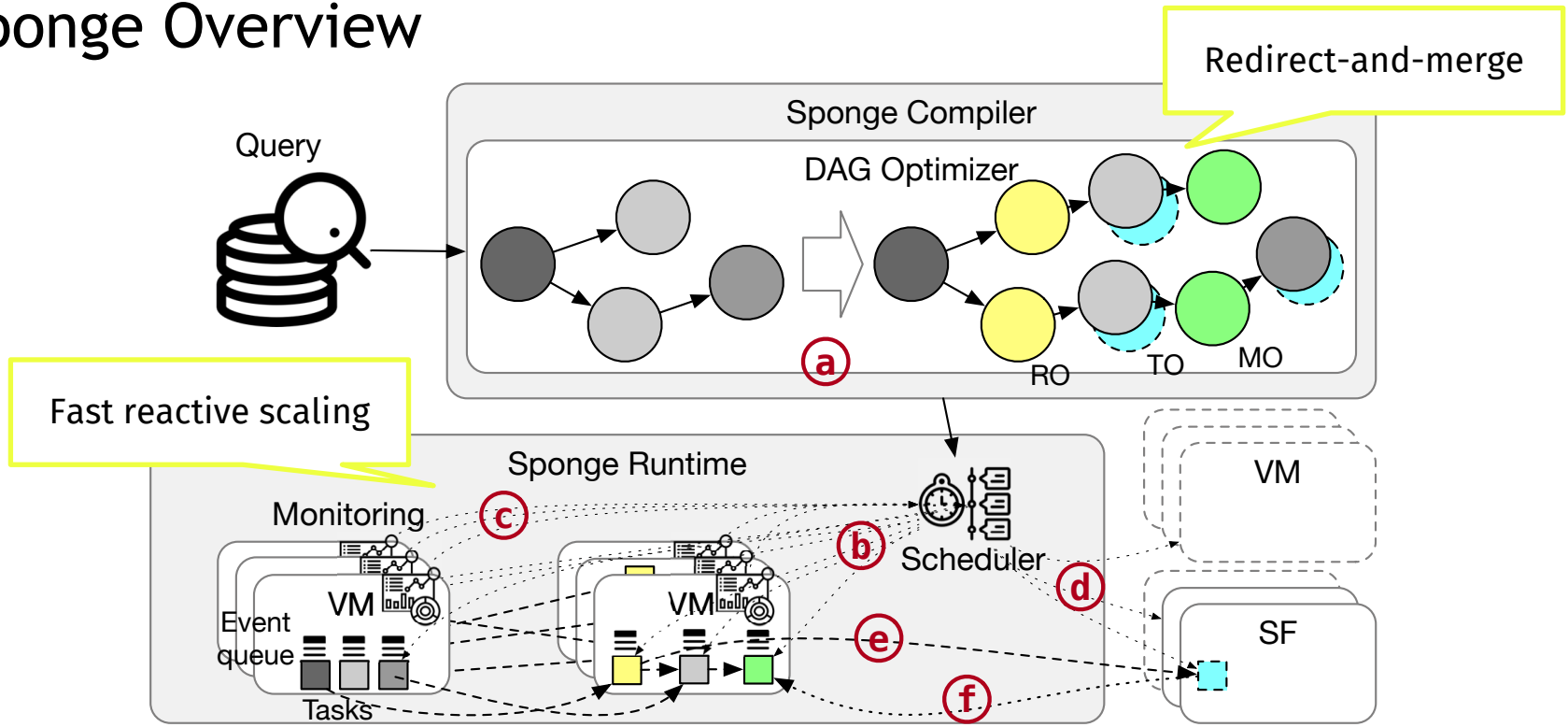
Various overheads exist for using VMs and serverless instances

Sponge Overview



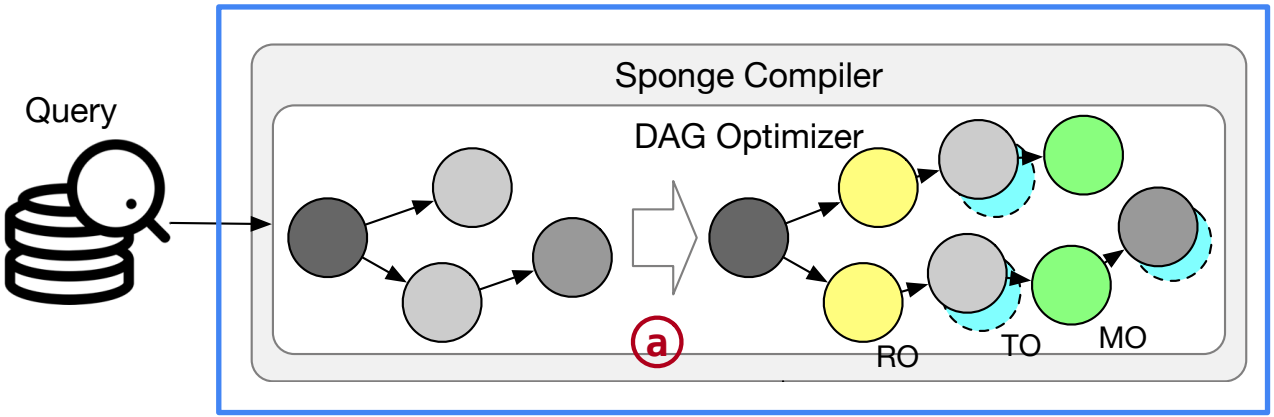
Sponge handles the challenges through compile-time and run-time

Sponge Overview



Sponge handles the challenges through compile-time and run-time

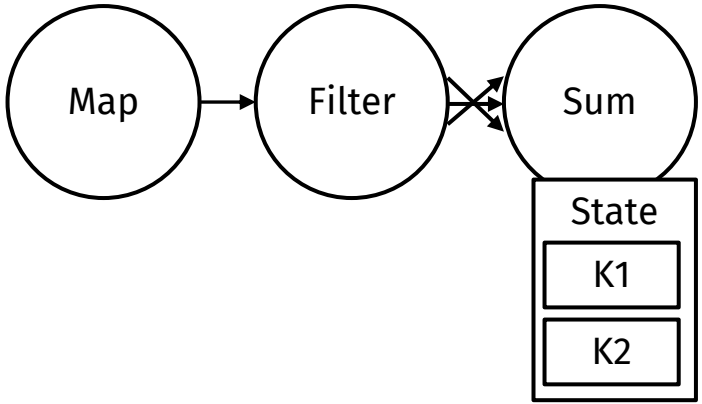
Injecting New Operators during Compile-Time



Query DAG is inserted with new operators at compile-time with ~200ms overhead

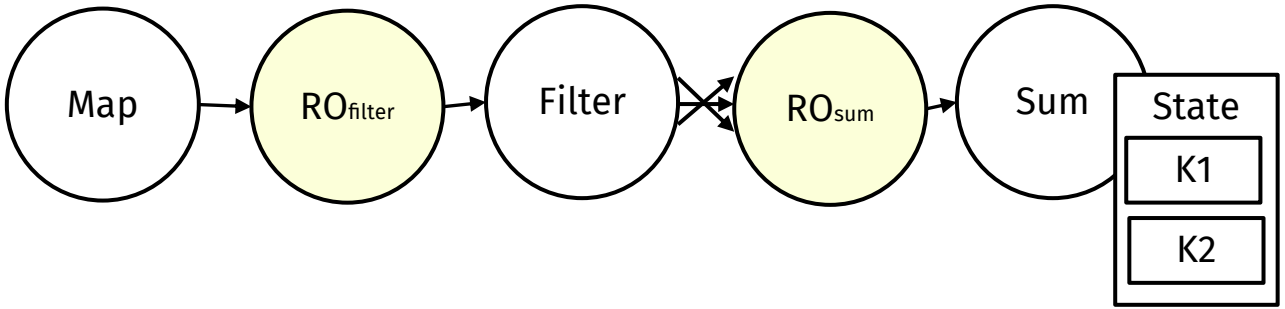
Injecting New Operators during Compile-Time

1. Router operators (ROs) enable redirection of input events to specific instances
2. Transient operators (TOs) enable execution of cloned operators on SFs
3. Merge operators (MOs) enable merges on partial states



