Retro

Targeted Resource Management in Multi-tenant Distributed Systems

Jonathan Mace
Brown University

Peter Bodik
MSR Redmond

Rodrigo Fonseca
Brown University

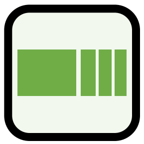
Madanlal Musuvathi
MSR Redmond

BROWN UNIVERSITY

Microsoft Research

Resource Management in Multi-Tenant Systems

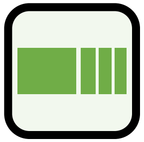
Resource Management in Multi-Tenant Systems



Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones

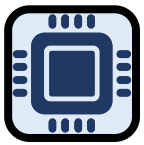
- April 2011 – Amazon EBS Failure

Resource Management in Multi-Tenant Systems



Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones

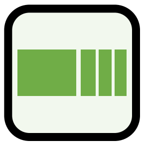
- April 2011 – Amazon EBS Failure



Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging

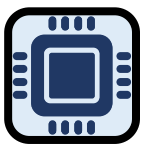
- Aug. 2012 – Azure Storage Outage

Resource Management in Multi-Tenant Systems



Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones

- April 2011 – Amazon EBS Failure



Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging

- Aug. 2012 – Azure Storage Outage

Code change increases database usage, shifts bottleneck to unmanaged **application-level lock**

- Nov. 2014 – Visual Studio Online outage

Resource Management in Multi-Tenant Systems

Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones

- April 2011 – Amazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging

- Aug. 2012 – Azure Storage Outage

Code change increases database usage, shifts bottleneck to unmanaged **application-level lock**

- Nov. 2014 – Visual Studio Online outage

Shared storage layer bottlenecks circumvent resource management layer

- 2014 – Communication with Cloudera

Resource Management in Multi-Tenant Systems

Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones

- April 2011 – Amazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging

- Aug. 2012 – Azure Storage Outage

Code change increases database usage, shifts bottleneck to unmanaged **application-level lock**

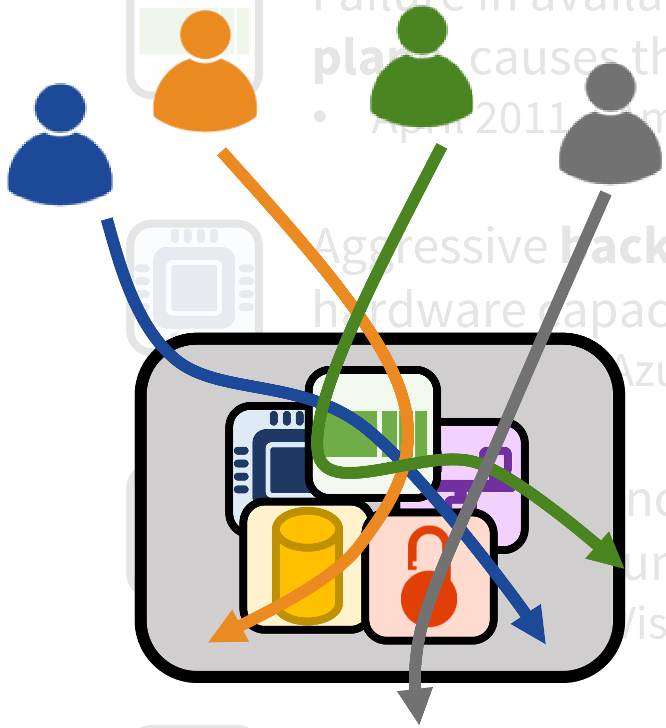
- Nov. 2014 – Visual Studio Online outage

Shared storage layer bottlenecks circumvent resource management layer

- 2014 – Communication with Cloudera

Degraded performance, Violated SLOs, system outages

Resource Management in Multi-Tenant Systems



Failure in availability zone **cascades to shared control pla**[...] causes thread pool starvation for all zones

- [...] 2011 [...] mazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging

- [...] Azure Storage Outage

[...] ncreases database usage, shifts [...] unmanaged **application-level lock**

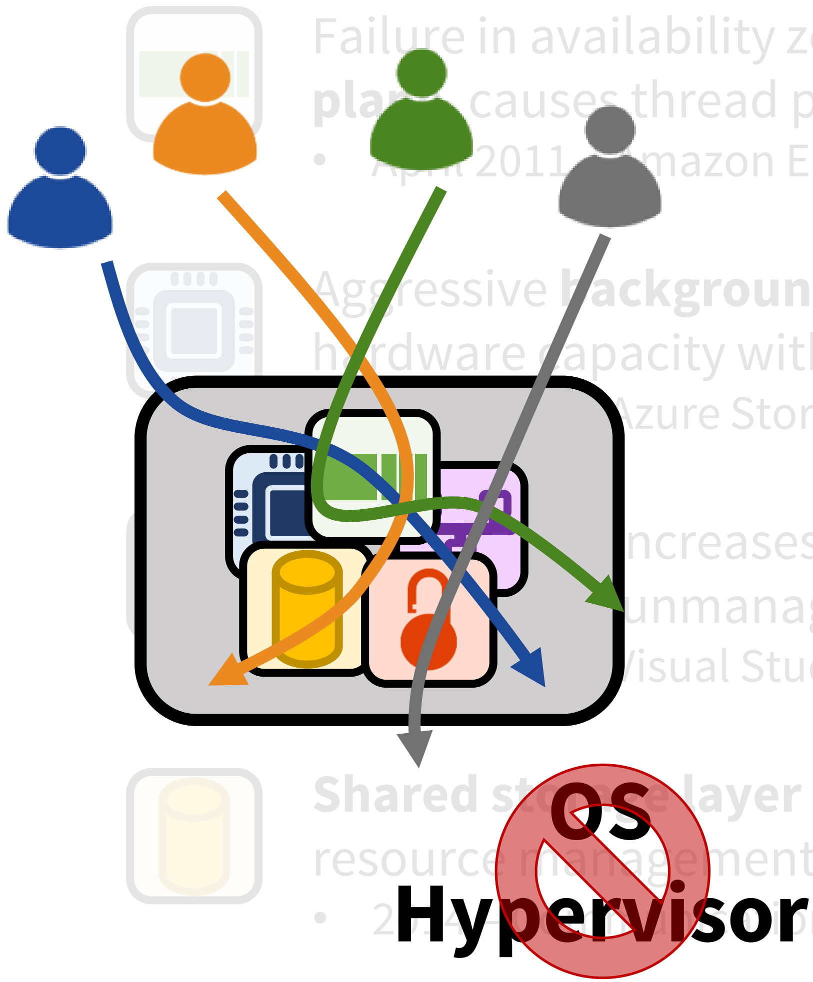
- [...] isual Studio Online outage

Shared storage layer bottlenecks circumvent resource management layer

- 2014 – Communication with Cloudera

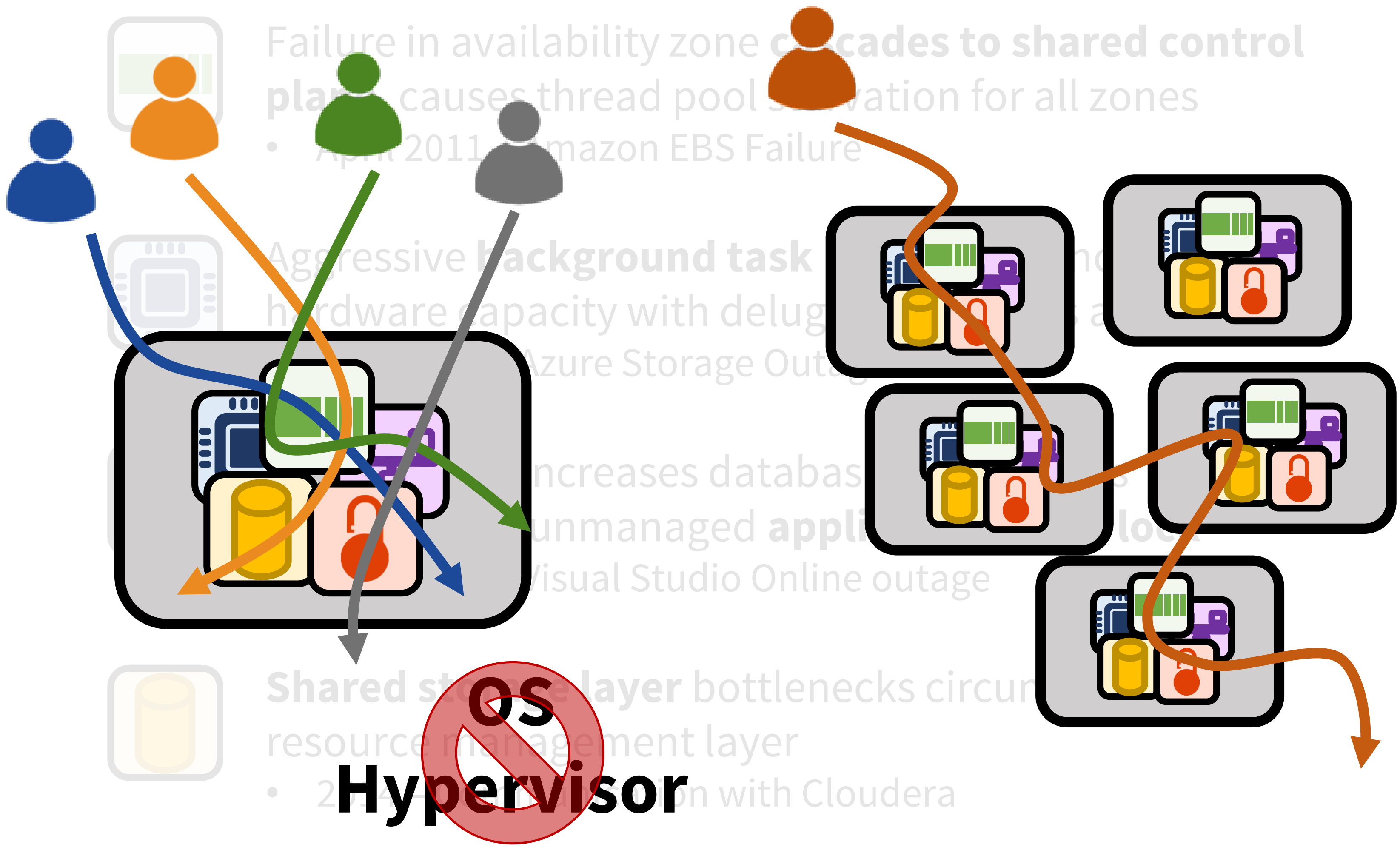
Degraded performance, Violated SLOs, system outages

Resource Management in Multi-Tenant Systems



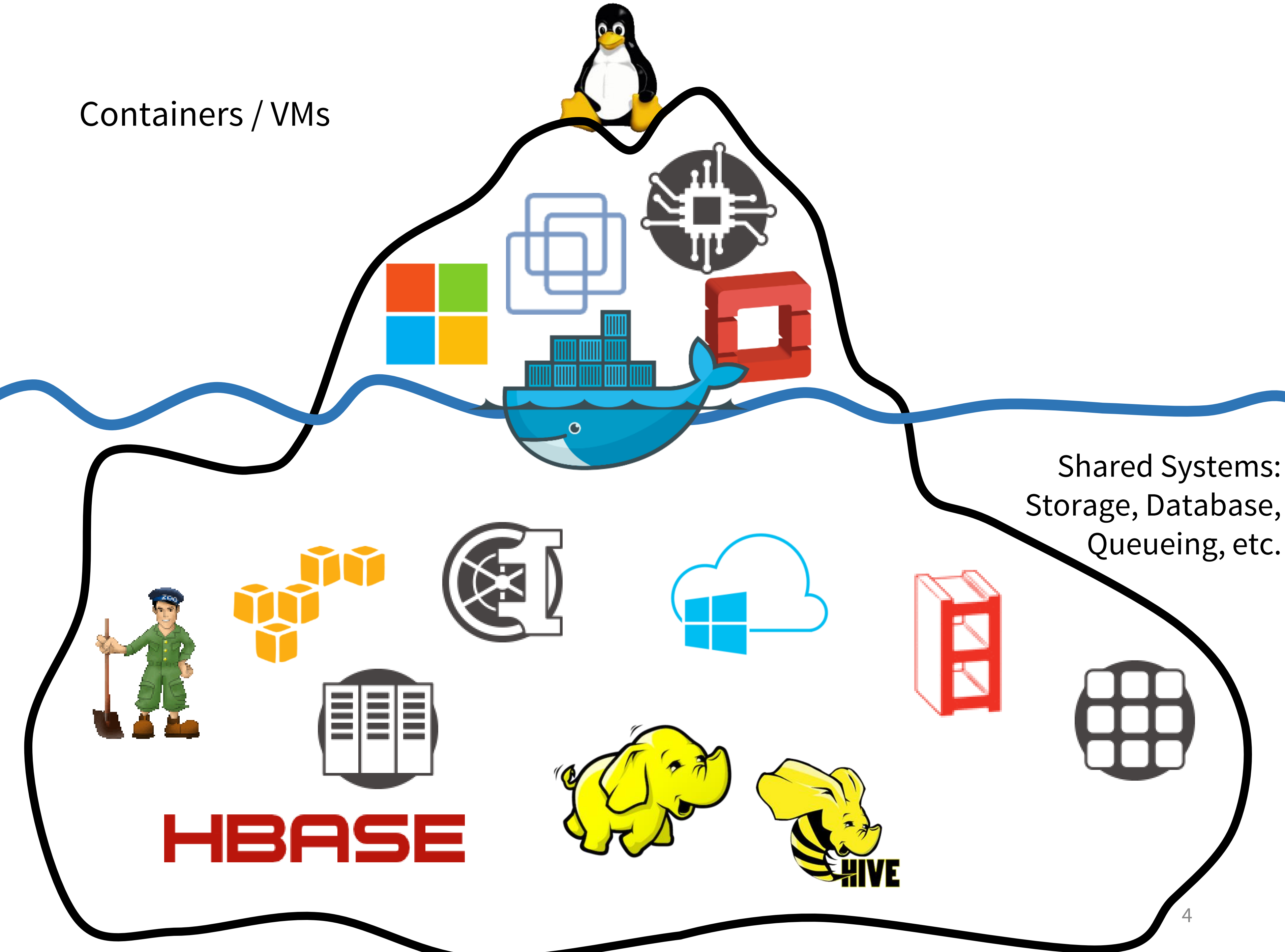
Degraded performance, Violated SLOs, system outages

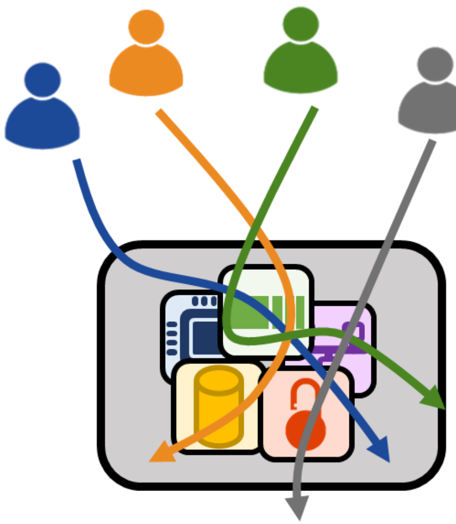
Resource Management in Multi-Tenant Systems

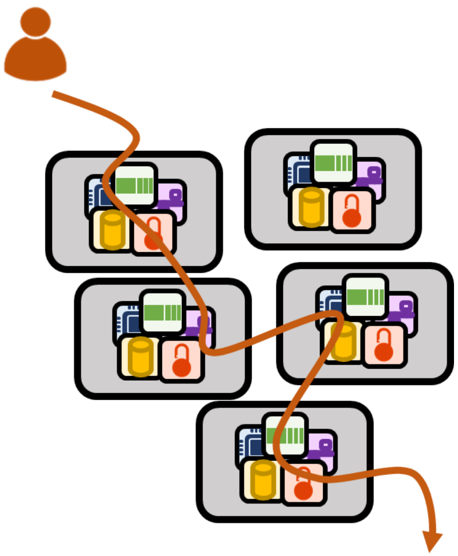


Degraded performance, Violated SLOs, system outages

Containers / VMs


Containers / VMs
Shared Systems:
Storage, Database,
Queueing, etc.
HBASE
HIVE





Retro

Monitors resource usage of each tenant in near real-time

Actively schedules tenants and activities

Monitors resource usage of each tenant in near real-time

Actively schedules tenants and activities

High-level, centralized policies:

- Encapsulates resource management logic

Monitors resource usage of each tenant in near real-time

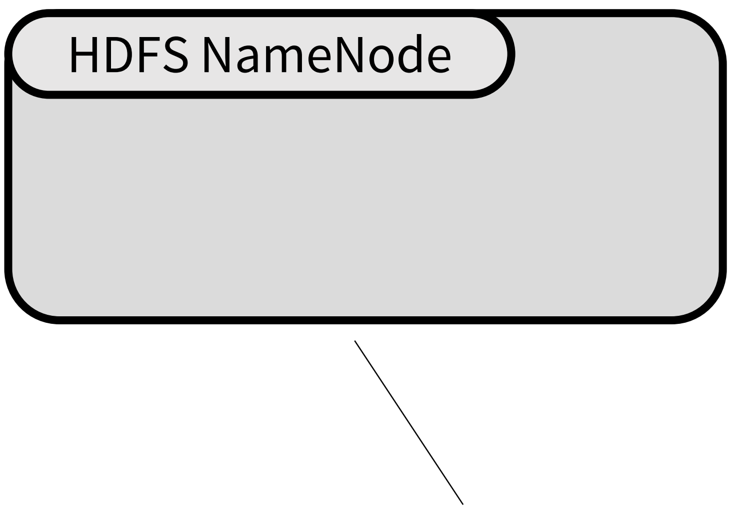
Actively schedules tenants and activities

High-level, centralized policies:

- Encapsulates resource management logic
- Abstractions – not specific to resource type, system
- Achieve different goals: guarantee average latencies, fair share a resource, etc.

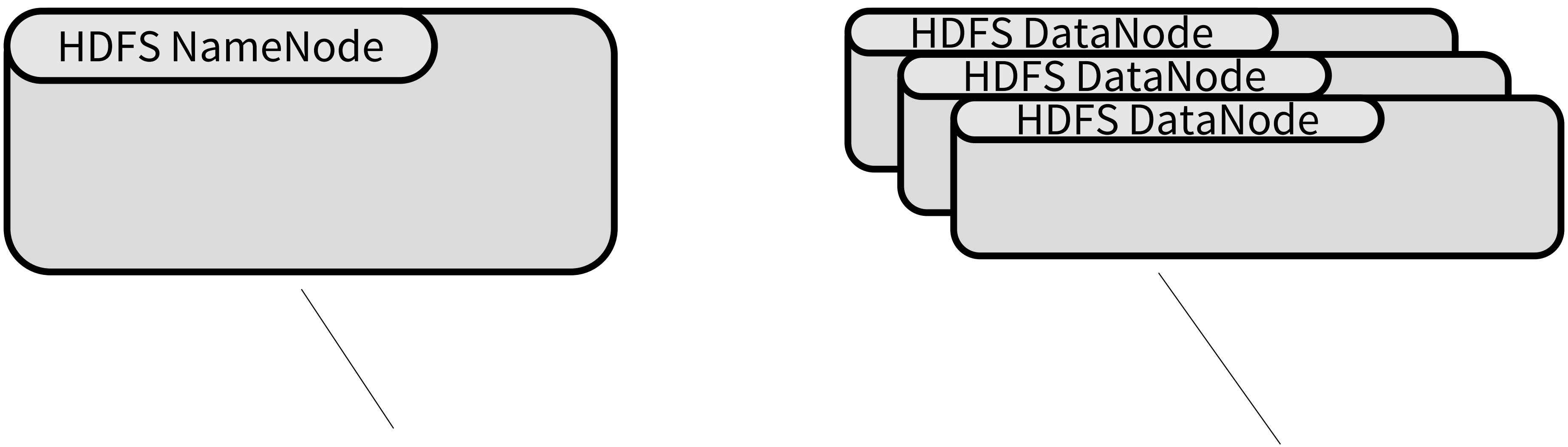
Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS)



Filesystem metadata

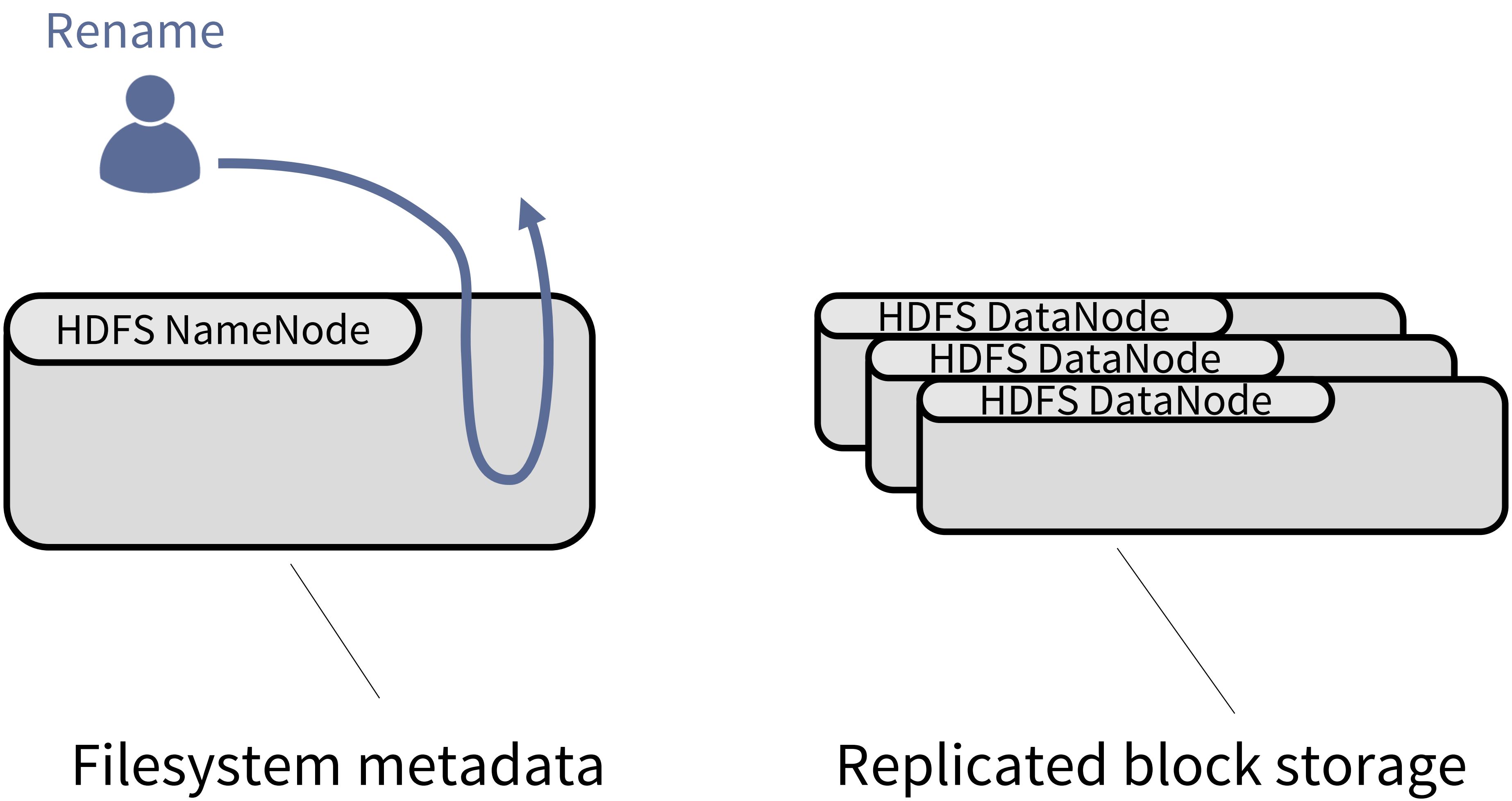
Hadoop Distributed File System (HDFS)



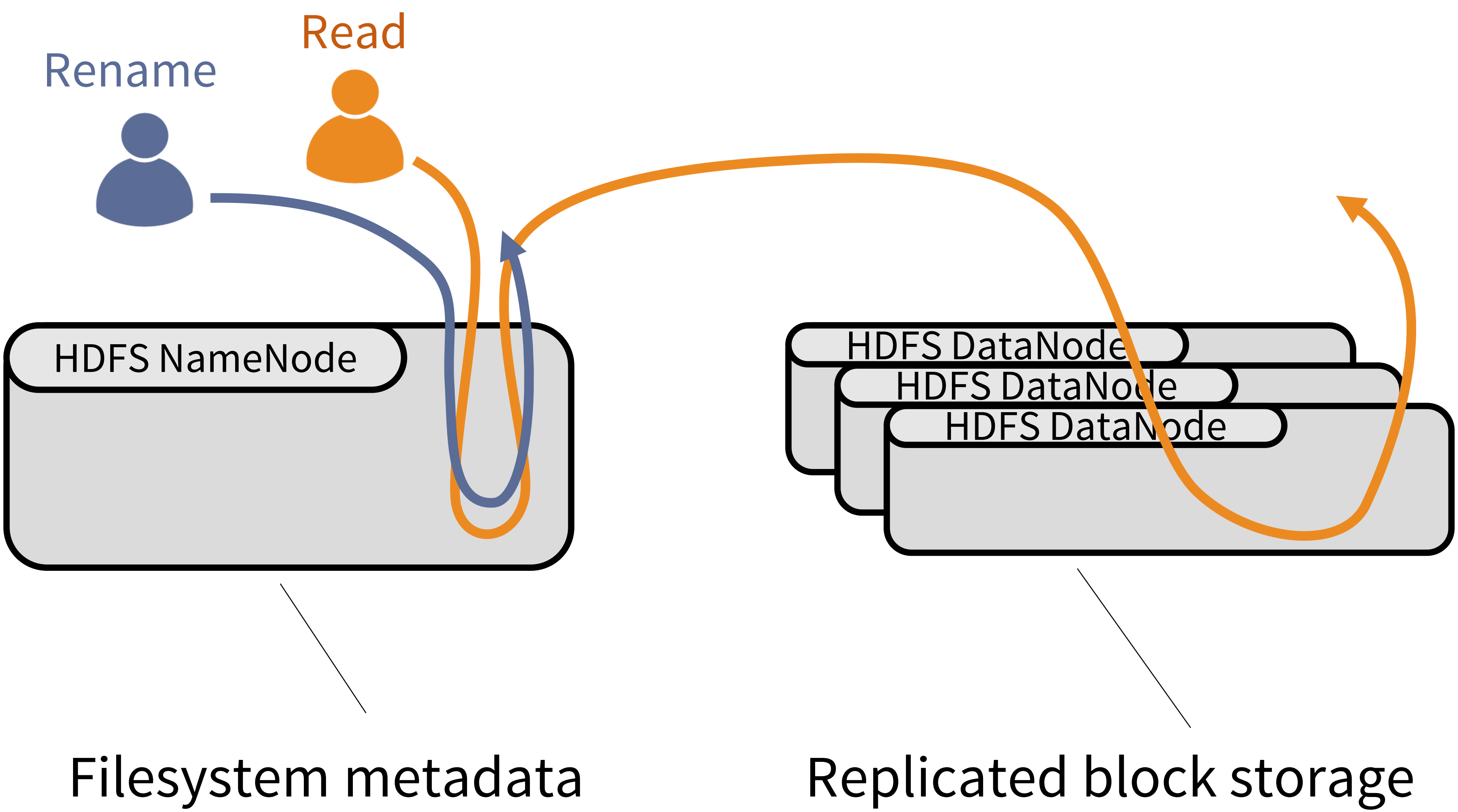
Filesystem metadata

Replicated block storage

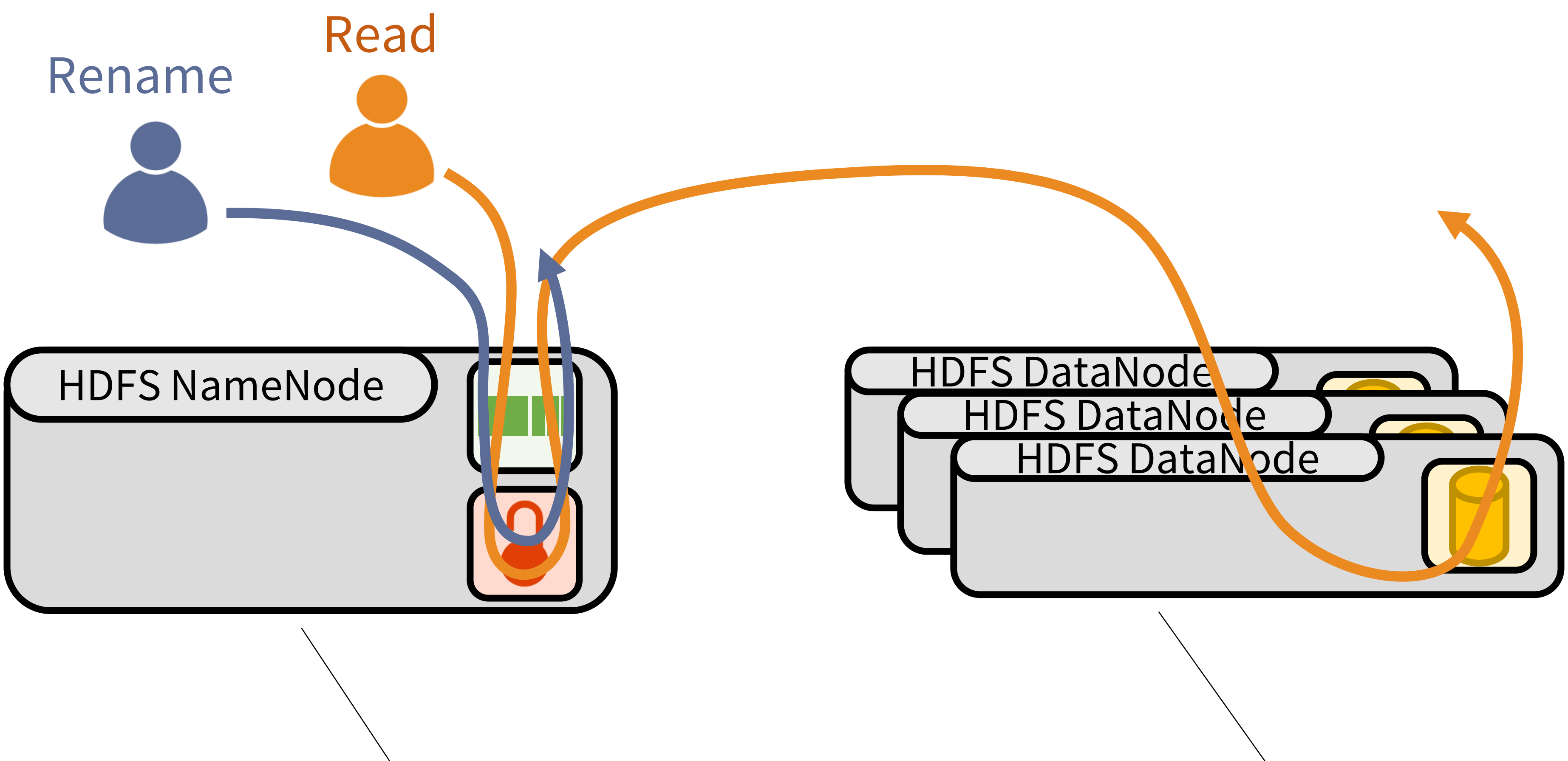
Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)

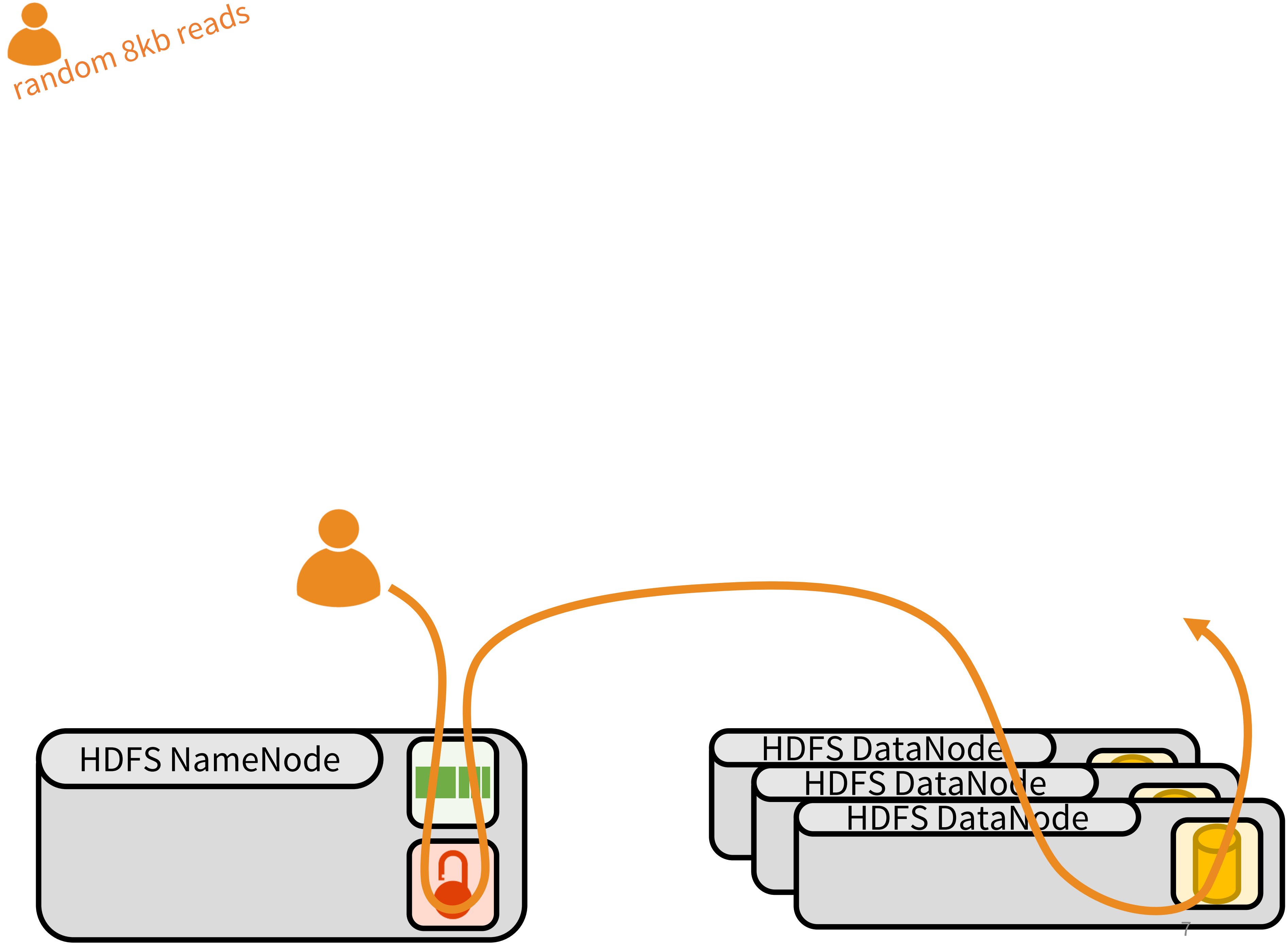


Hadoop Distributed File System (HDFS)