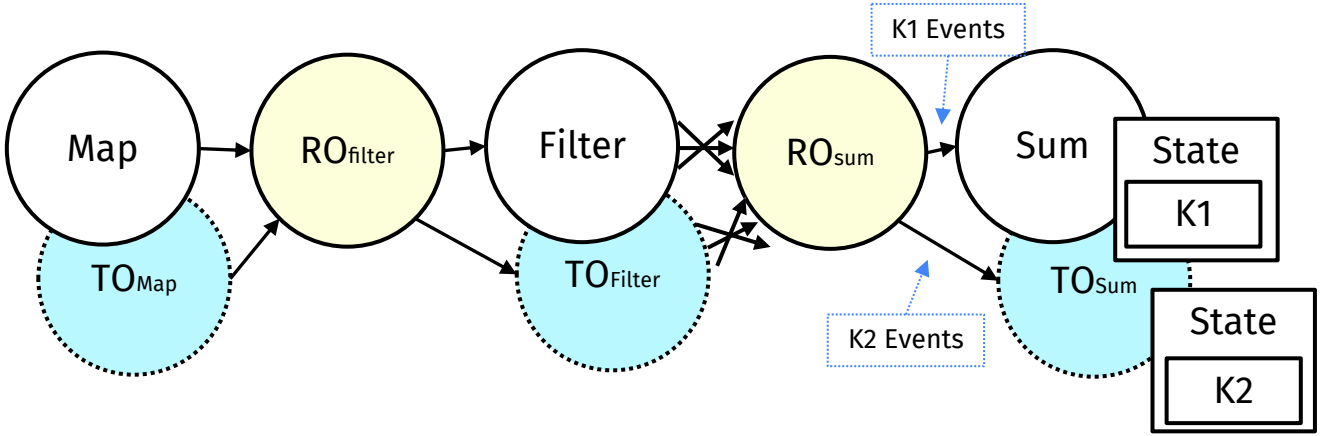
Injecting New Operators during Compile-Time

1. **Router operators (ROs) enable redirection of input events to specific instances**
2. Transient operators (TOs) enable execution of cloned operators on SFs
3. Merge operators (MOs) enable merges on partial states



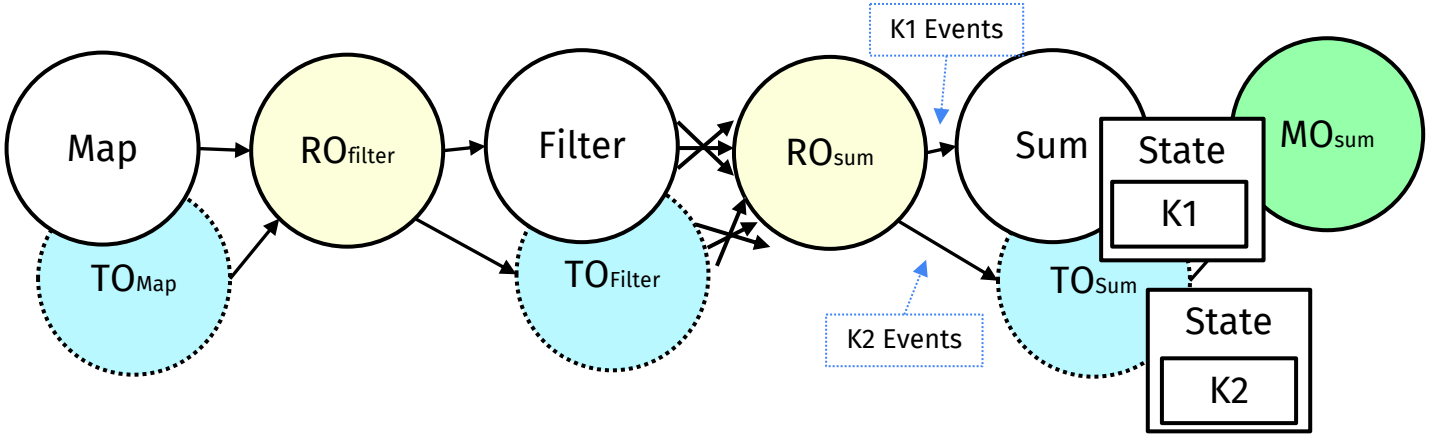
Injecting New Operators during Compile-Time

1. Router operators (ROs) enable redirection of input events to specific instances
2. **Transient operators (TOs) enable execution of cloned operators on SFs**
3. Merge operators (MOs) enable merges on partial states



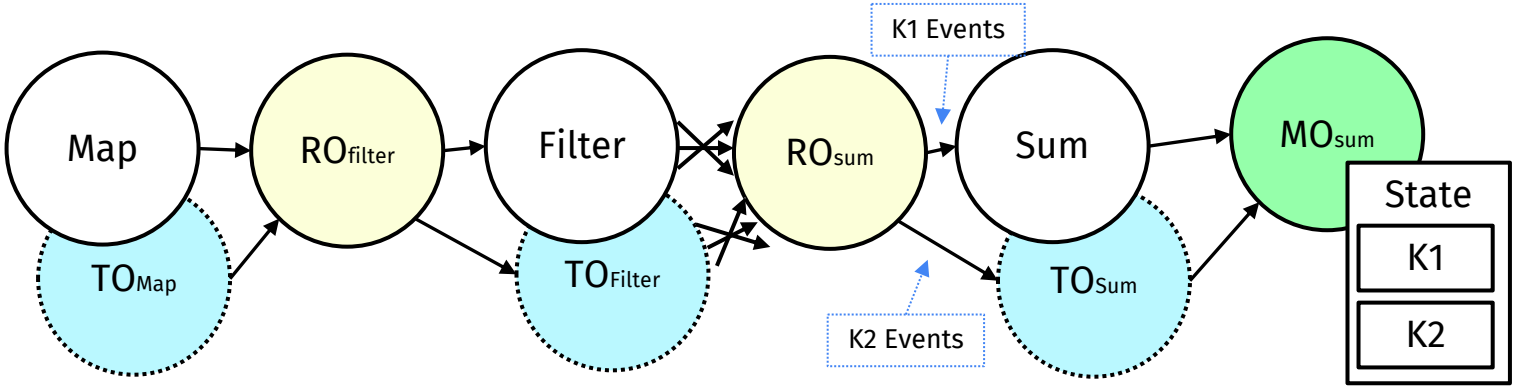
Injecting New Operators during Compile-Time

1. Router operators (ROs) enable redirection of input events to specific instances
2. Transient operators (TOs) enable execution of cloned operators on SFs
3. **Merge operators (MOs) enable merges on partial states**

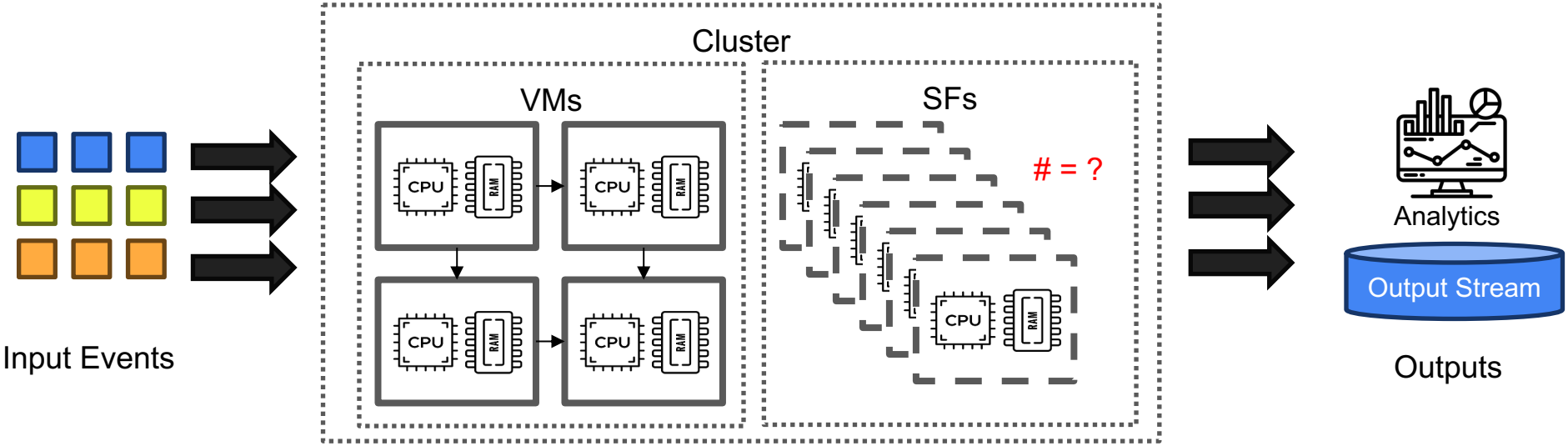


Injecting New Operators during Compile-Time

1. Router operators (ROs) enable redirection of input events to specific instances
2. Transient operators (TOs) enable execution of cloned operators on SFs
3. **Merge operators (MOs) enable merges on partial states**