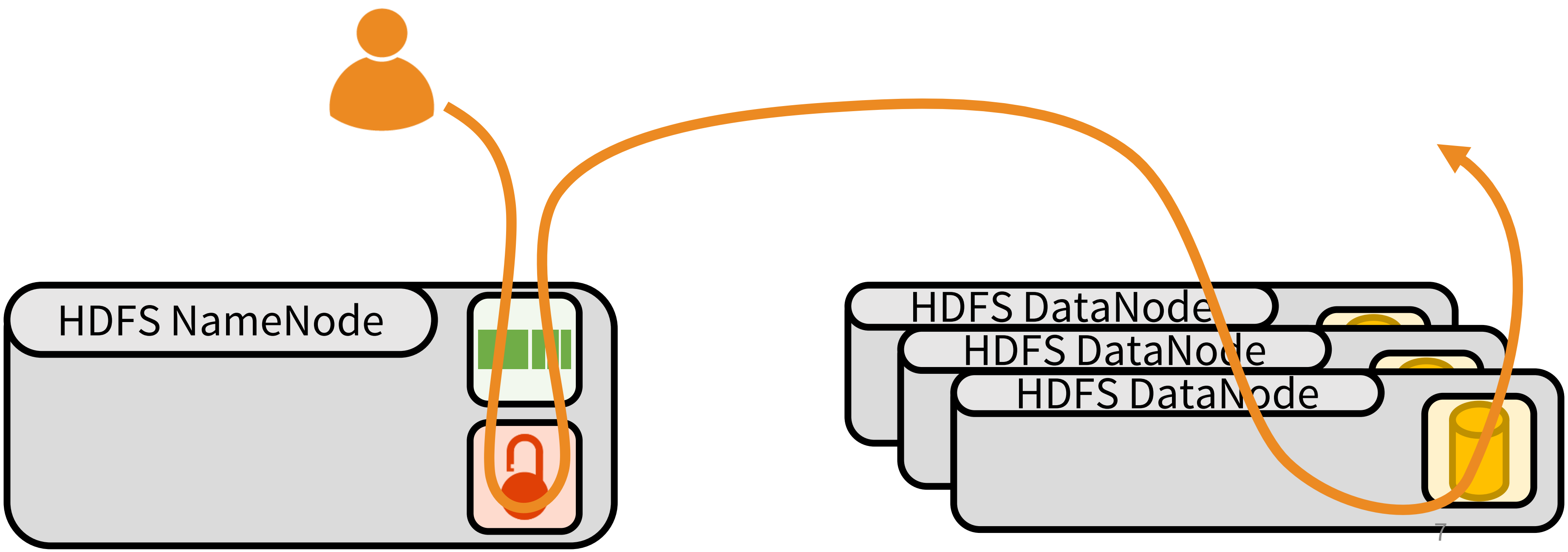


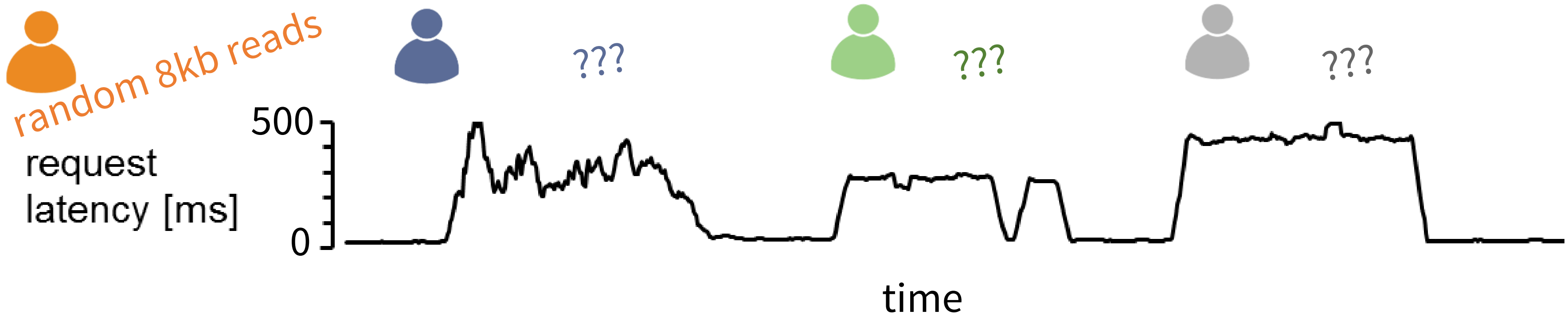
Filesystem metadata

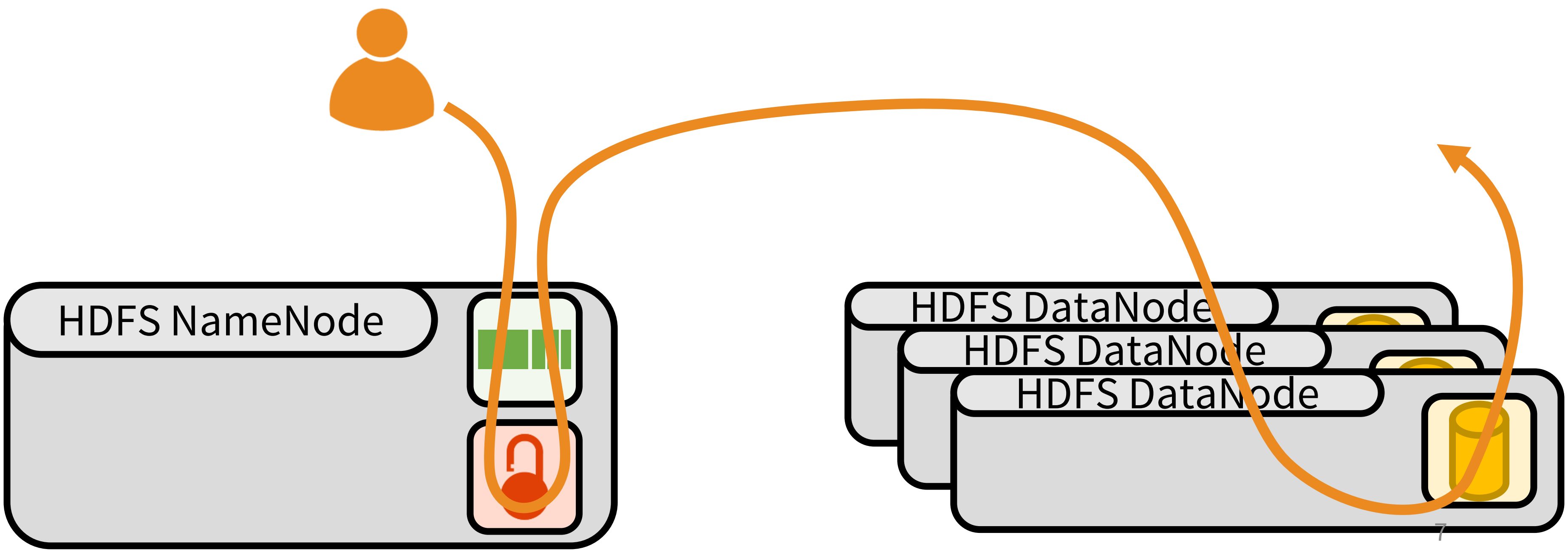
Replicated block storage

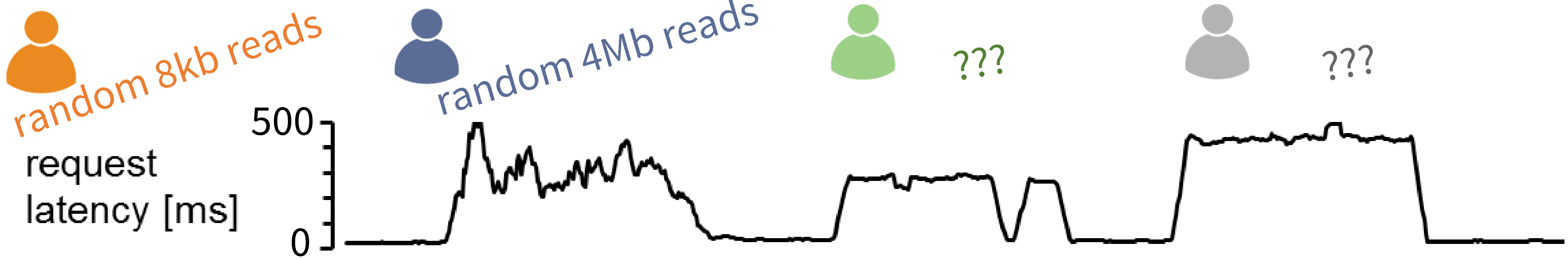

random 8kb reads
HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode


random 8kb reads
request latency [ms]
500
0
time

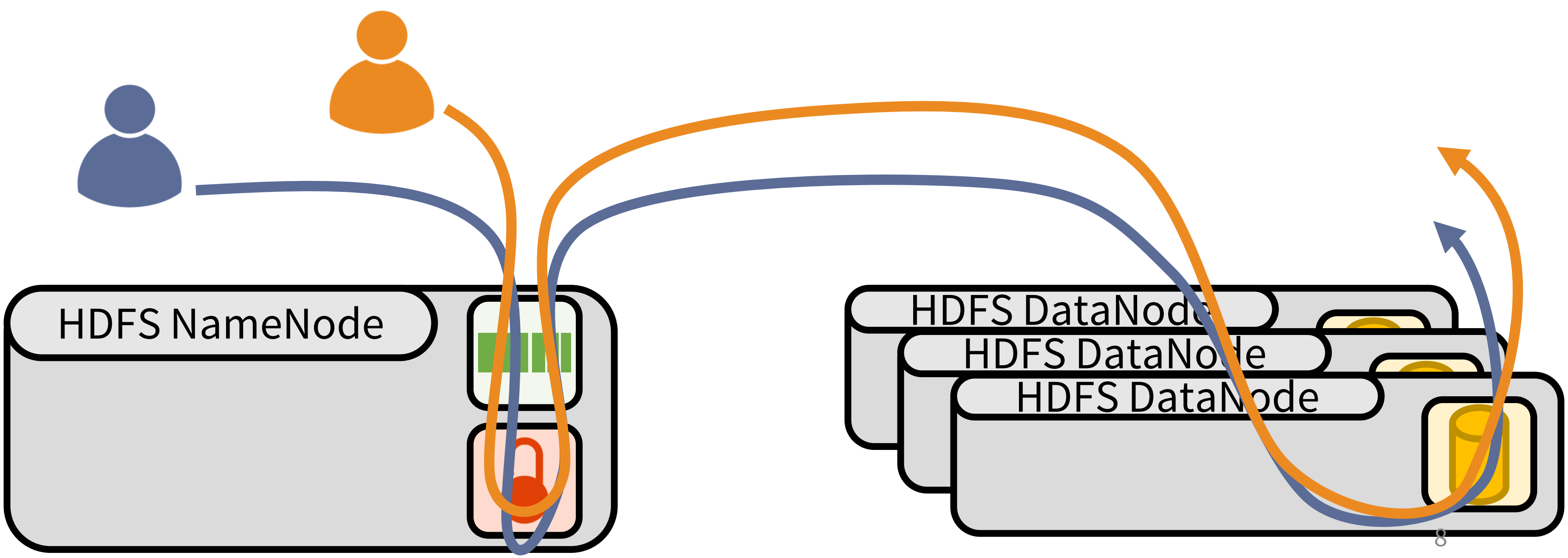

HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode


random 8kb reads
???
???
???
500
request
latency [ms]
0
time


HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

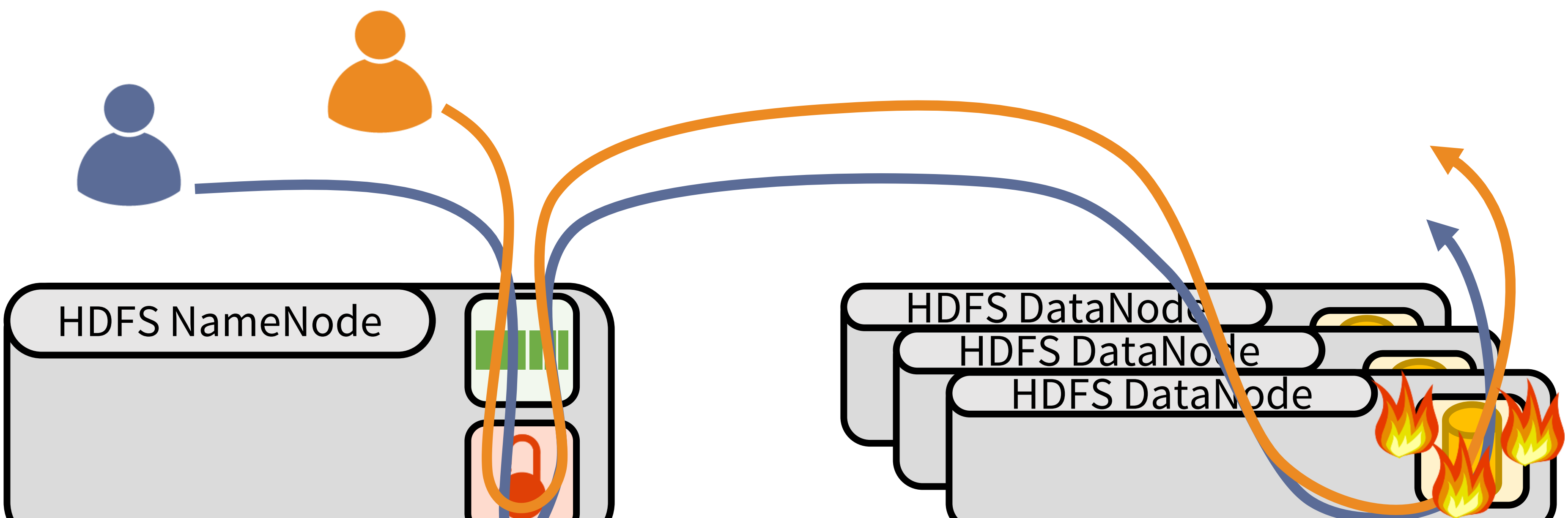


random 8kb reads
random 4Mb reads
???
???
500
request latency [ms]
0

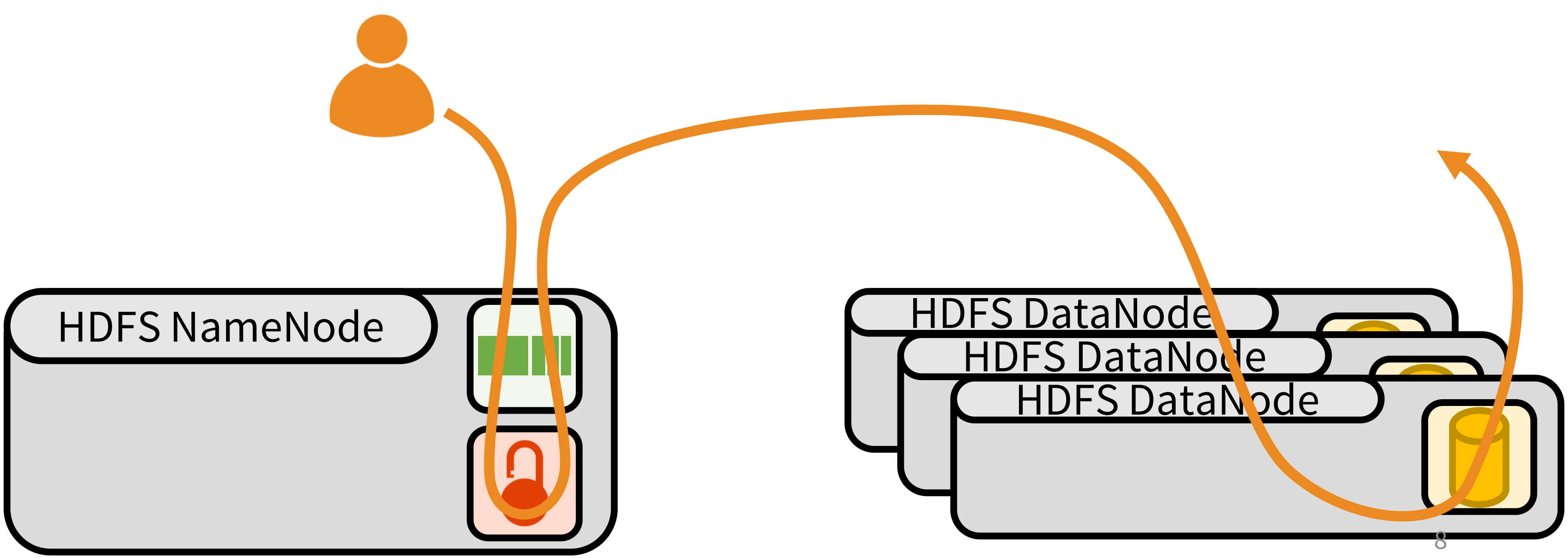


HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

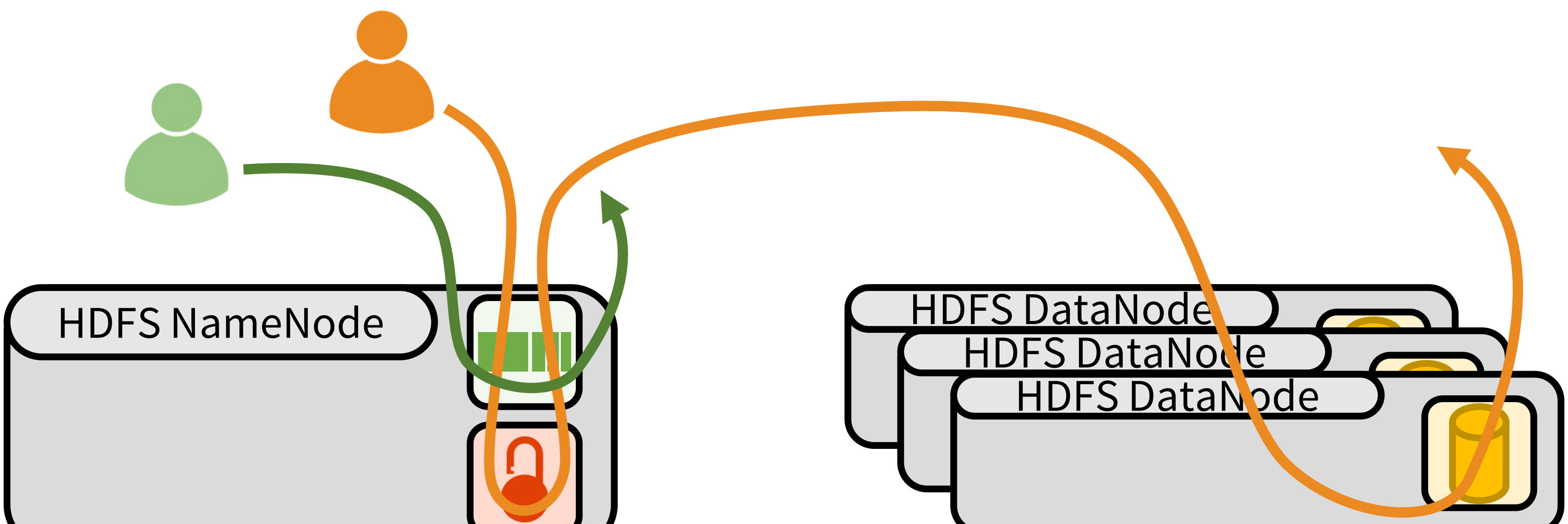

random 8kb reads
random 4Mb reads
???
???
500
request latency [ms]
0
12
disk op latency [ms]
0

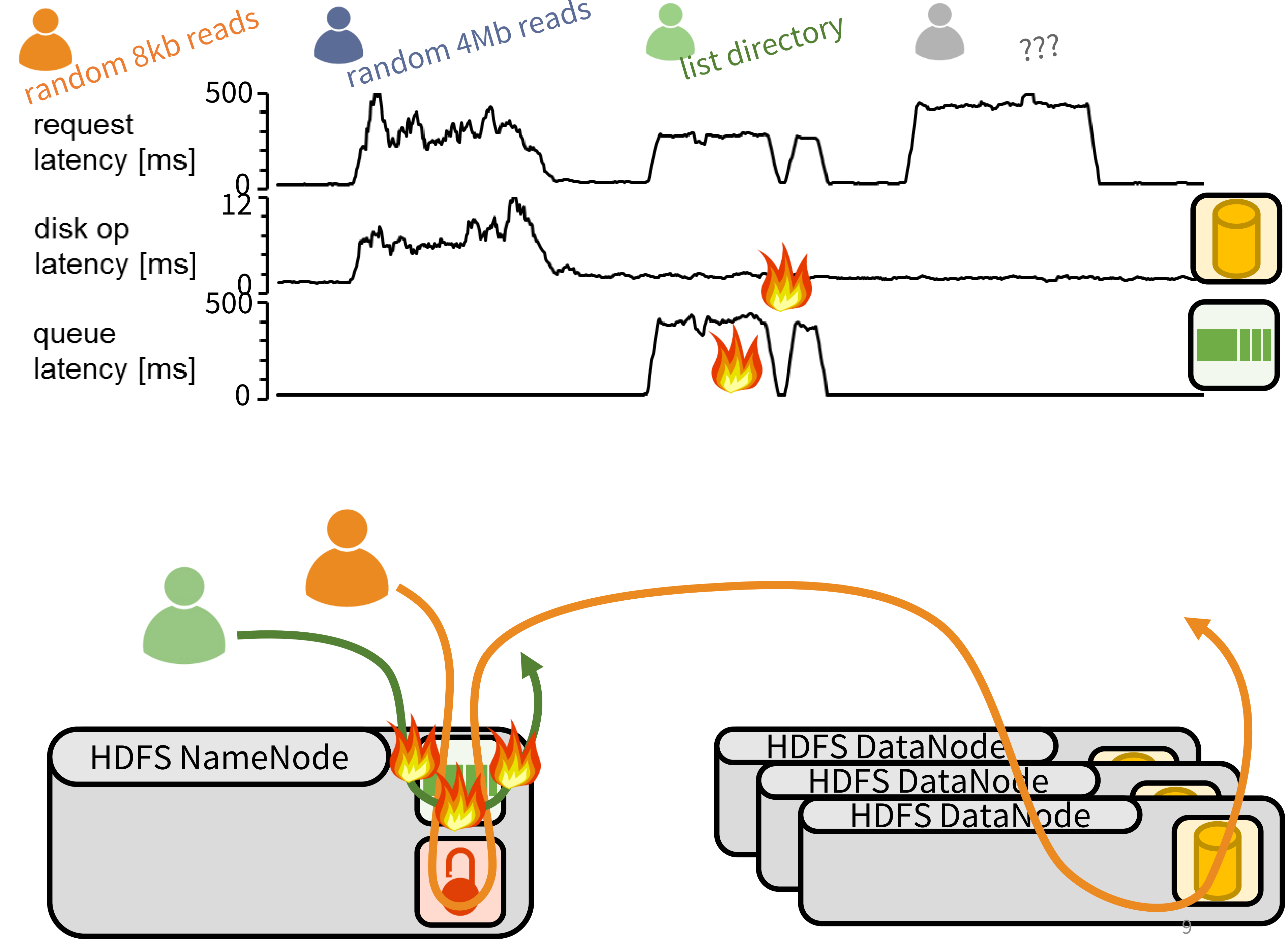

HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

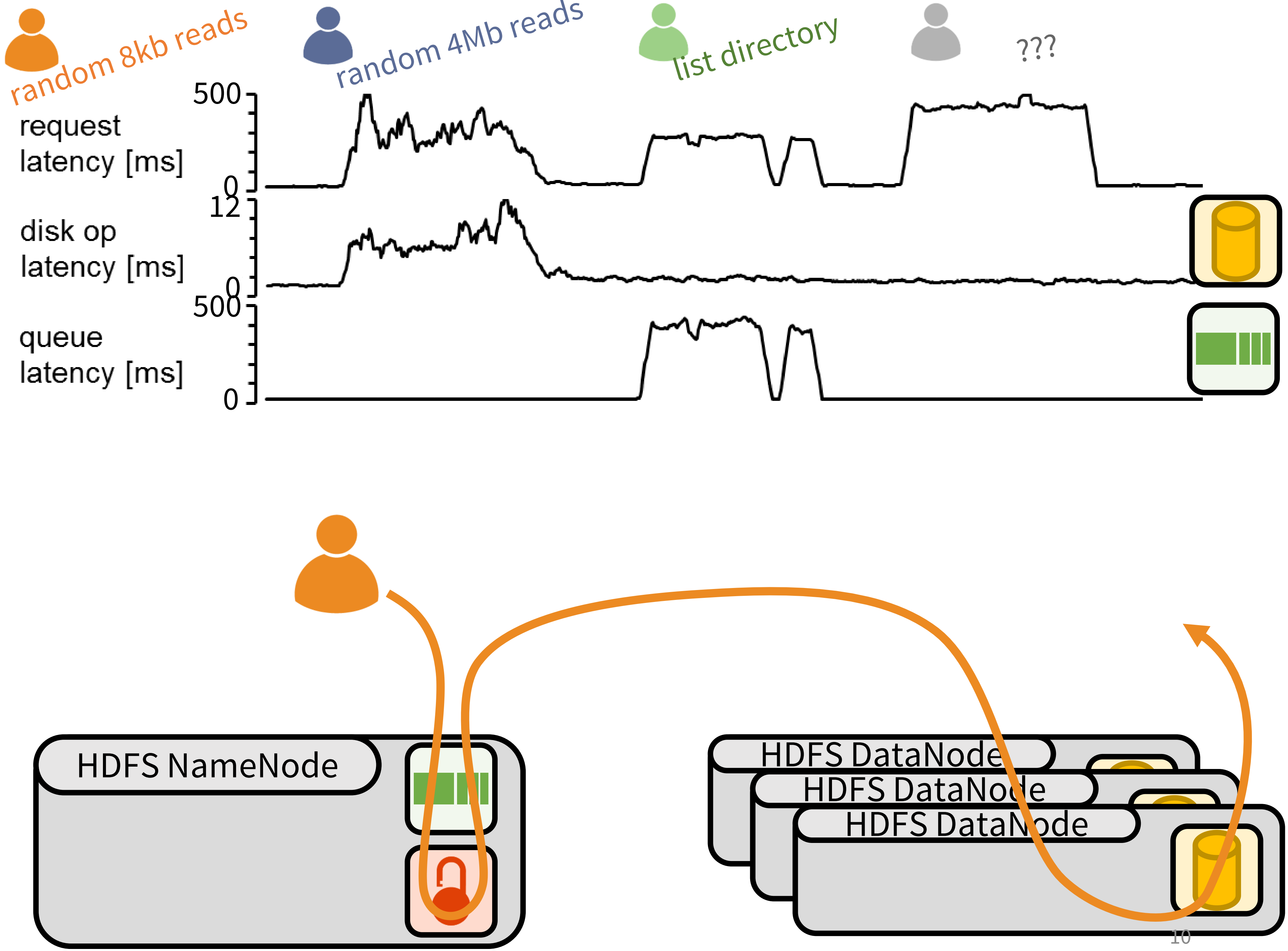

random 8kb reads
random 4Mb reads
???
???
500
request latency [ms]
0
12
disk op latency [ms]
0

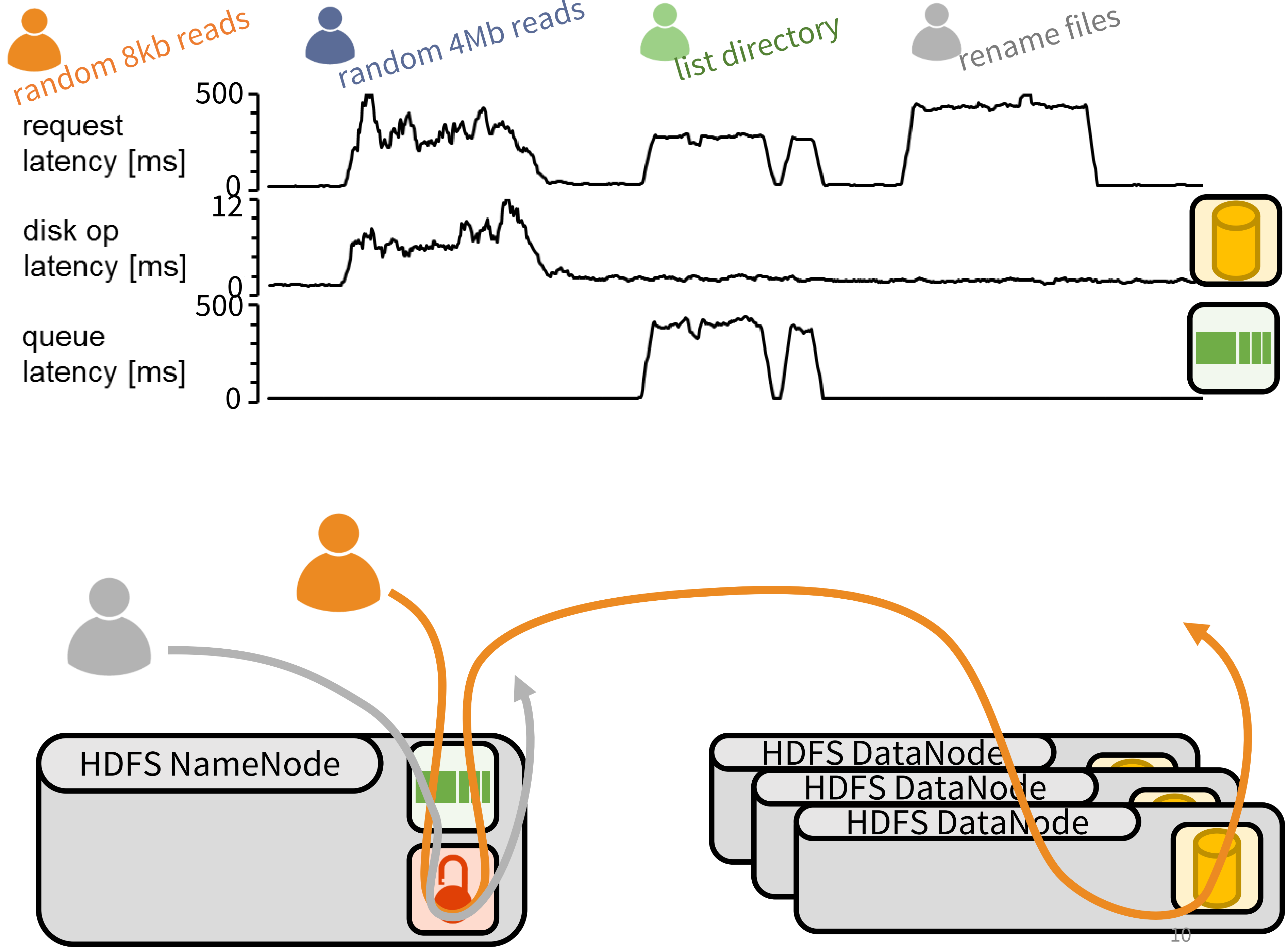

HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

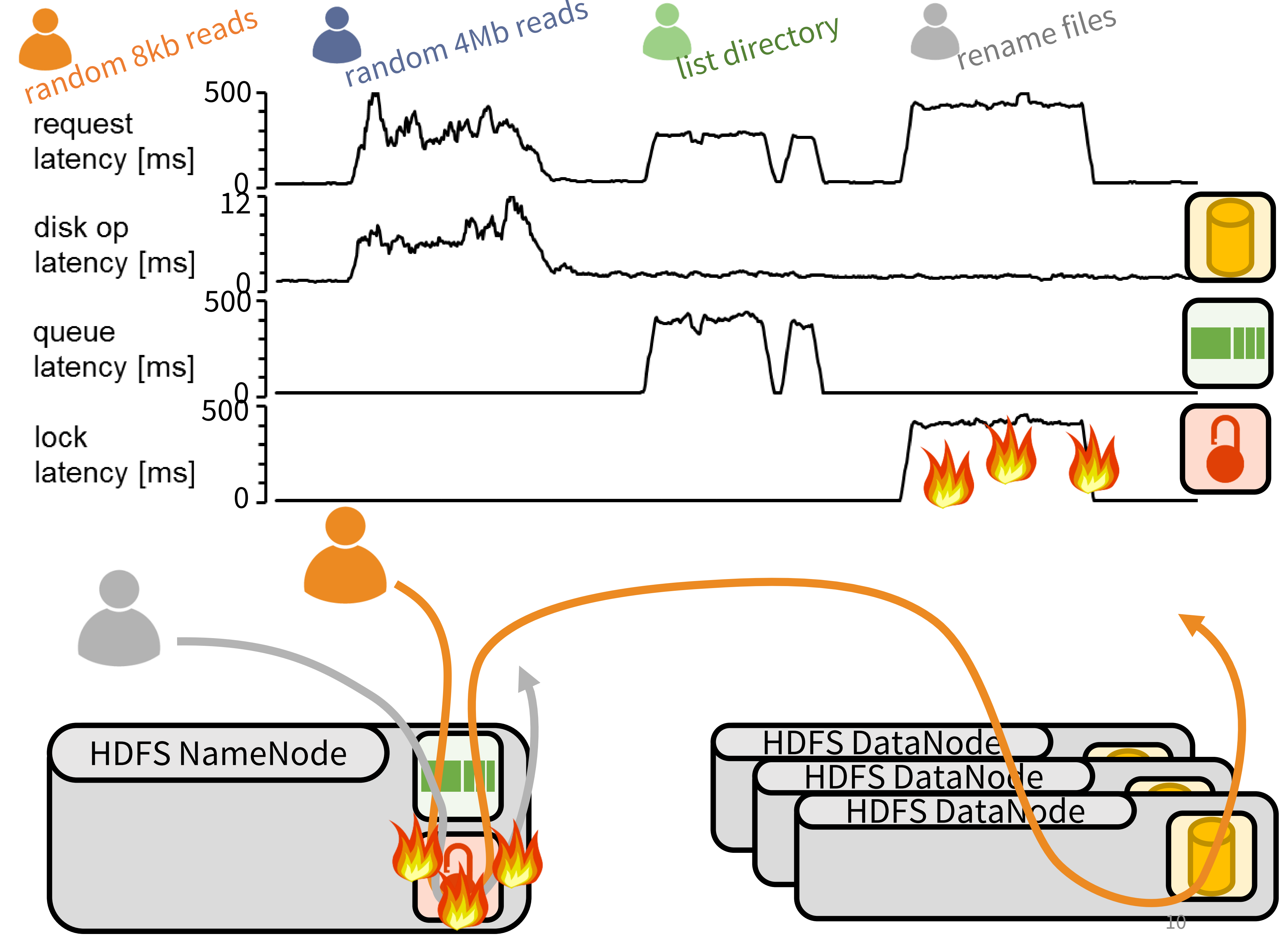

random 8kb reads
random 4Mb reads
list directory
???
request
latency [ms]
500
0
disk op
latency [ms]
12
0


HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

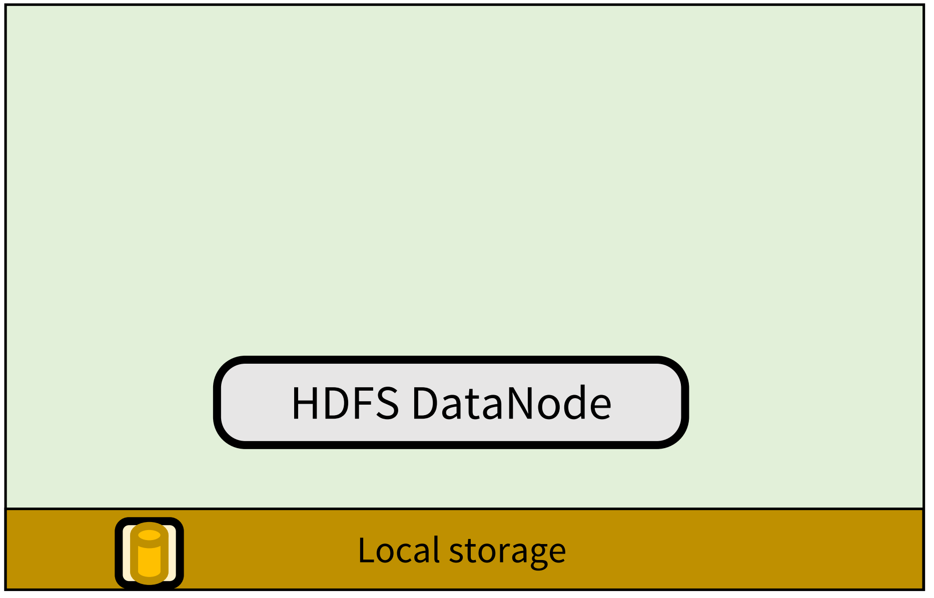

random 8kb reads
random 4Mb reads
list directory
???
request latency [ms]
500
0
disk op latency [ms]
12
0
queue latency [ms]
500
0
HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