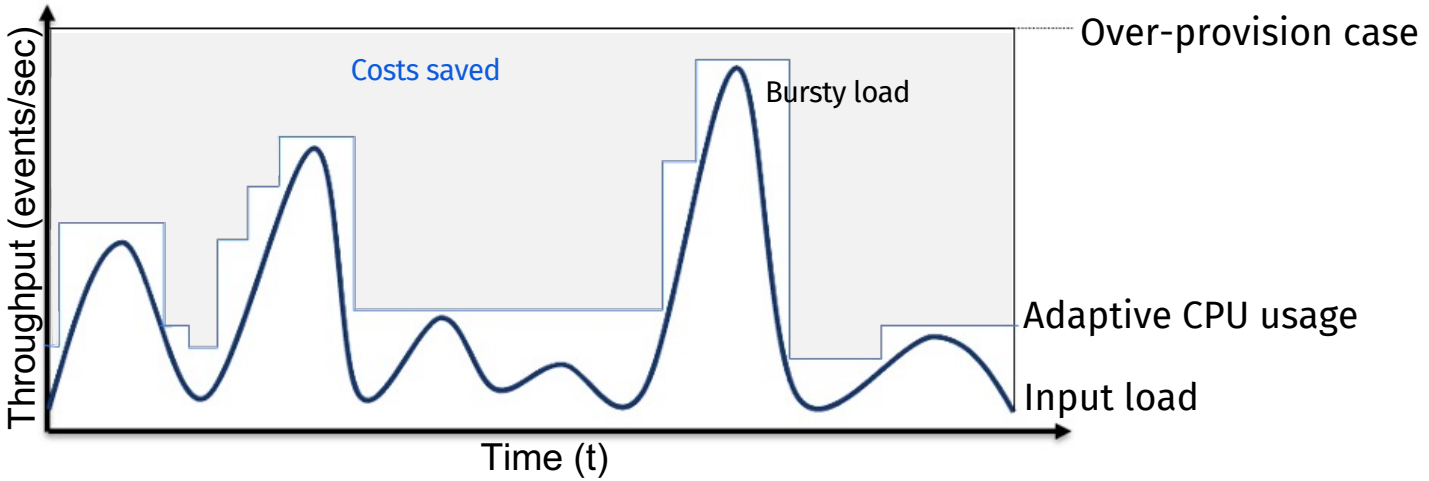
How Much Resources are Required for Our Pipeline?



How much data should we redirect to serverless functions?

How many serverless instances should we be using?

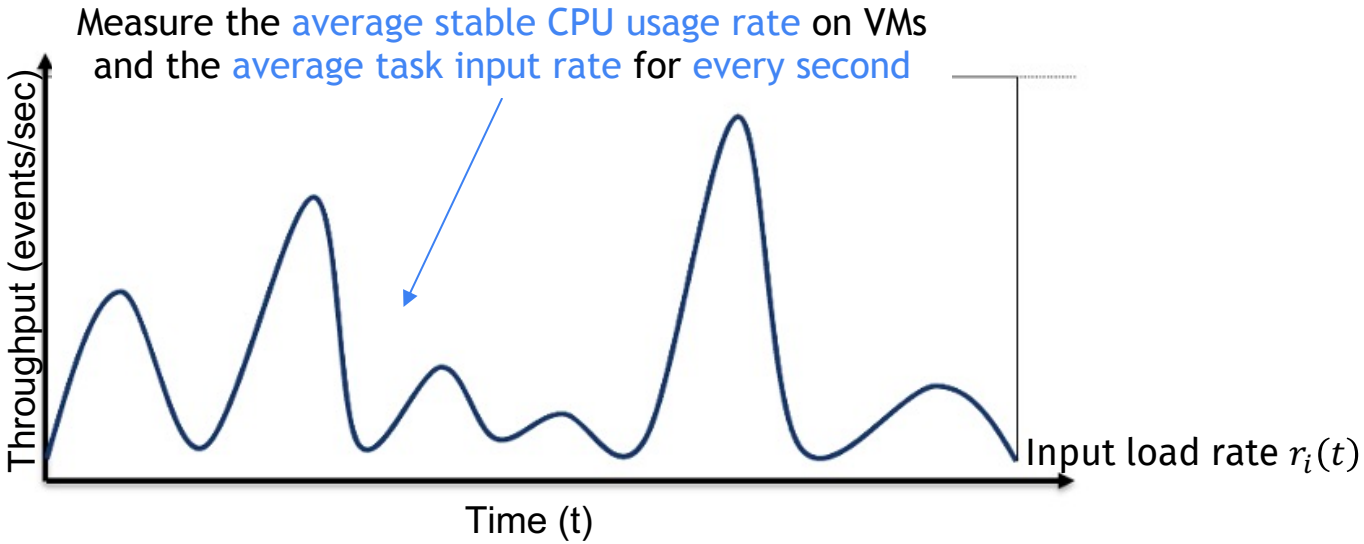
Dynamic Resource Management during Runtime

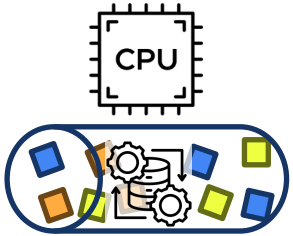


CPU and input rates are monitored every second (~10ms overhead)