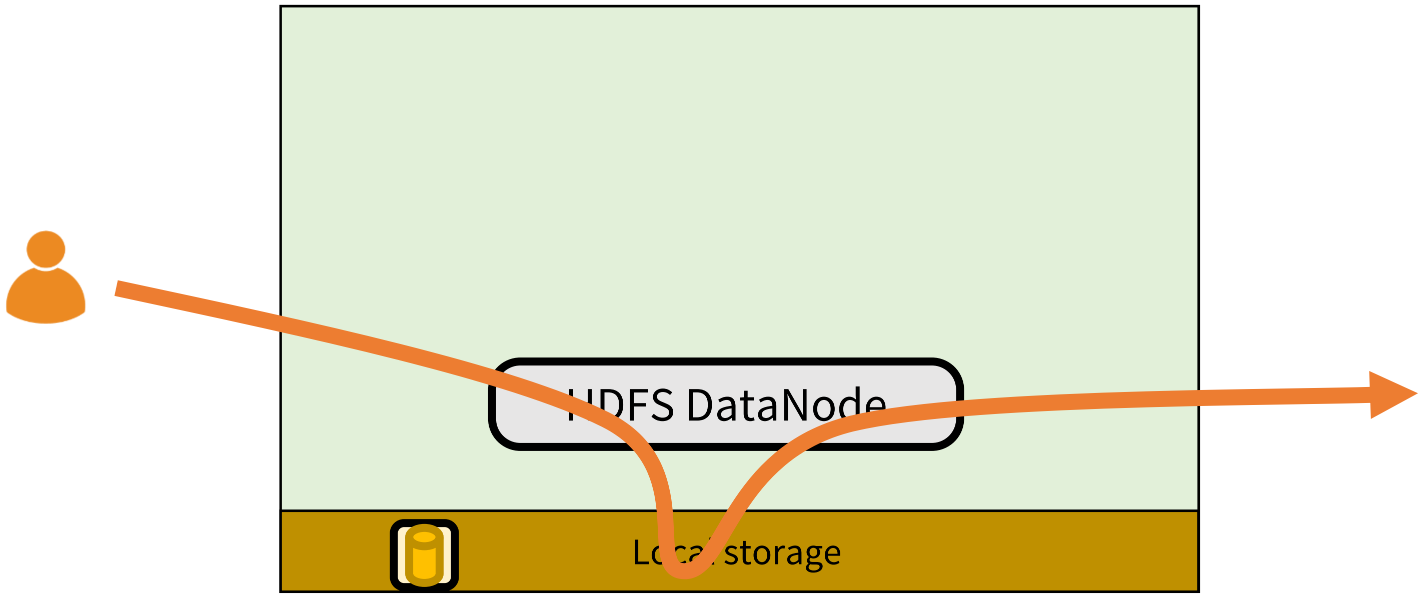

random 8kb reads
random 4Mb reads
list directory
???
request latency [ms]
500
0
disk op latency [ms]
12
0
queue latency [ms]
500
0
HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

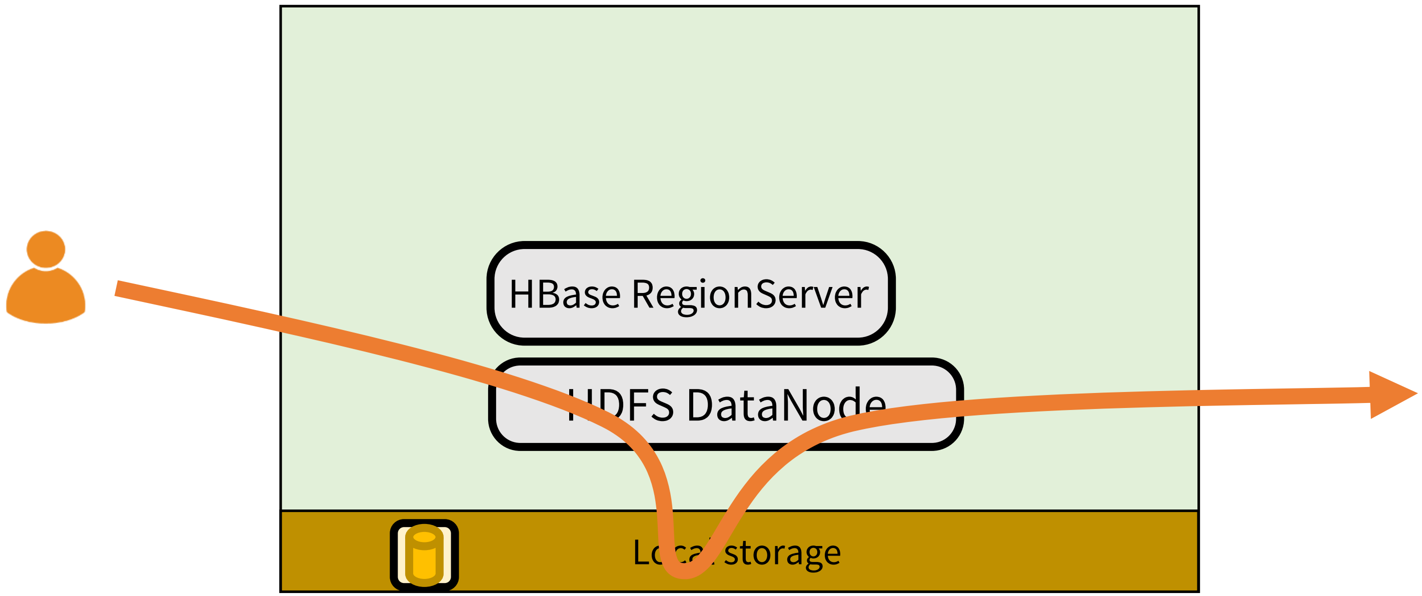

random 8kb reads
random 4Mb reads
list directory
rename files
request latency [ms]
500
0
disk op latency [ms]
12
0
queue latency [ms]
500
0
HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

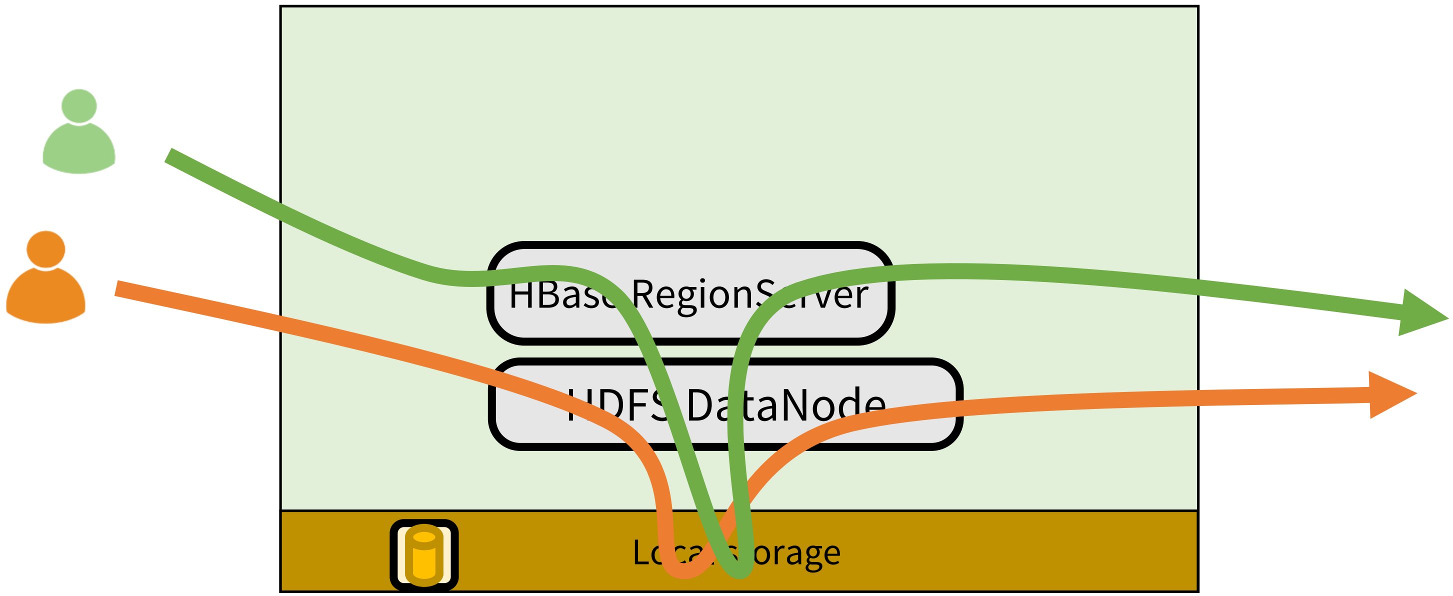


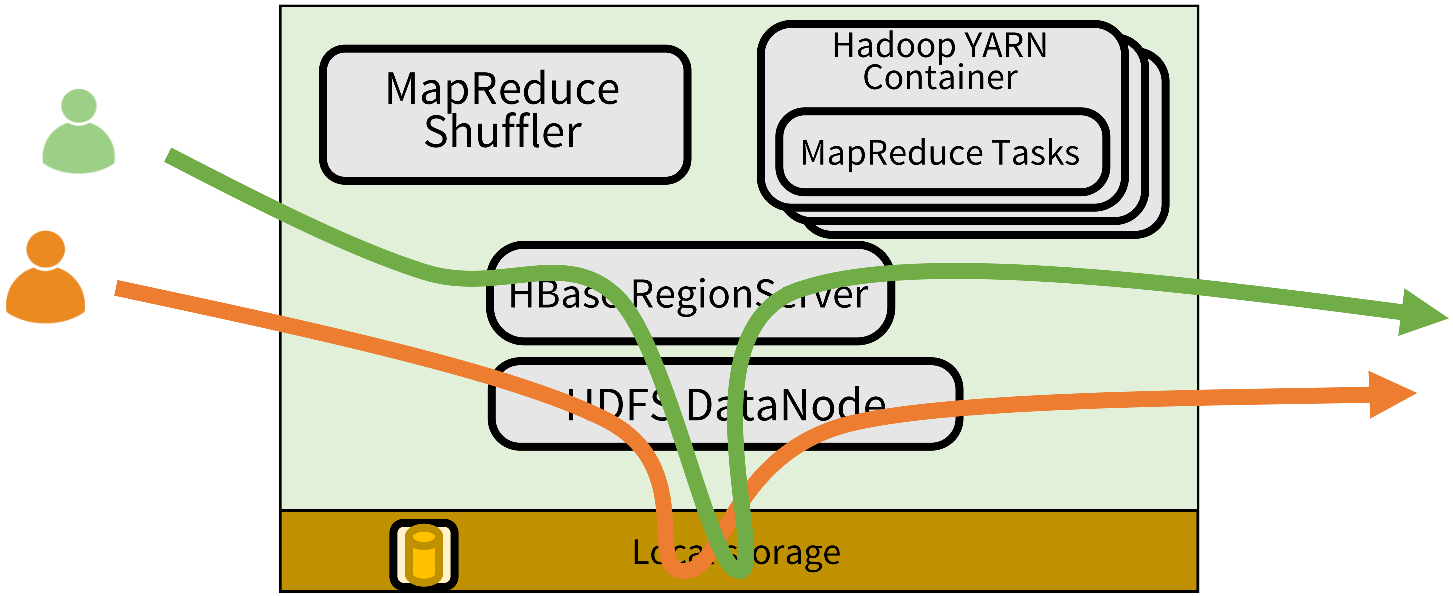
random 8kb reads
random 4Mb reads
list directory
rename files
request latency [ms]
500
0
disk op latency [ms]
12
0
queue latency [ms]
500
0
lock latency [ms]
500
0
HDFS NameNode
HDFS DataNode
HDFS DataNode
HDFS DataNode

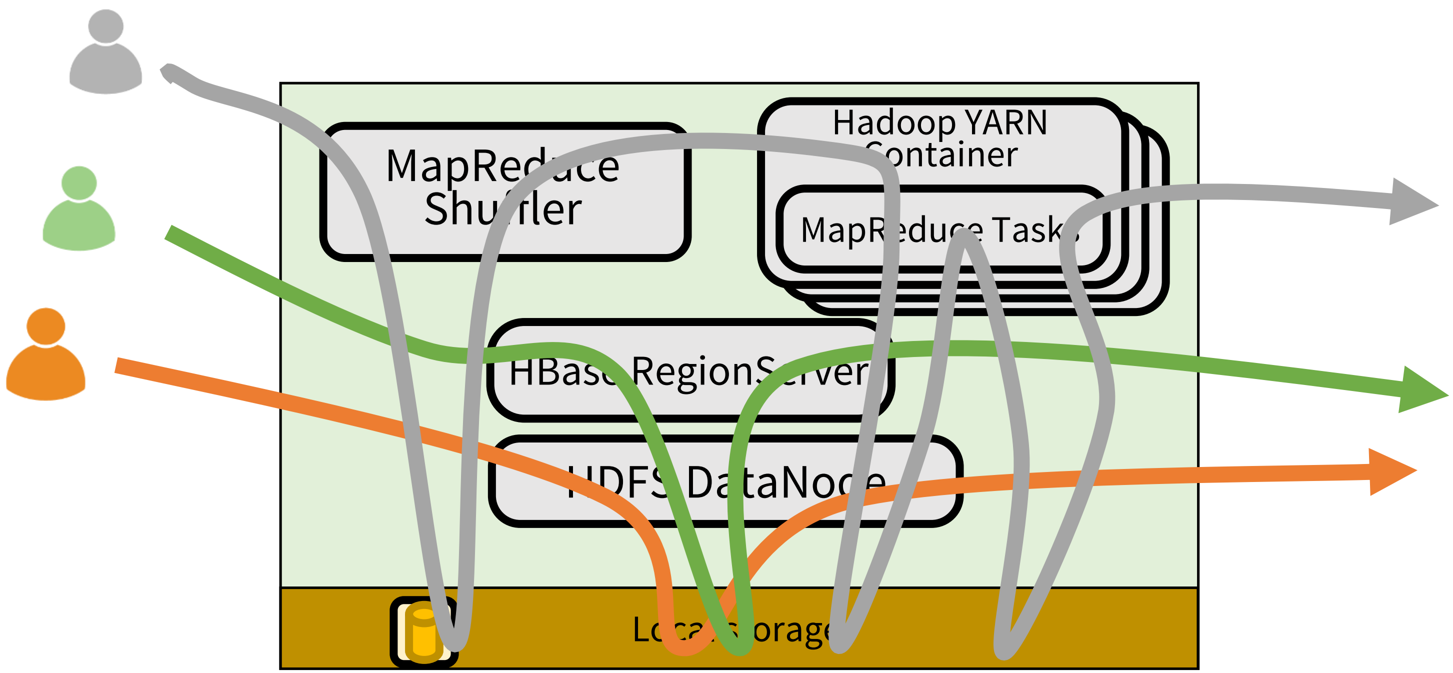

HDFS DataNode
Local storage


HDFS DataNode
Local storage

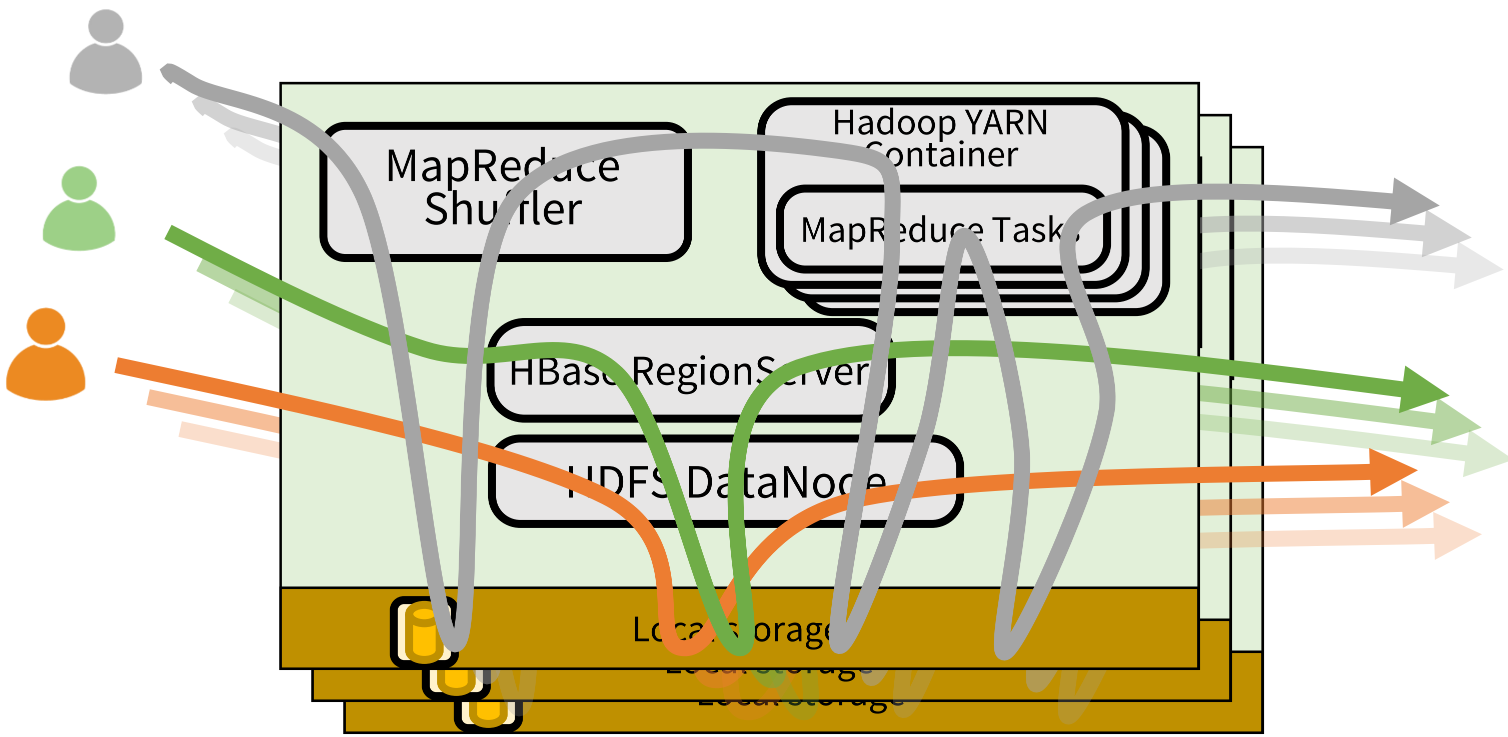

HBase RegionServer
HDFS DataNode
Local storage


HBase RegionServer
HDFS DataNode
Local Storage


MapReduce Shuffler
Hadoop YARN Container
MapReduce Tasks
HBase RegionServer
HDFS DataNode
Local Storage



MapReduce Shuffler
Hadoop YARN Container
MapReduce Tasks
HBase RegionServer
HDFS DataNode
Local storage


MapReduce Shuffler
Hadoop YARN Container
MapReduce Tasks
HBase RegionServer
HDFS DataNode
Local Storage

Retro Goals

Retro Goals

Co-ordinated control across processes, machines, and services

Retro Goals

Co-ordinated control across processes, machines, and services

Handle system and application level resources

Retro Goals

Co-ordinated control across processes, machines, and services

Handle system and application level resources

Principals: tenants, background tasks

Retro Goals

Co-ordinated control across processes, machines, and services

Handle system and application level resources

Principals: tenants, background tasks

Real-time and reactive

Retro Goals

Co-ordinated control across processes, machines, and services

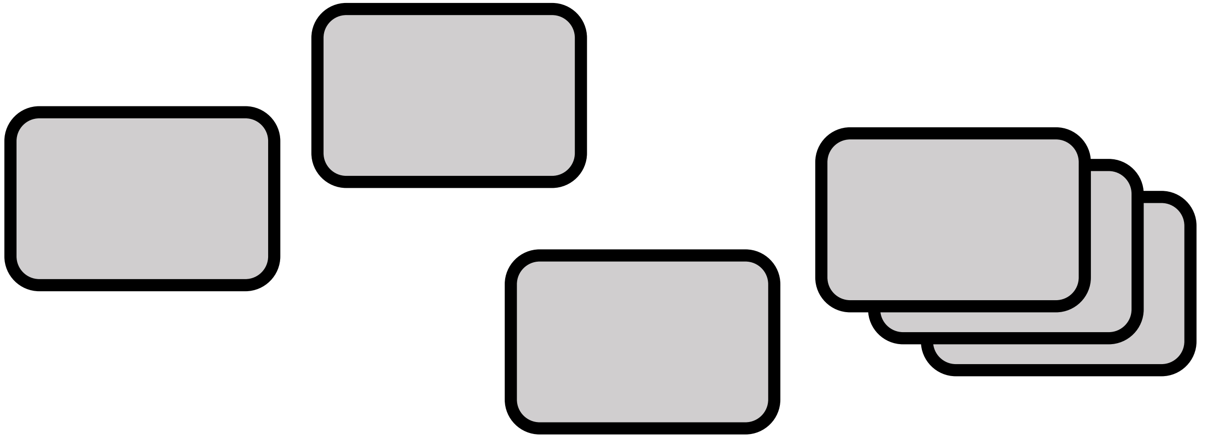
Handle system and application level resources

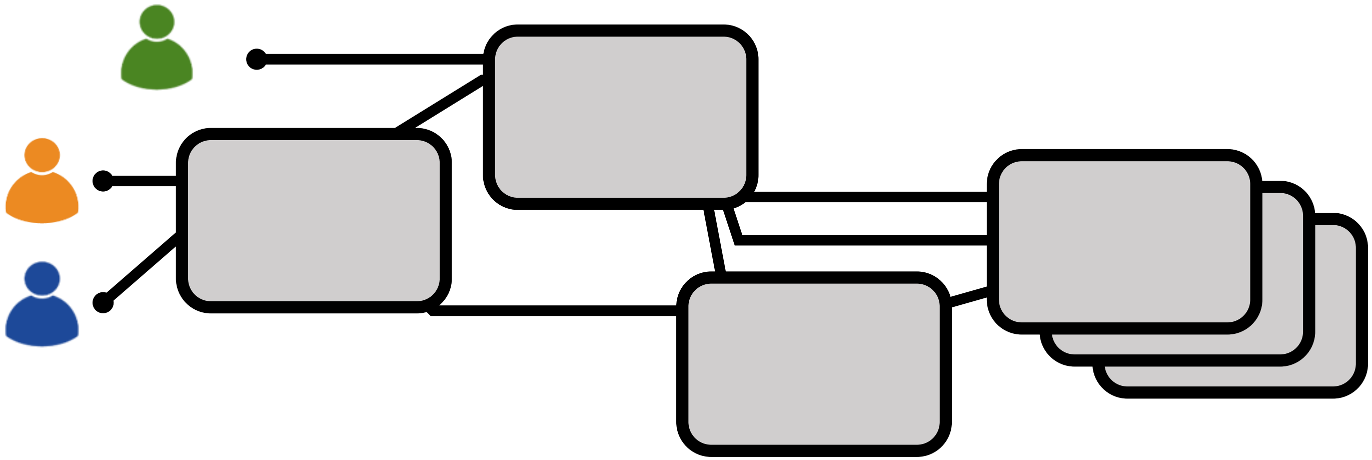
Principals: tenants, background tasks

Real-time and reactive

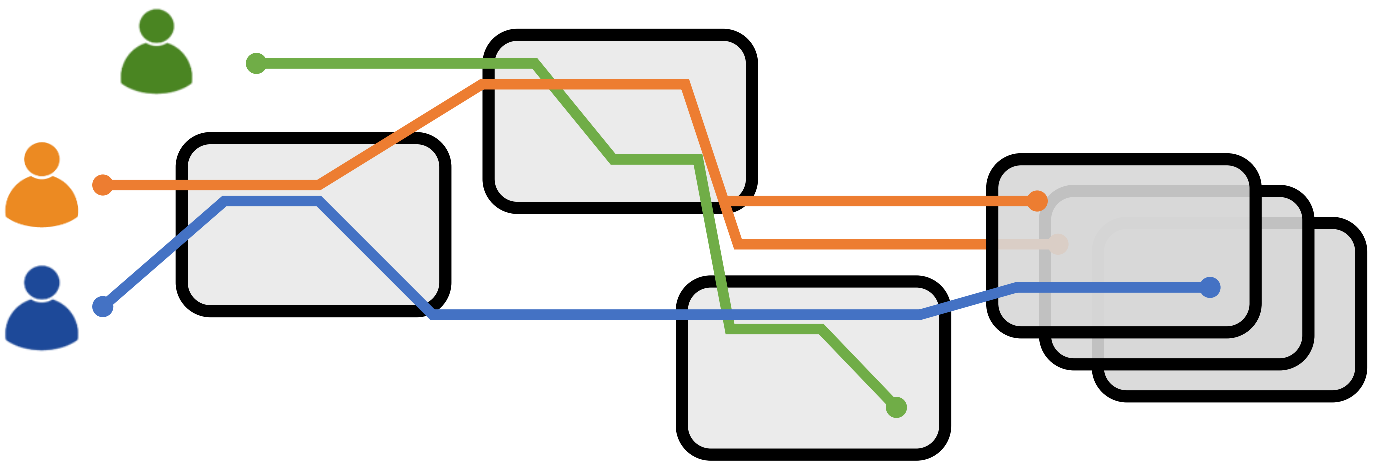
Efficient: Only control what is needed

Retro Architecture





Tenant Requests

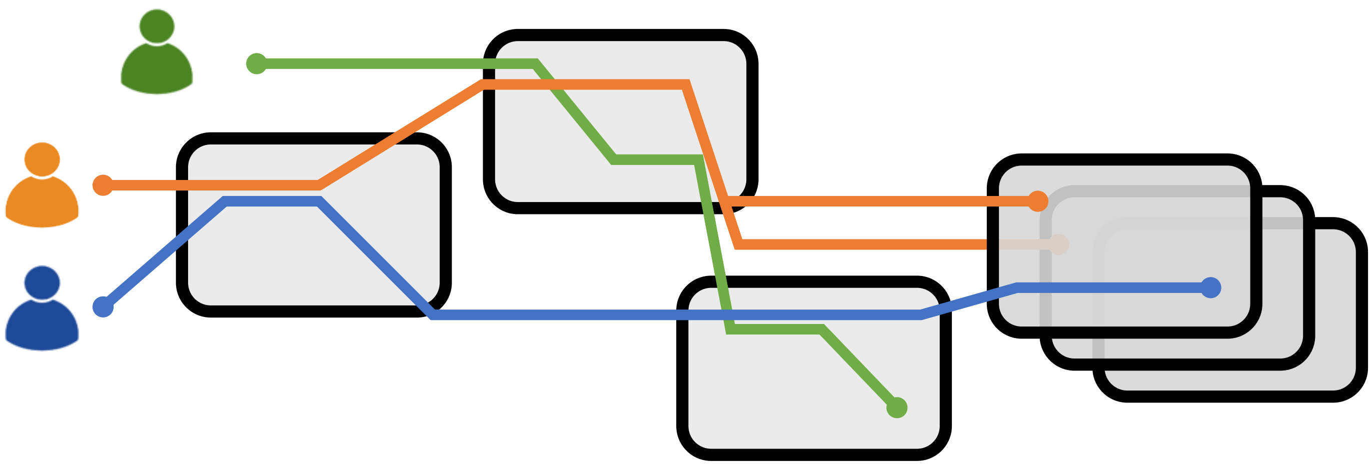


Workflows

Purpose: identify requests from different users, background activities

eg, all requests from a tenant over time

eg, data balancing in HDFS



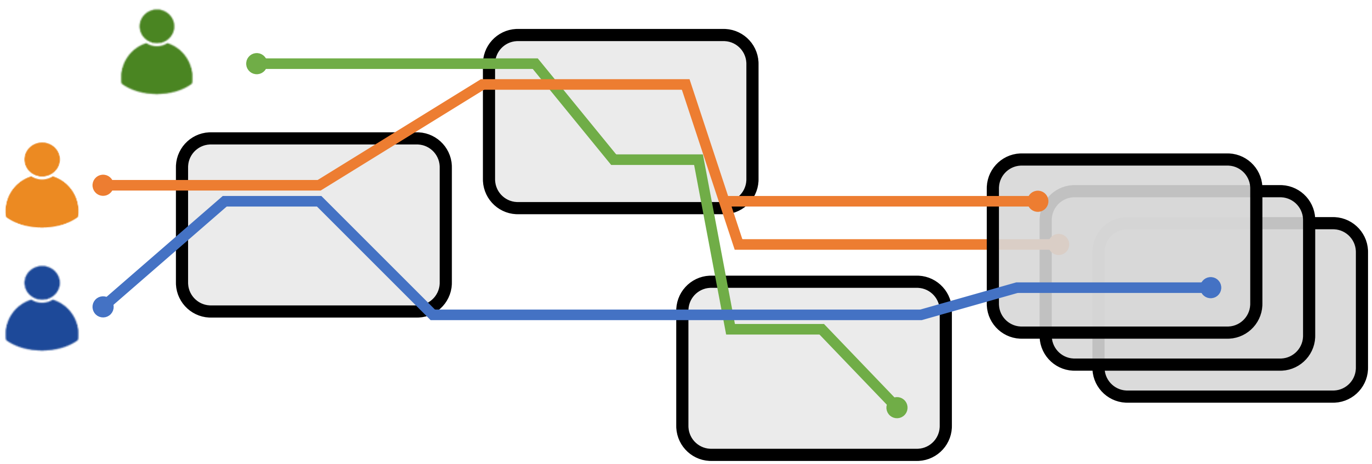
Workflows

Purpose: identify requests from different users, background activities

eg, all requests from a tenant over time

eg, data balancing in HDFS

Unit of resource measurement, attribution, and enforcement



 Workflows

Purpose: identify requests from different users, background activities

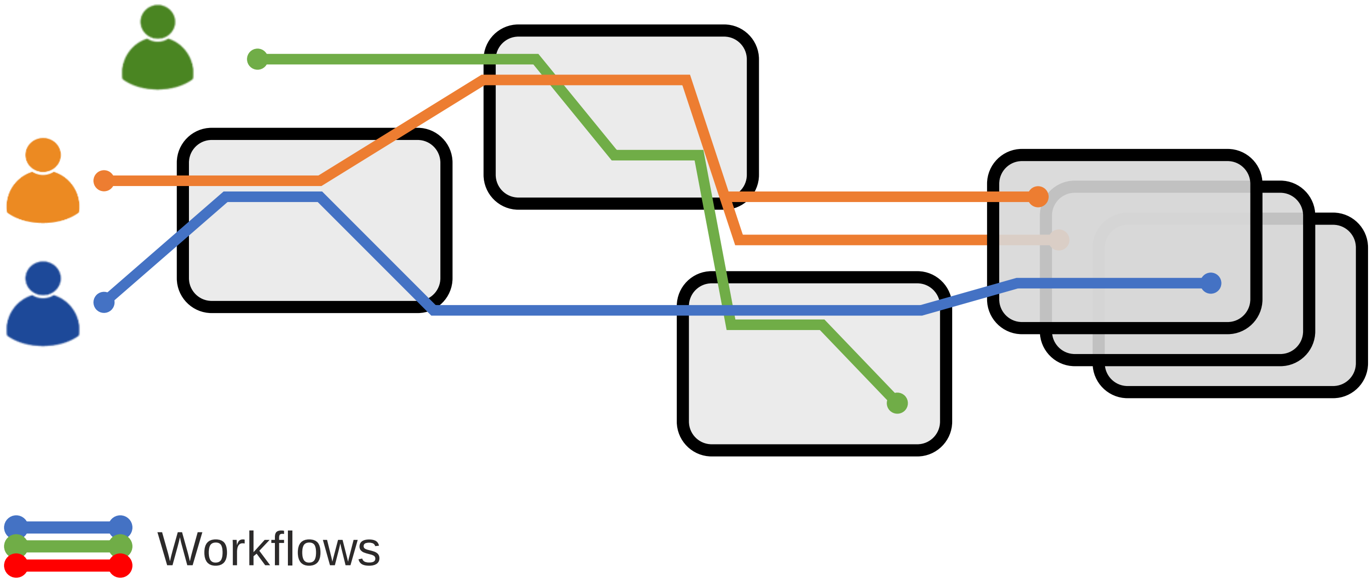
eg, all requests from a tenant over time

eg, data balancing in HDFS

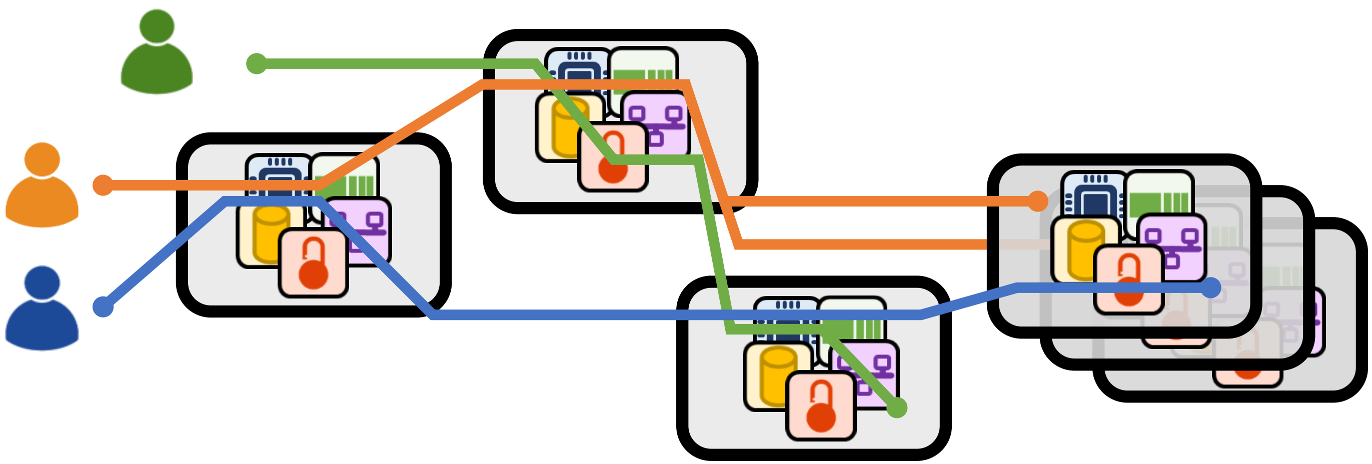
Unit of resource measurement, attribution, and enforcement

Tracks a request across varying levels of granularity

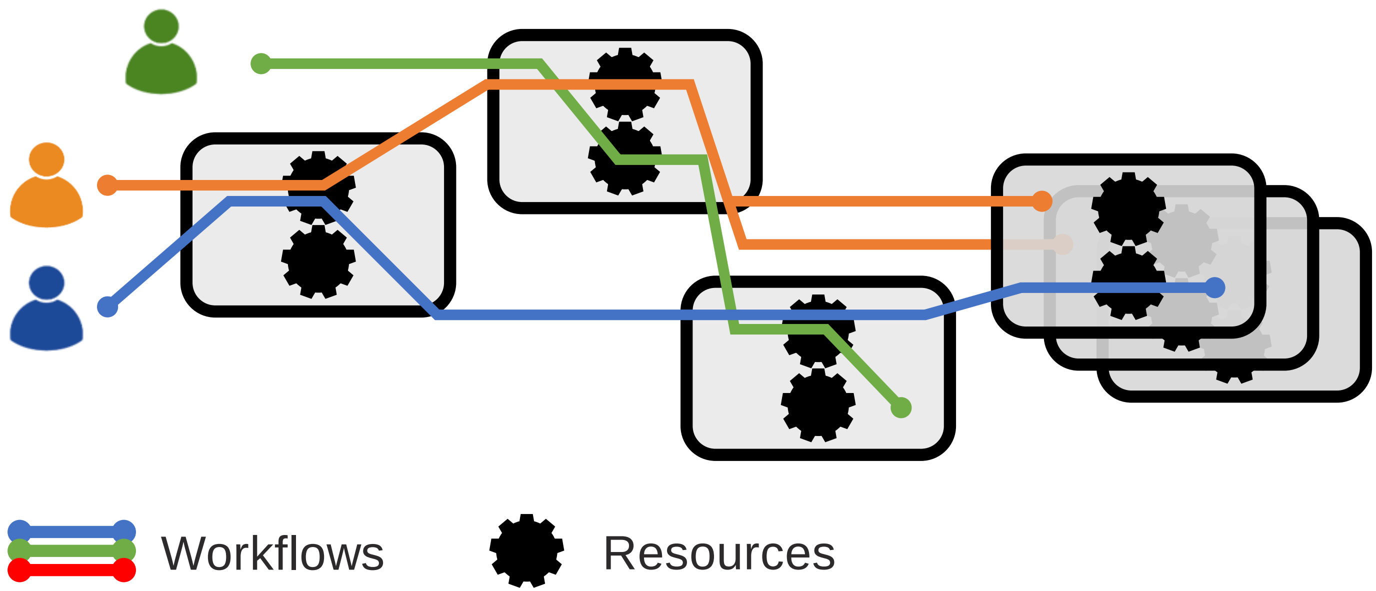
Orthogonal to threads, processes, network flows, etc.