CPU utilization goal: 60-80%

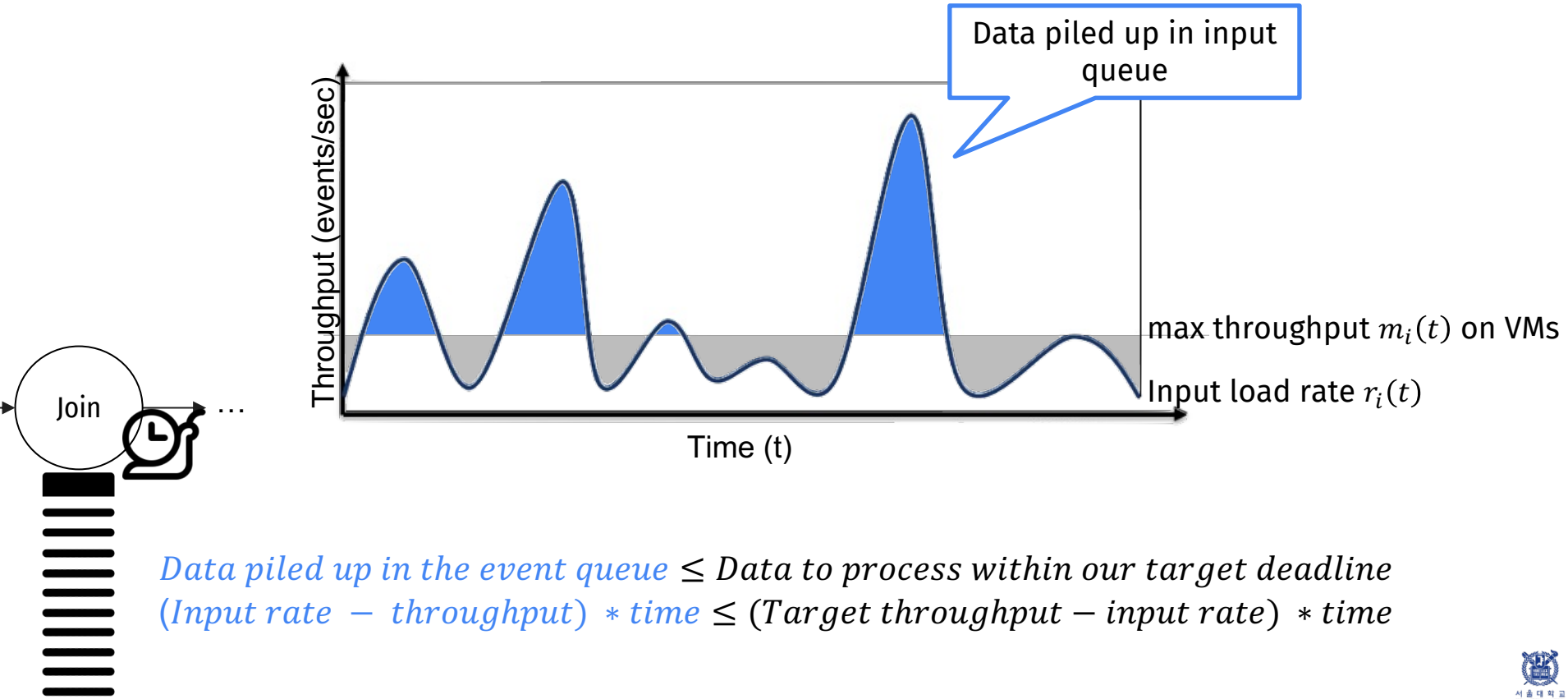
Stable Input Load per CPU Core





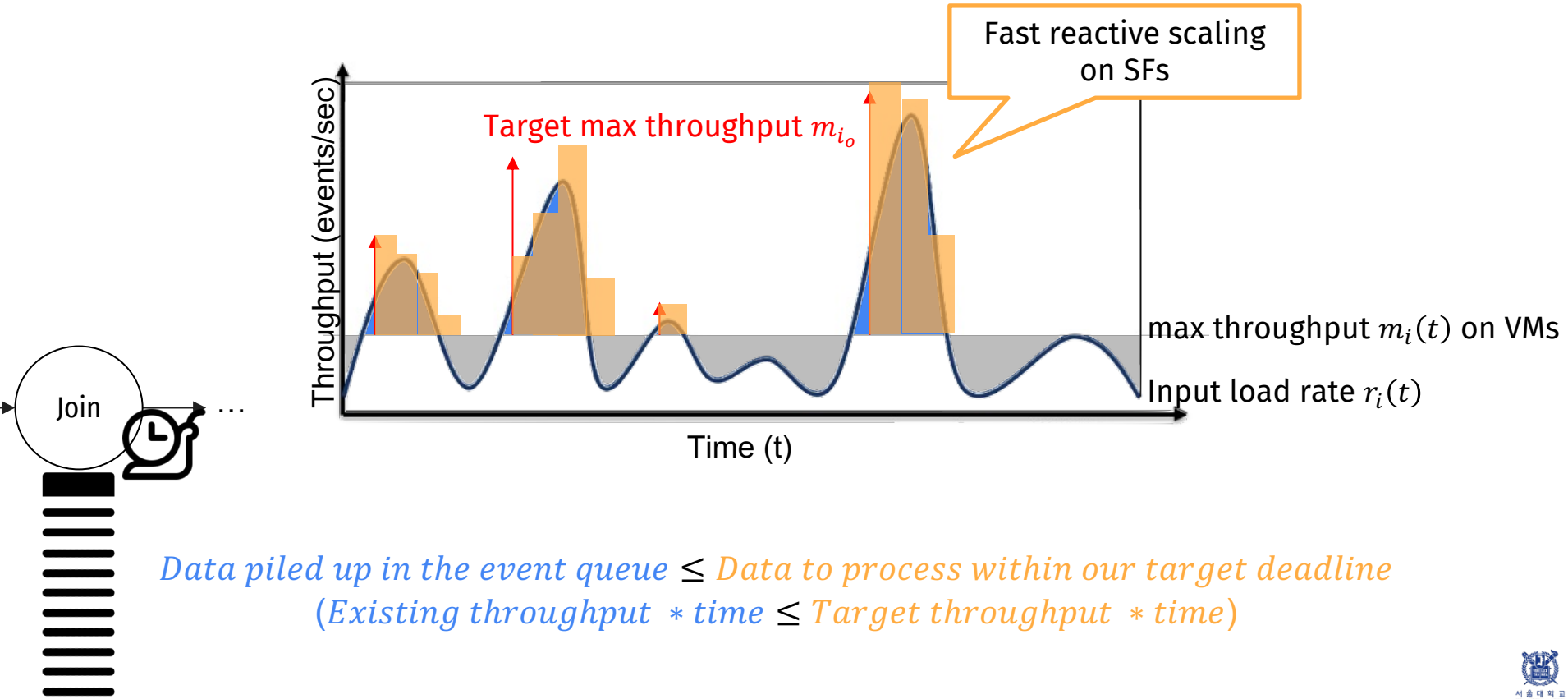
$$Approx.\,throughput\ rate\ per\ VM\ core = \frac{\overline{input\ rate}}{number\ of\ VM\ cores}$$

Recovery Deadline and Target Throughput



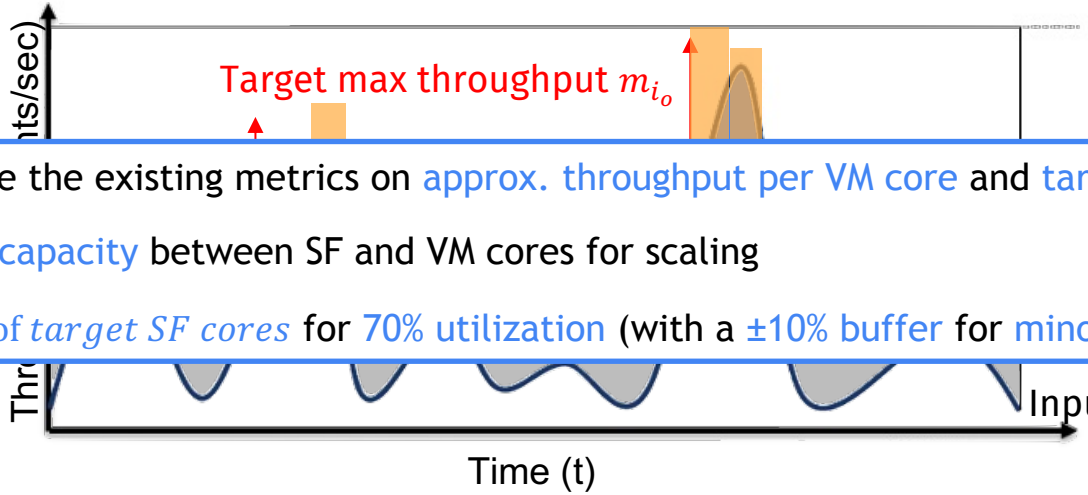
$$Data\ piled\ up\ in\ the\ event\ queue \leq Data\ to\ process\ within\ our\ target\ deadline$$
$$(Input\ rate\ -\ throughput)\ *time \leq (Target\ throughput - input\ rate)\ *time$$

Recovery Deadline and Target Throughput

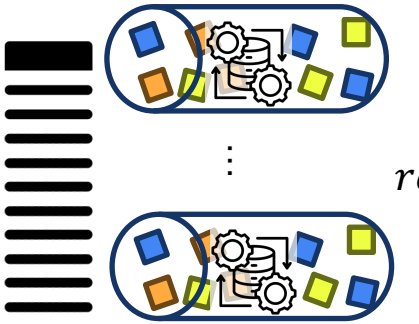


$$Data\ piled\ up\ in\ the\ event\ queue \leq Data\ to\ process\ within\ our\ target\ deadline$$
$$(Existing\ throughput\ * time \leq Target\ throughput\ * time)$$

Preparing SFs to Reduce Runtime Launch Overhead

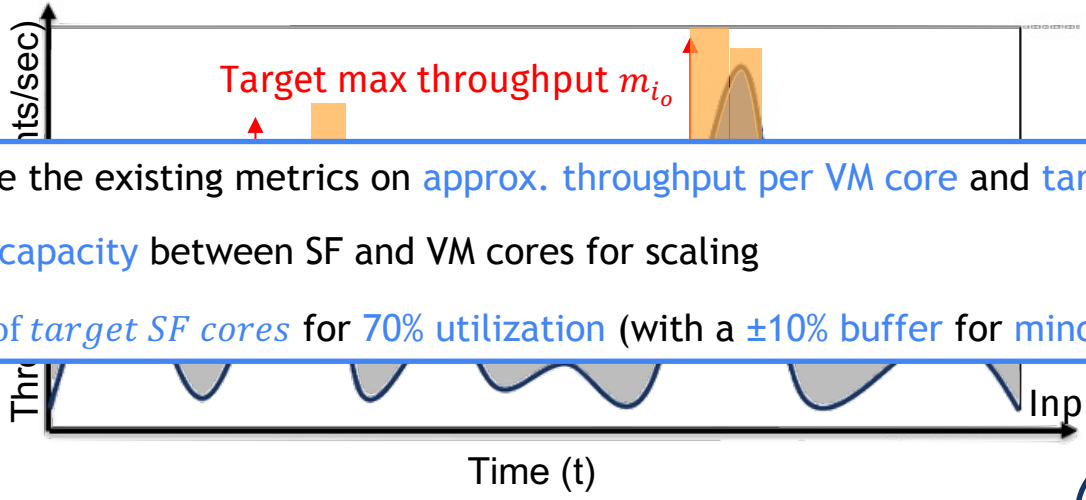


1. Retrieve the existing metrics on approx. throughput per VM core and target throughput
2. Profile capacity between SF and VM cores for scaling
3. Find # of *target SF cores* for 70% utilization (with a ±10% buffer for minor profiling errors)



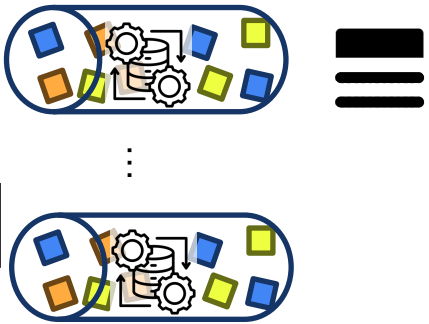
$$required\ SF\ cores = \left\lceil \frac{Target\ additional\ throughput}{70\% * Approx.\ throughput\ per\ SF\ core} \right\rceil$$