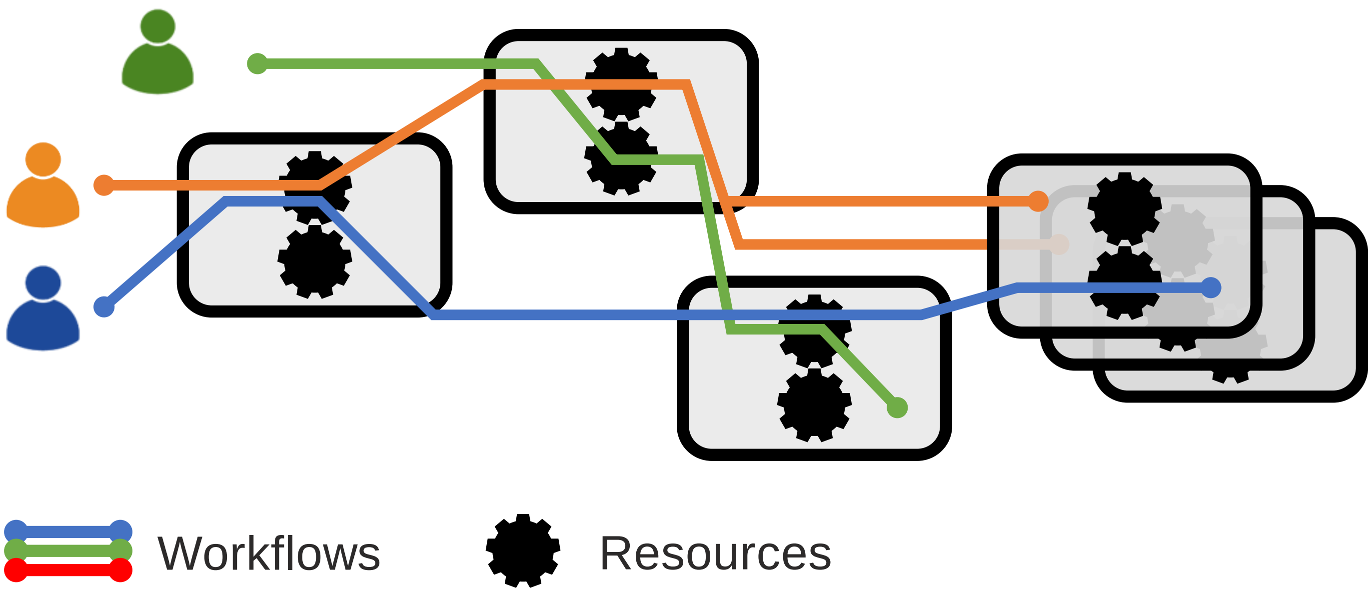
Workflows



Workflows



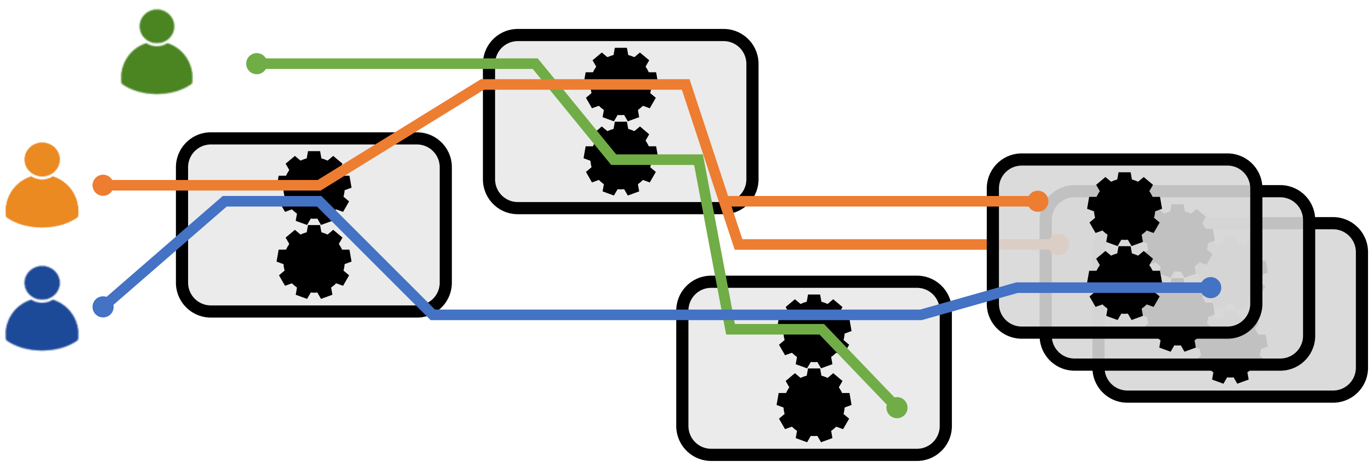
Purpose: cope with diversity of resources



Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

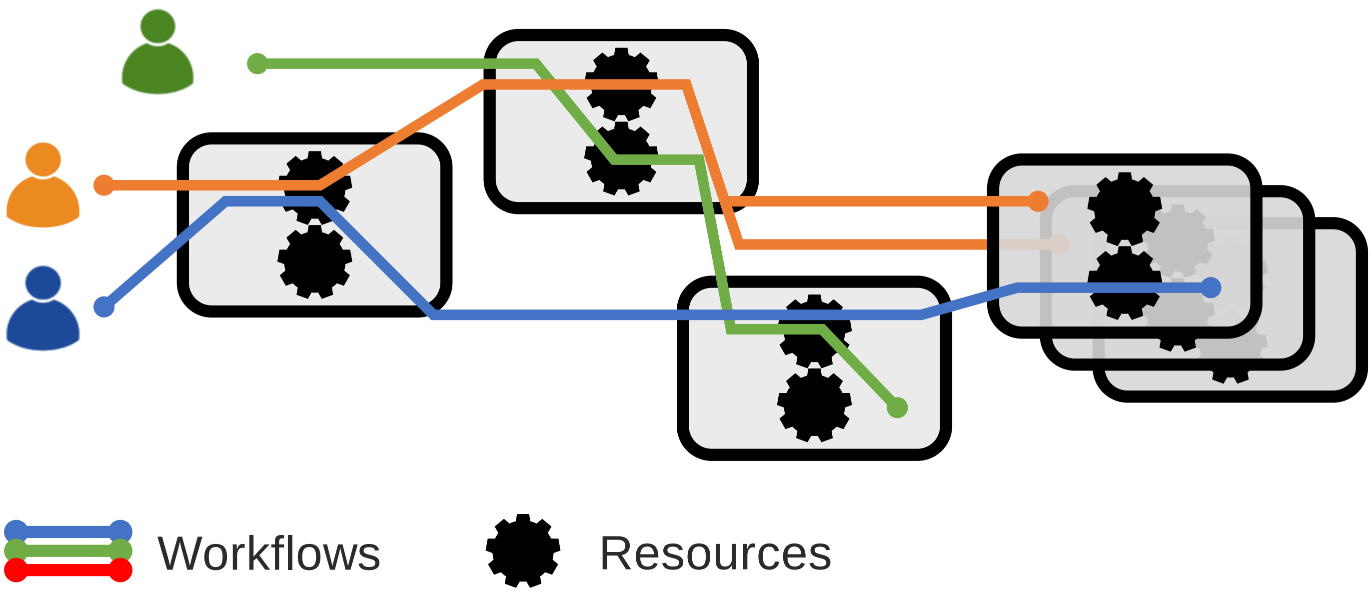


Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

2. Identify culprit workflows



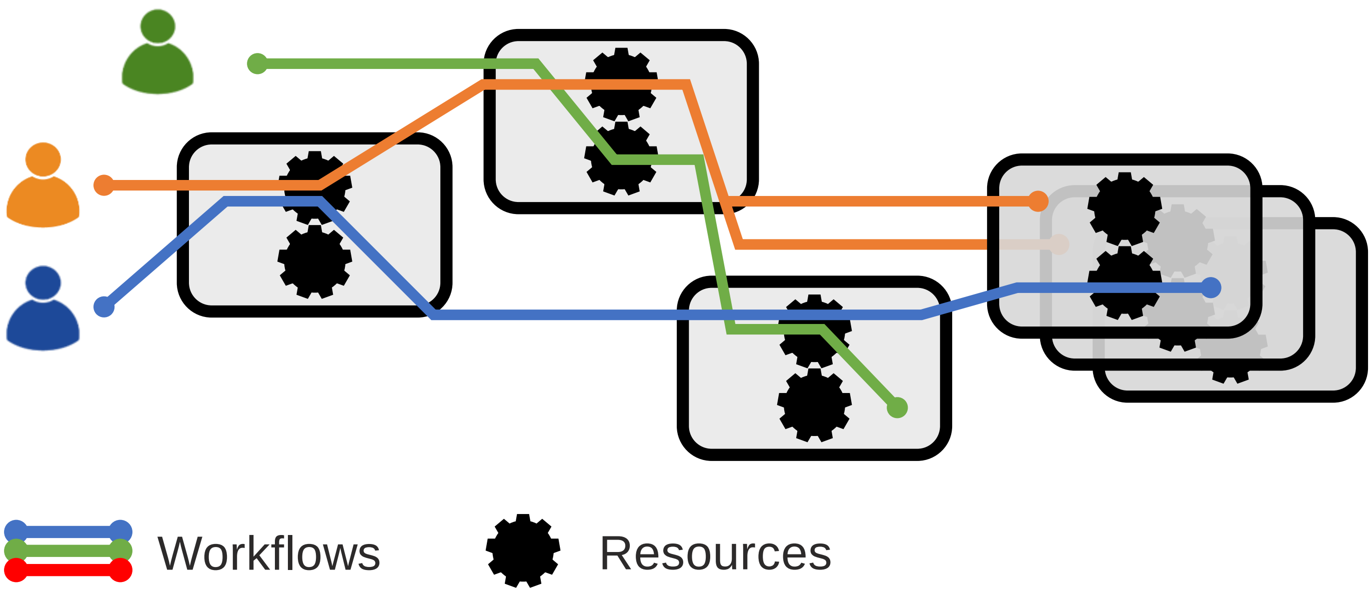
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

Slowdown	Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows



Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

Slowdown	Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows

Load	Fraction of current utilization that we can attribute to each workflow

Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

   Slowdown — Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows

   Load — Fraction of current utilization that we can attribute to each workflow

Purpose: cope with diversity of resources

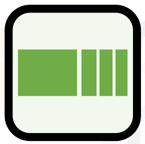
What we need:

1. Identify overloaded resources

Slowdown	Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows

Load	Fraction of current utilization that we can attribute to each workflow



Slowdown (queue time + execute time) / execute time
eg. 100ms queue, 10ms execute
=> slowdown 11

Load time spent executing
eg. 10ms execute
=> load 10

Workflows Resources

Purpose: cope with diversity of resources

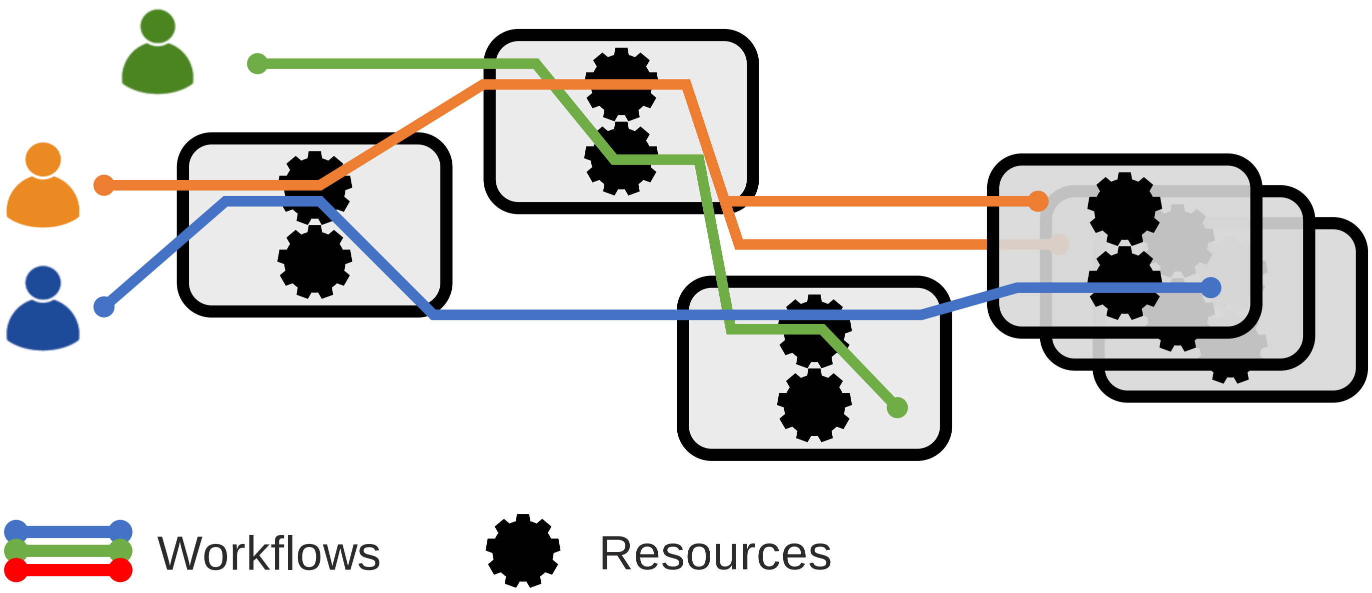
What we need:

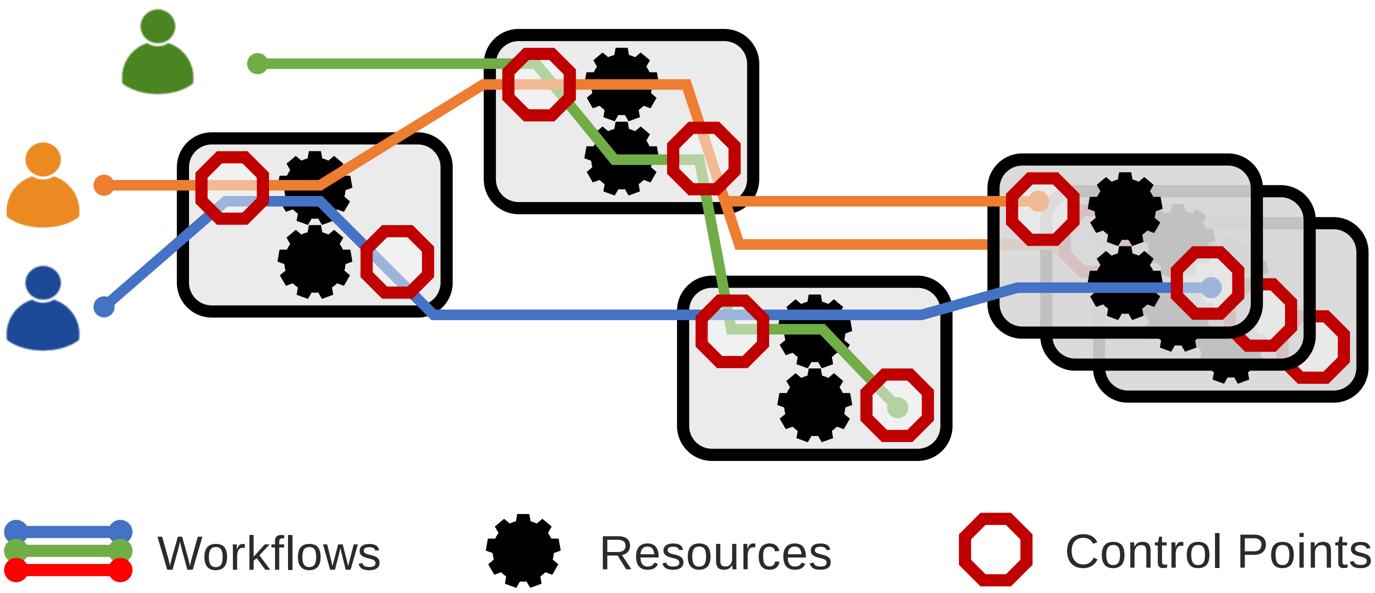
1. Identify overloaded resources

Slowdown	Ratio of how slow the resource is now compared to its baseline performance with no contention.