Preparing SFs to Reduce Runtime Launch Overhead

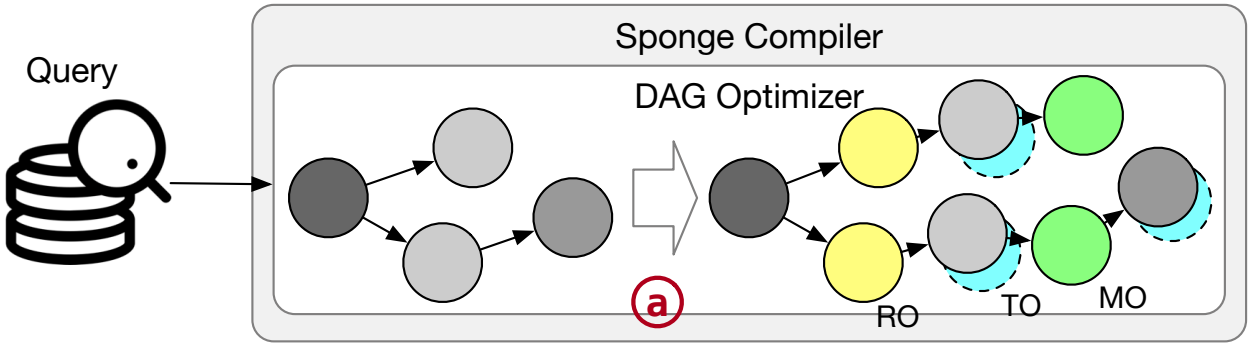


1. Retrieve the existing metrics on approx. throughput per VM core and target throughput
2. Profile capacity between SF and VM cores for scaling
3. Find # of *target SF cores* for 70% utilization (with a ±10% buffer for minor profiling errors)

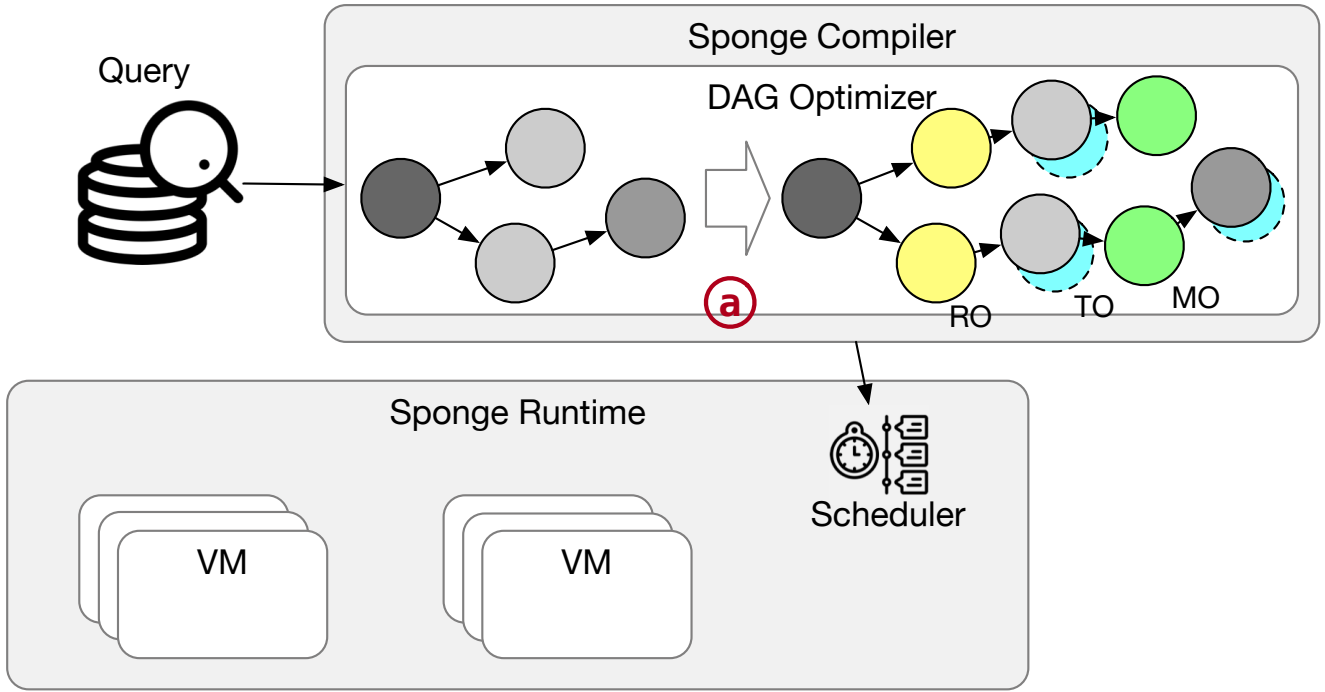
$$required\ SF\ cores = \left\lceil \frac{Target\ additional\ throughput}{70\% * Approx.throughput\ per\ SF\ core} \right\rceil$$



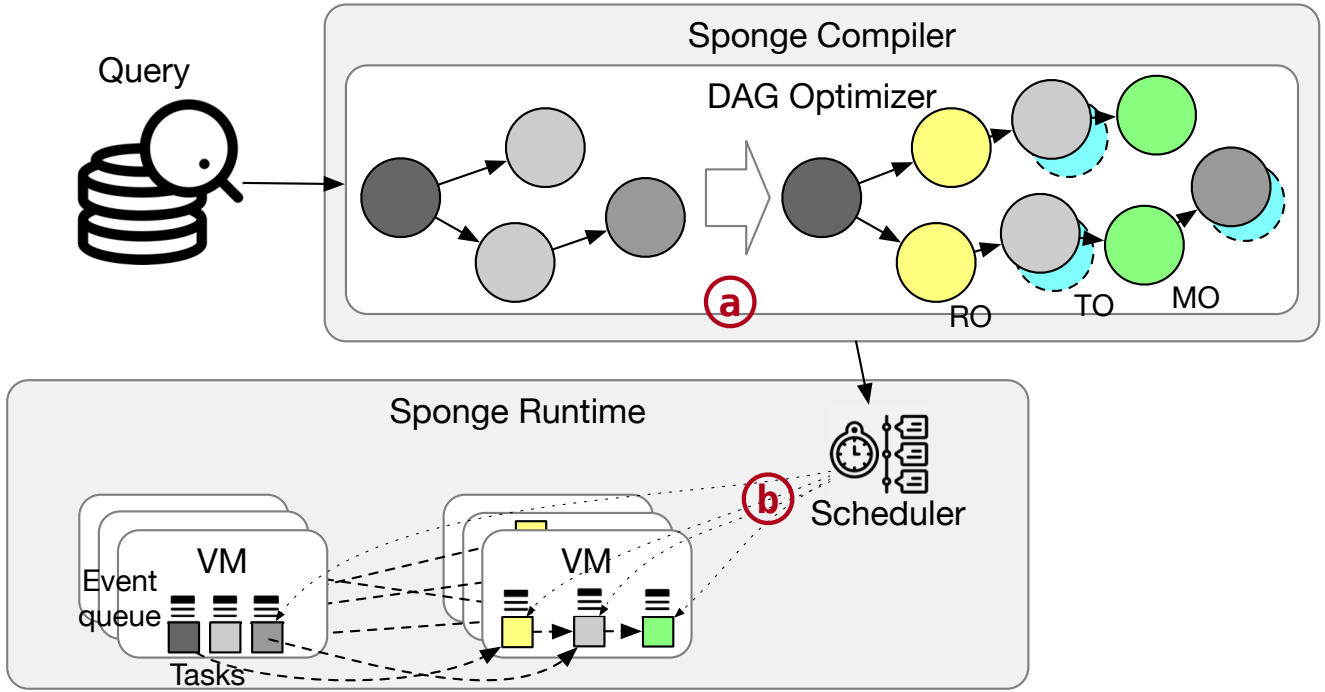
Sponge In Action



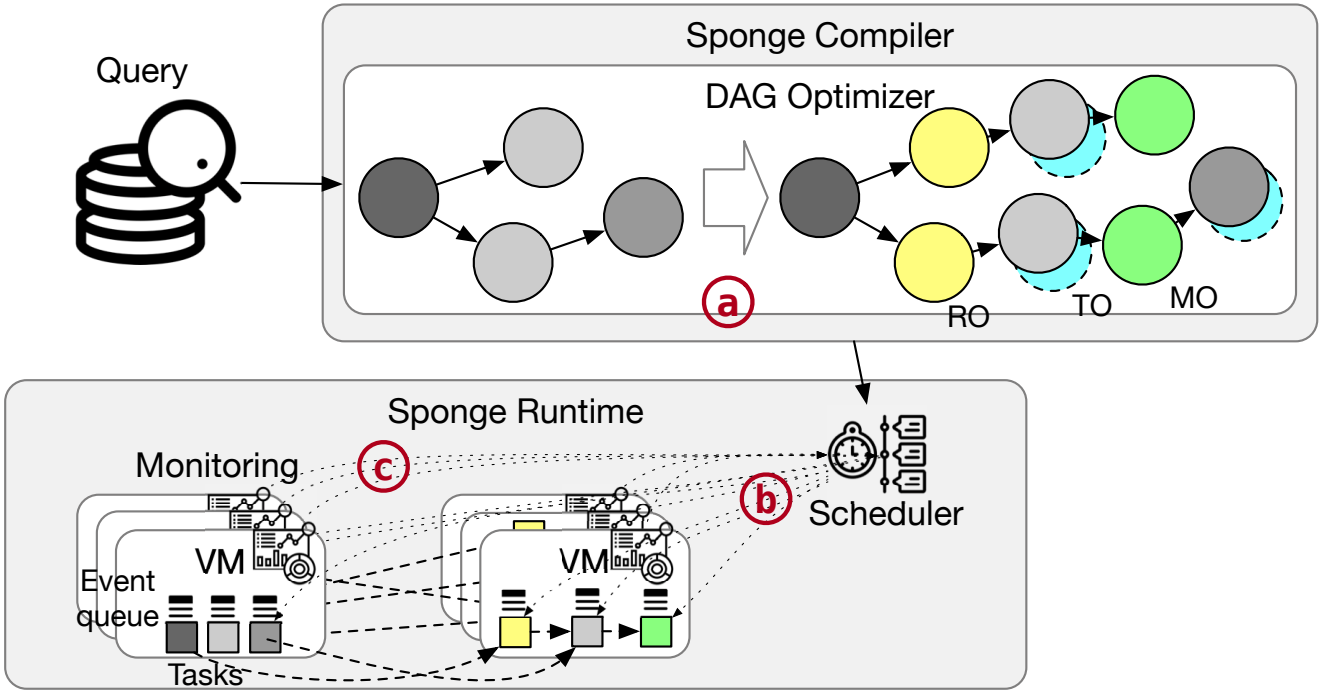
Sponge In Action



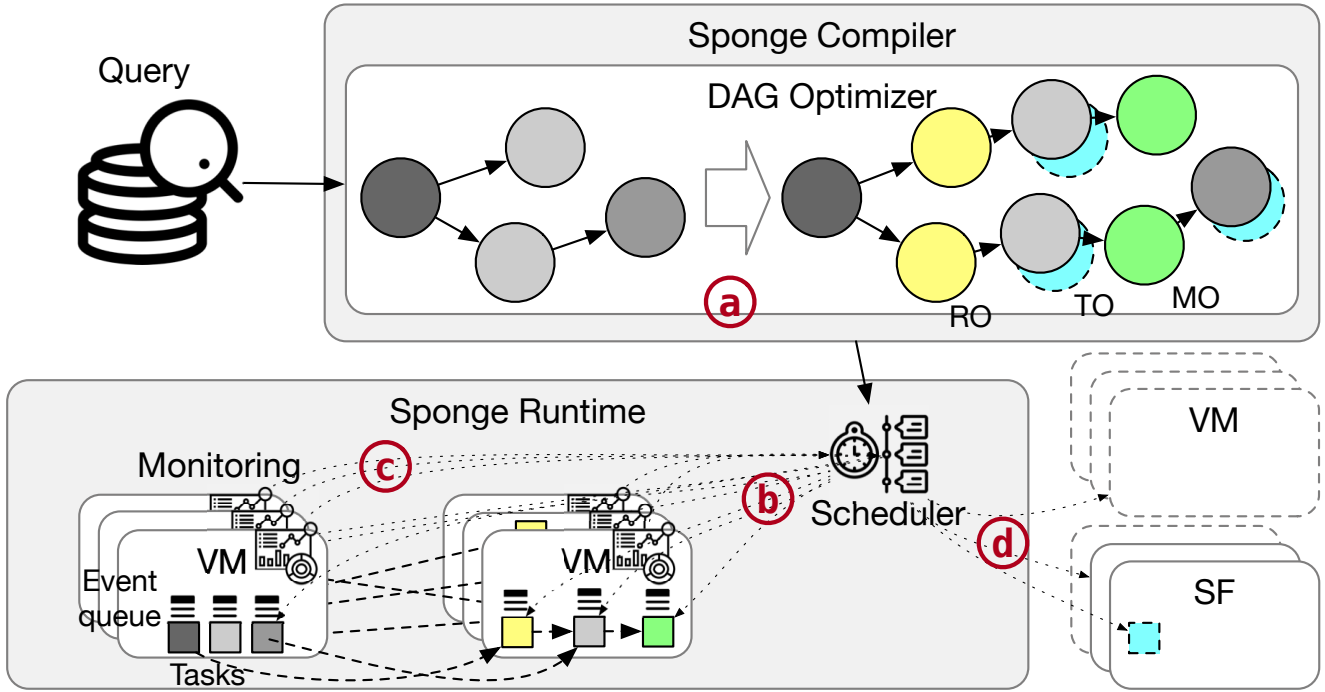
Sponge In Action



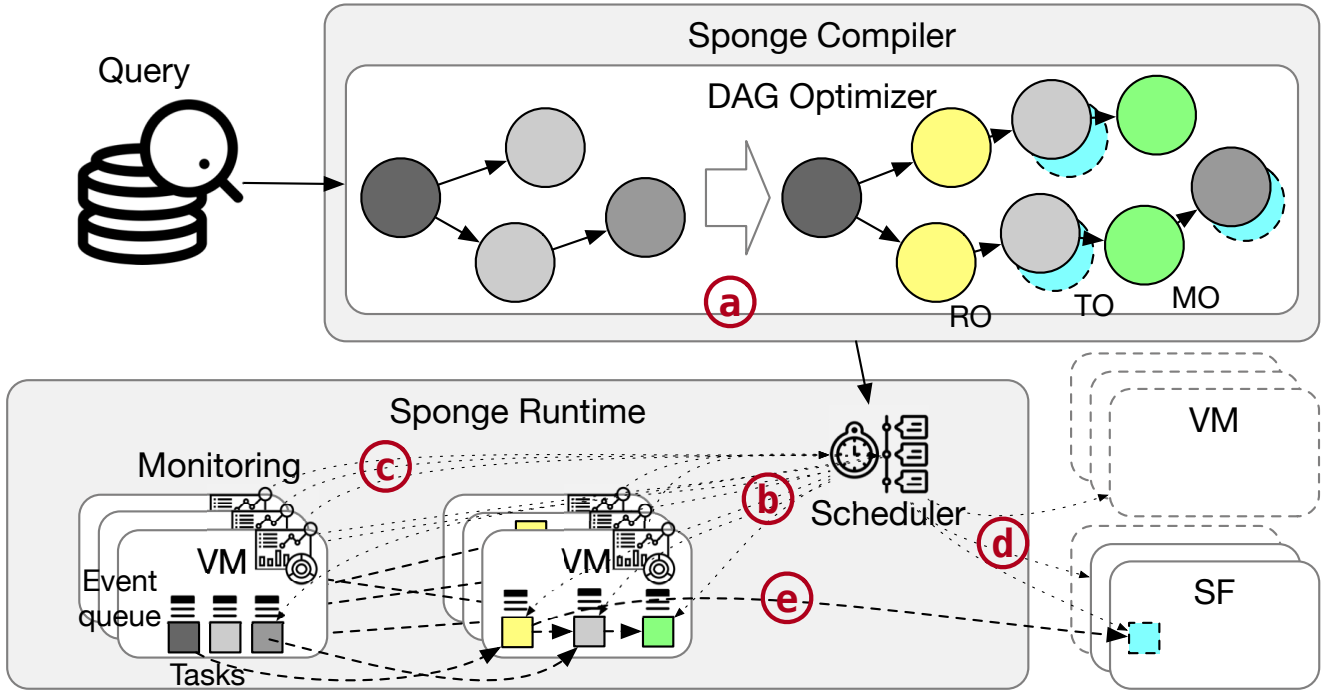
Sponge In Action



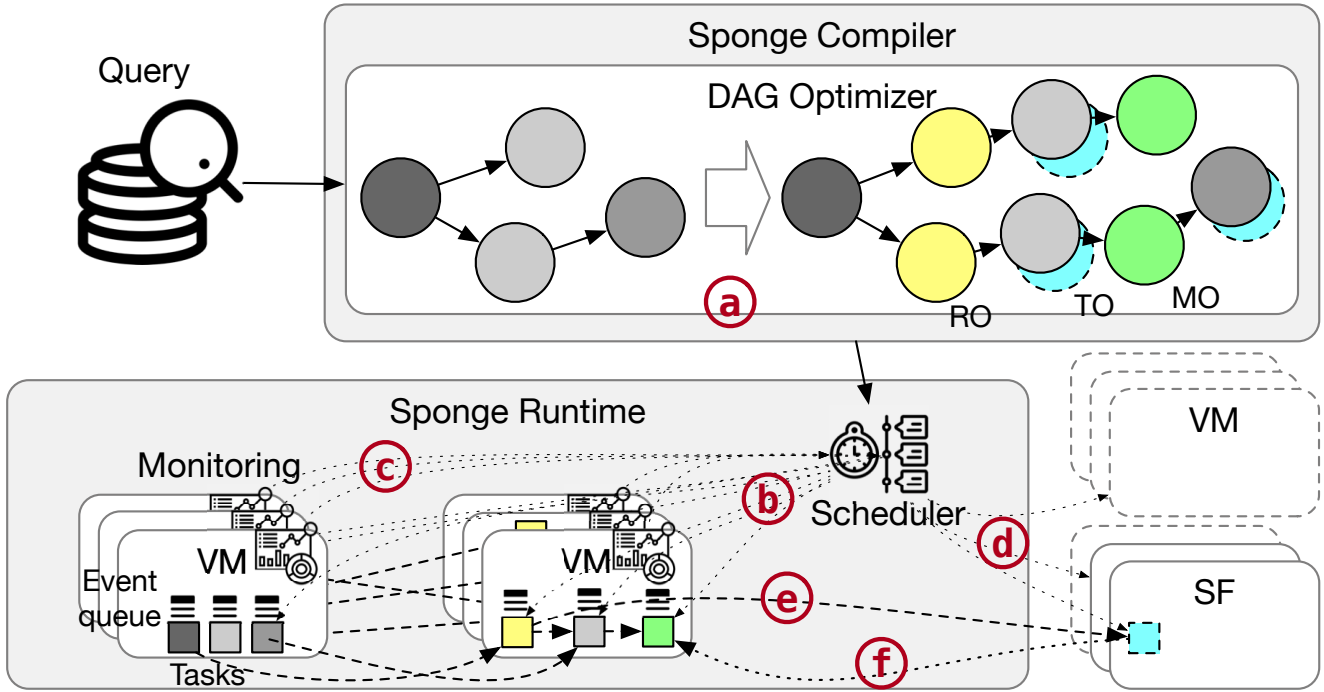
Sponge In Action



Sponge In Action



Sponge In Action

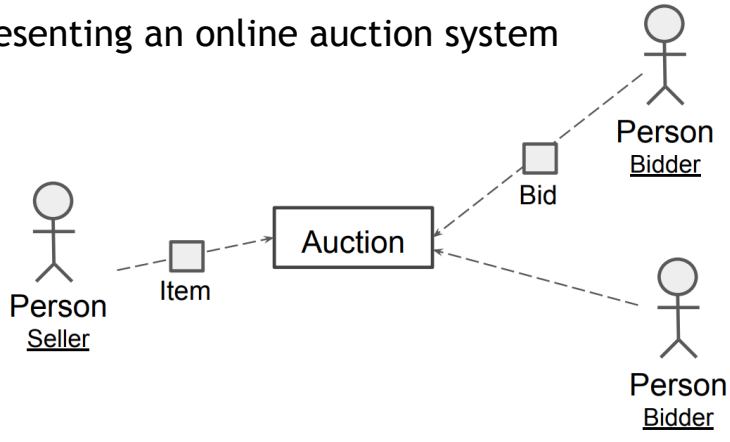


Sponge Implementation

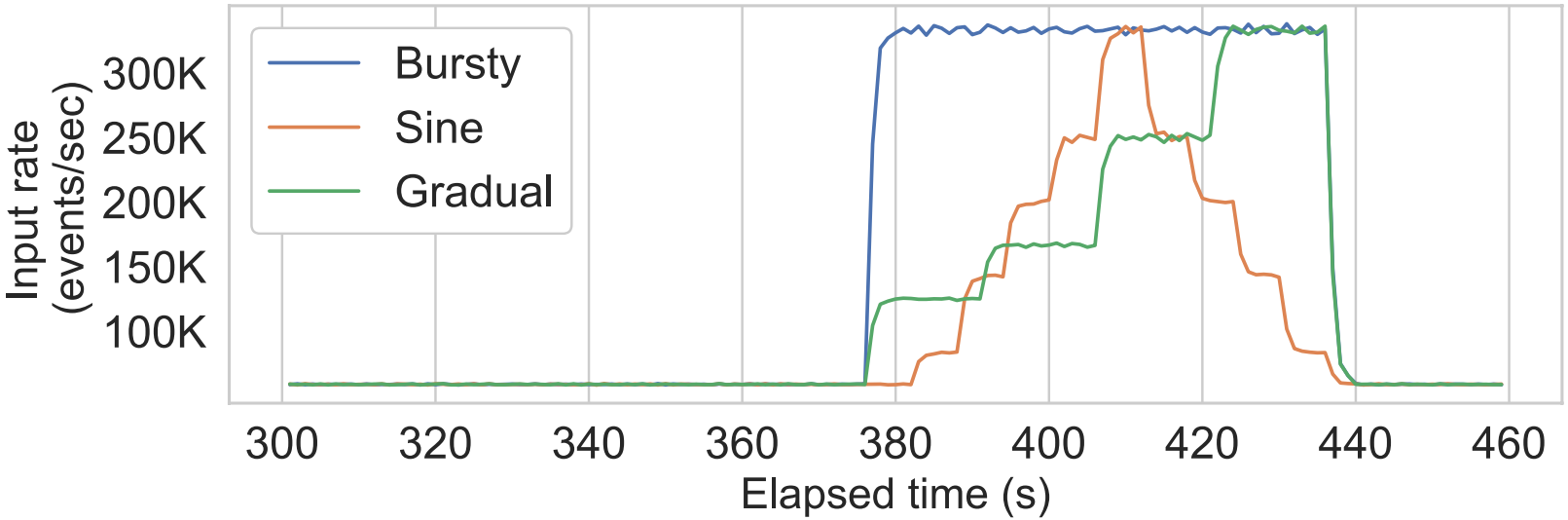
- Programming interface: Apache Beam
  - Associative and commutative operators are extracted to implement the merge operators
- DAG reshaping mechanisms & data processing runtime: Apache Nemo
  - Operator insertion can be expressed as reusable algorithms
- Serverless frameworks: AWS Lambda
- Managing & deploying different instances: boto3
  - AWS SDK API for controlling AWS instances

Evaluation Results

- AWS Cluster of *5 nodes* for execution + *1 large node* for data generation
  - r5.xlarge (4vCPUs, 32GB Memory) * 5
  - c5d.12xlarge for data generation (48vCPUs, 96GB Memory) * 1
  - 1769MB AWS Lambda instances (1769MB offers instance with 1 vCore) * up to 200
- NEXMark Benchmark Suite
  - A suite of pipelines, provided by Apache Beam, representing an online auction system
  - *Queries* include
    - *1 (currency conversion)*
    - *4, **6** (avg. price per category, avg. price by seller)*
    - *5, 7 (hot items, highest bid)*
    - ***8** (monitor new users)*

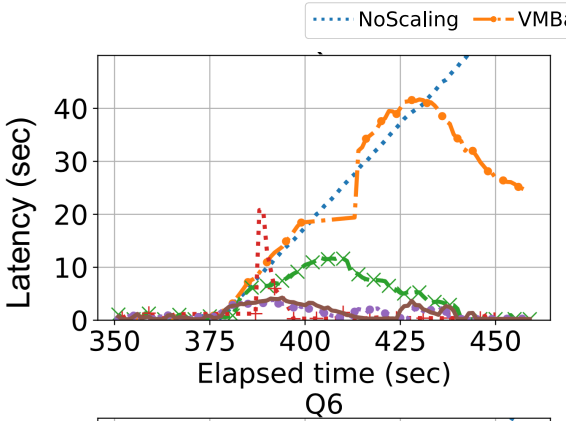


Evaluation Results: Input Patterns



3 different input patterns with different burstiness and duration

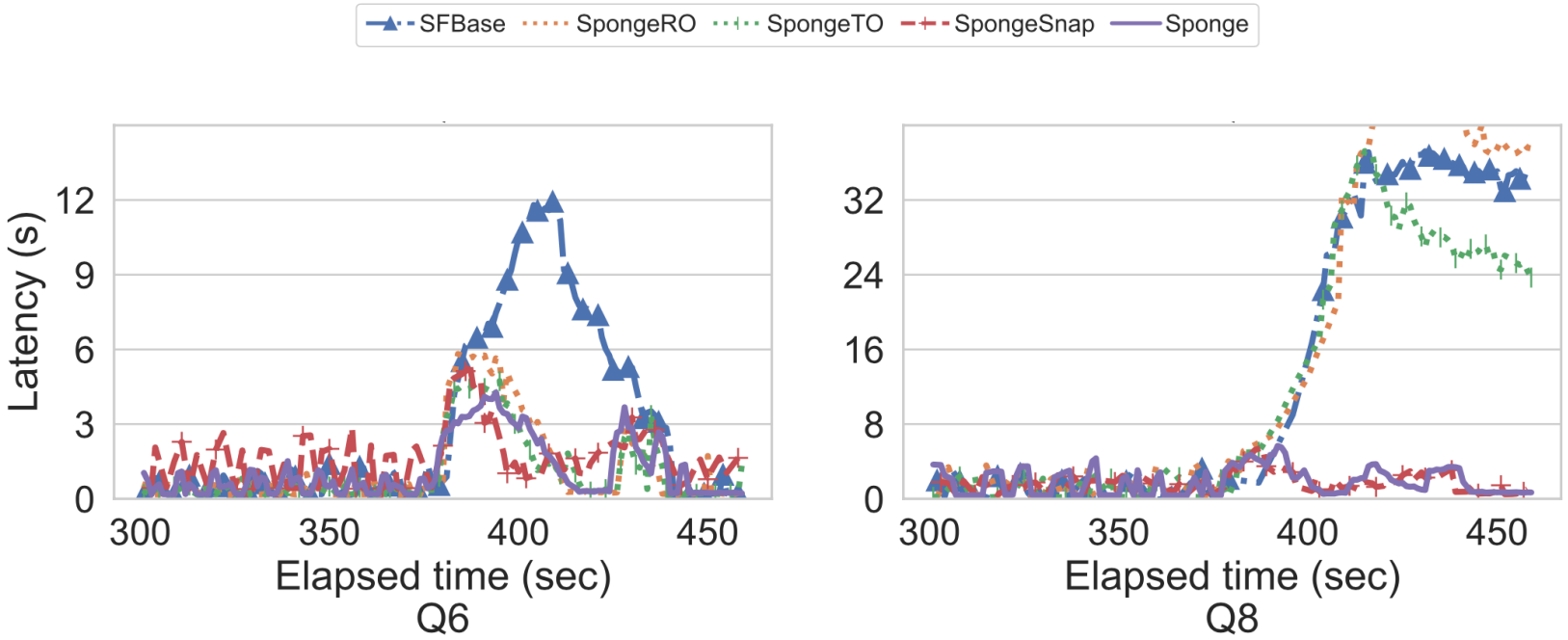
Evaluation Results: Latency and CPU Utilizations



Latency for 5x burst at 380s

CPU Utilizations for 5x burst at 380s

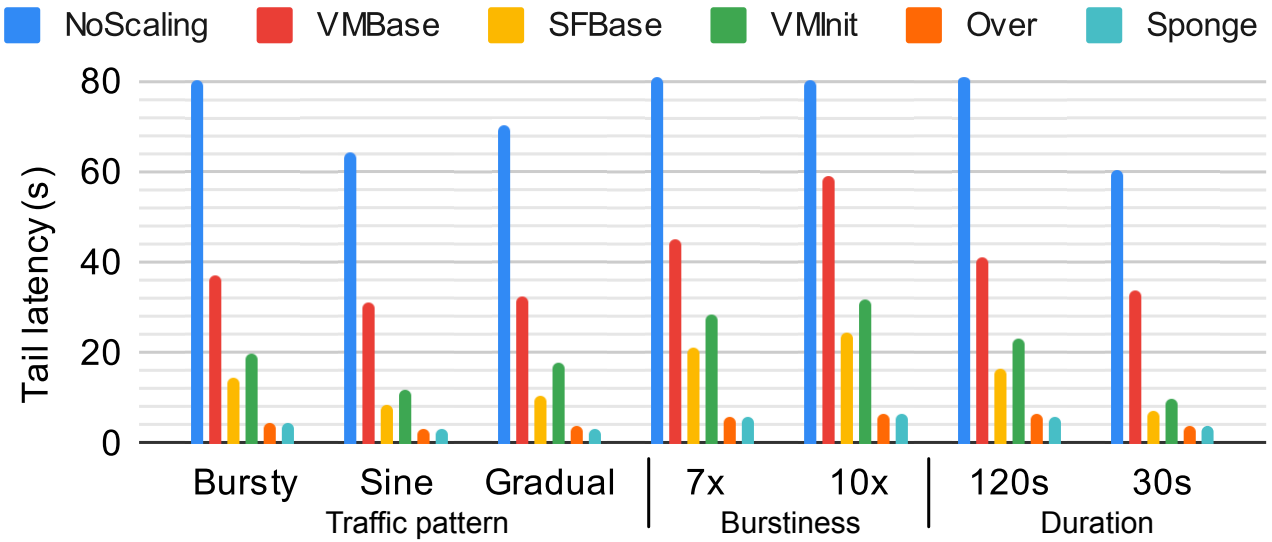
Sponge Evaluation: Performance Breakdown



Performance breakdown of different Sponge components

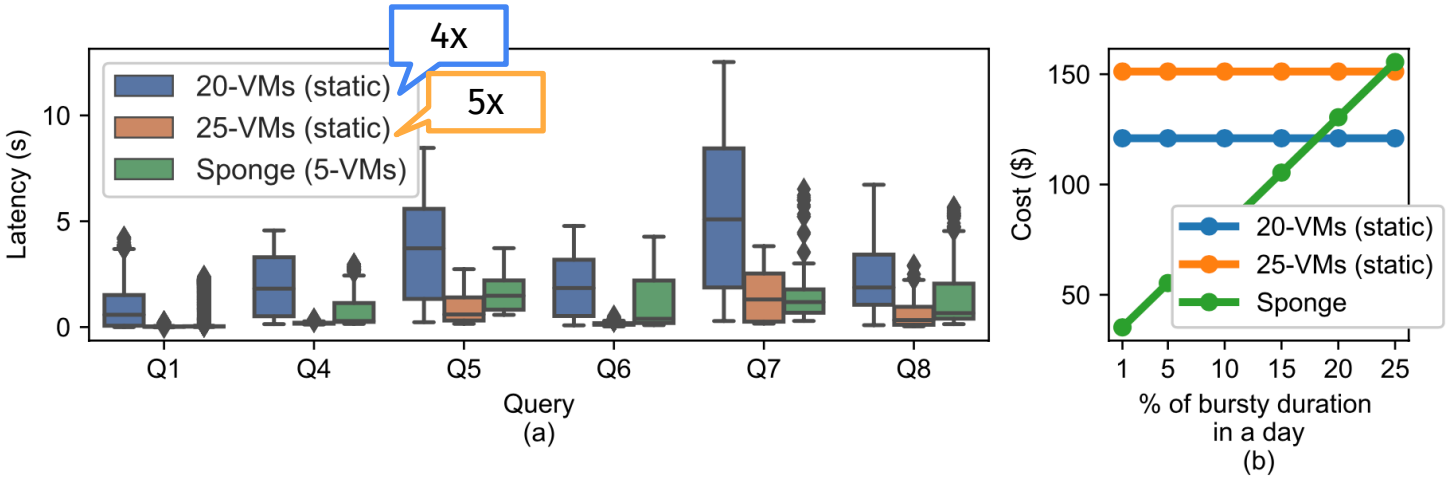
+ Router operators → + Transient operators → + Merge operators

Evaluation Results: Different Input Patterns



Tail latency for different patterns, burstiness, and durations

Evaluation Results: Cost Analysis



(a) Sponge effectively keeps latencies low compared to over-provisioned solutions

(b) Bursty duration falls below 15% of total time, making Sponge cost-effective

Conclusion

- **Bursts of input events** → input data to piles up in the input queue

- Sponge prevents **launch** and **migration overheads**
  - By **redirecting** bursts of input data to **fast-starting serverless frameworks**
  - SFs are **automatically scaled** to keep **latencies** and **budget** within our target

- Sponge reduces **tail latencies** by **88%** on average vs. VM scaling
- Sponge reduces **cost** to **17% (83% reduction)** vs. over-provisioning

Thank you!
Questions?

Won Wook SONG

https://wonook.github.io

wonook@apache.org

Sponge:
Fast Reactive Scaling for Stream Processing with Serverless Frameworks

Won Wook SONG, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, Byung-Gon Chun

Seoul National University, Samsung Research, Microsoft Research, UNIST, FriendliAI