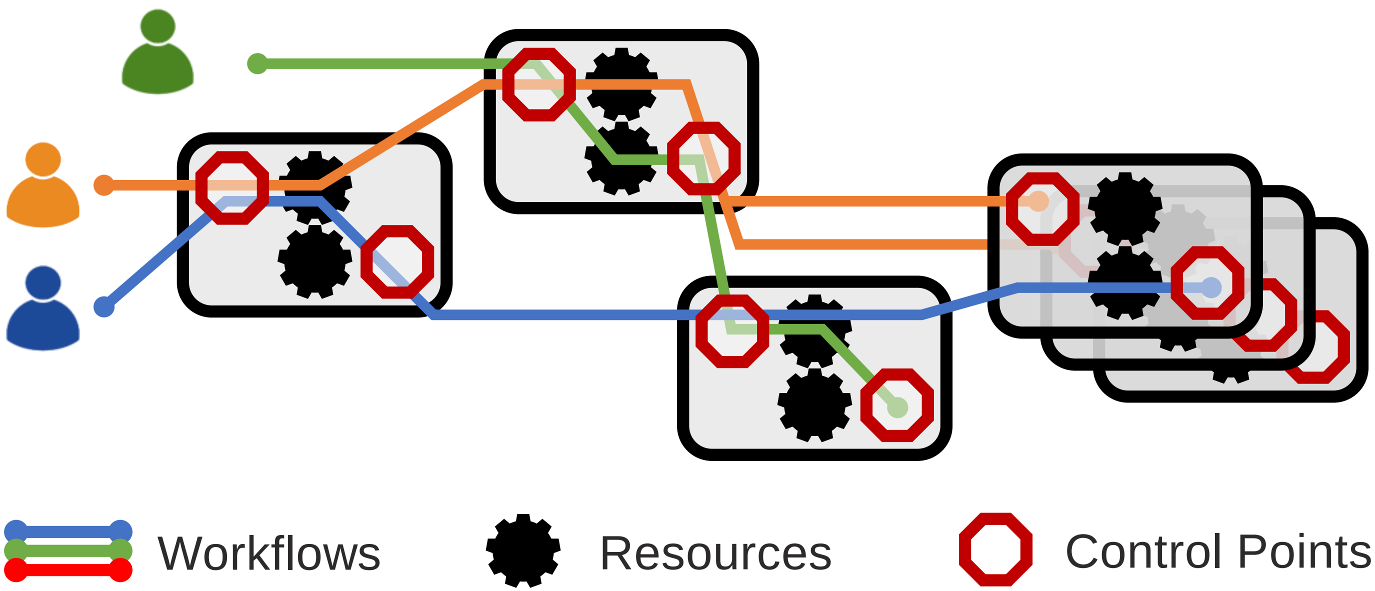
2. Identify culprit workflows

Load	Fraction of current utilization that we can attribute to each workflow


Workflows
Resources



Goal: enforce resource management decisions

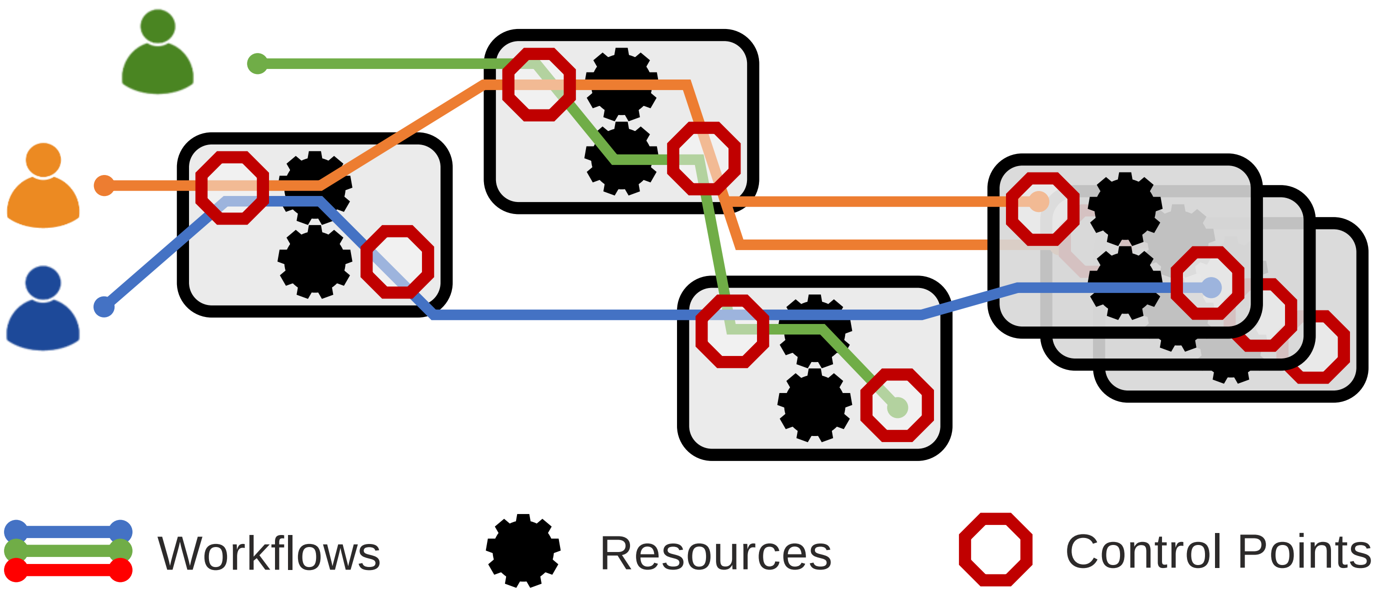


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue

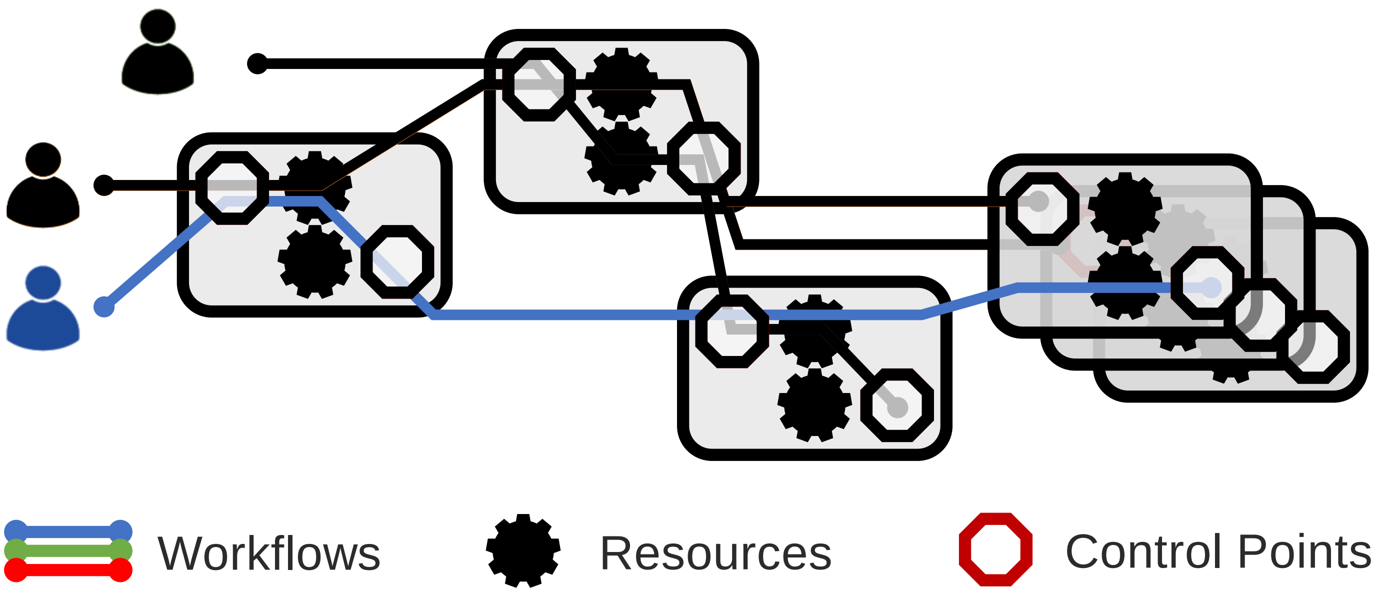


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue

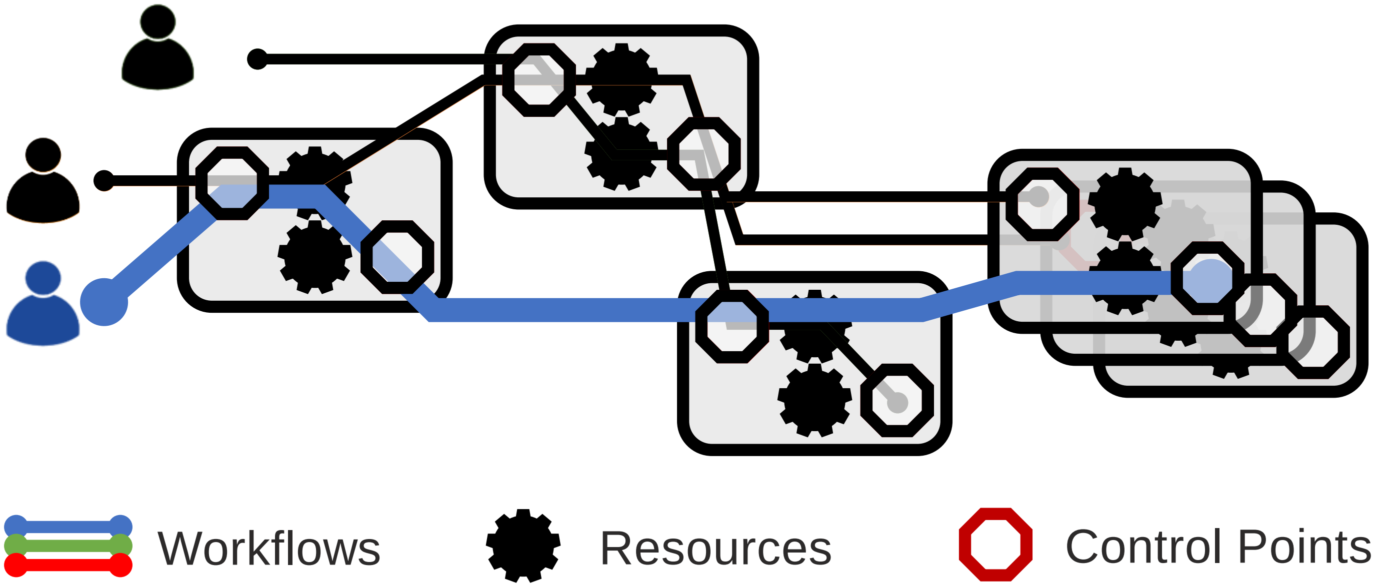


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue

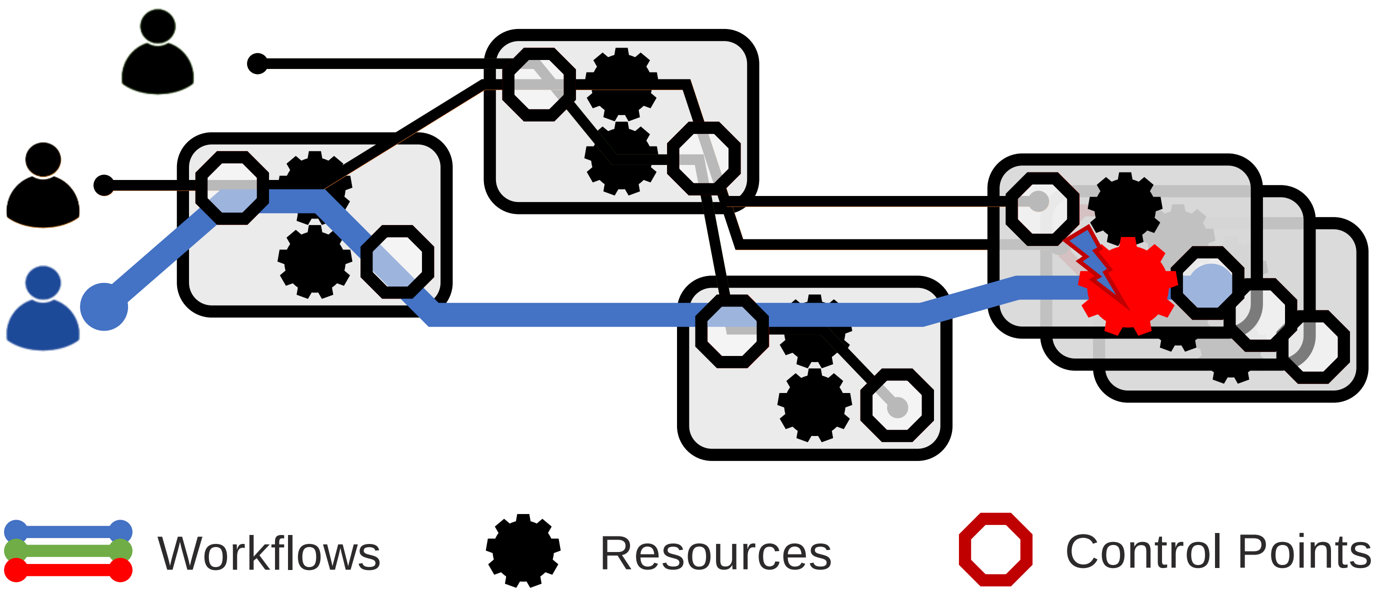


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue

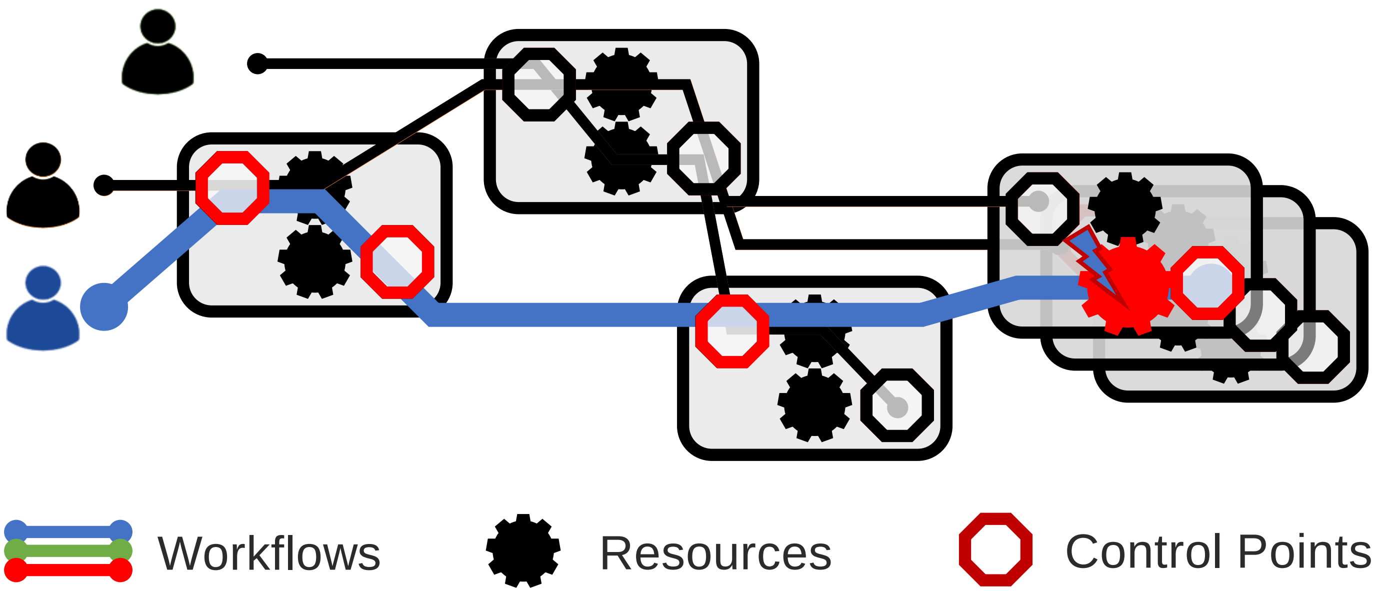


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue

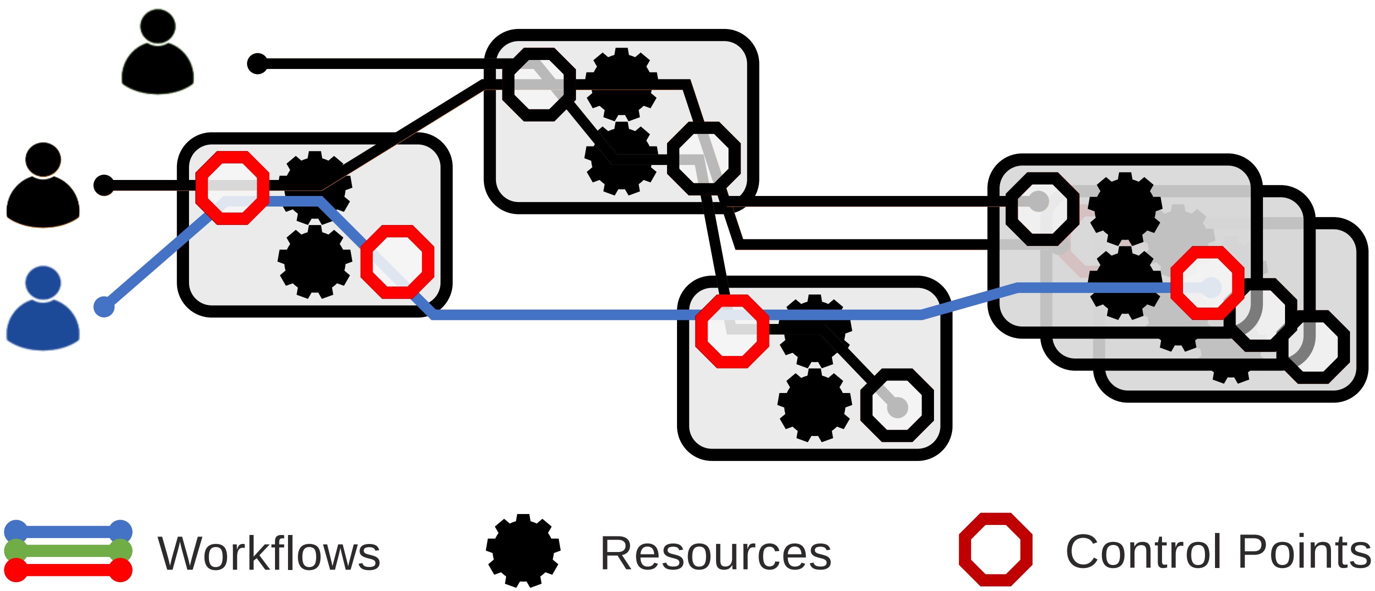


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue

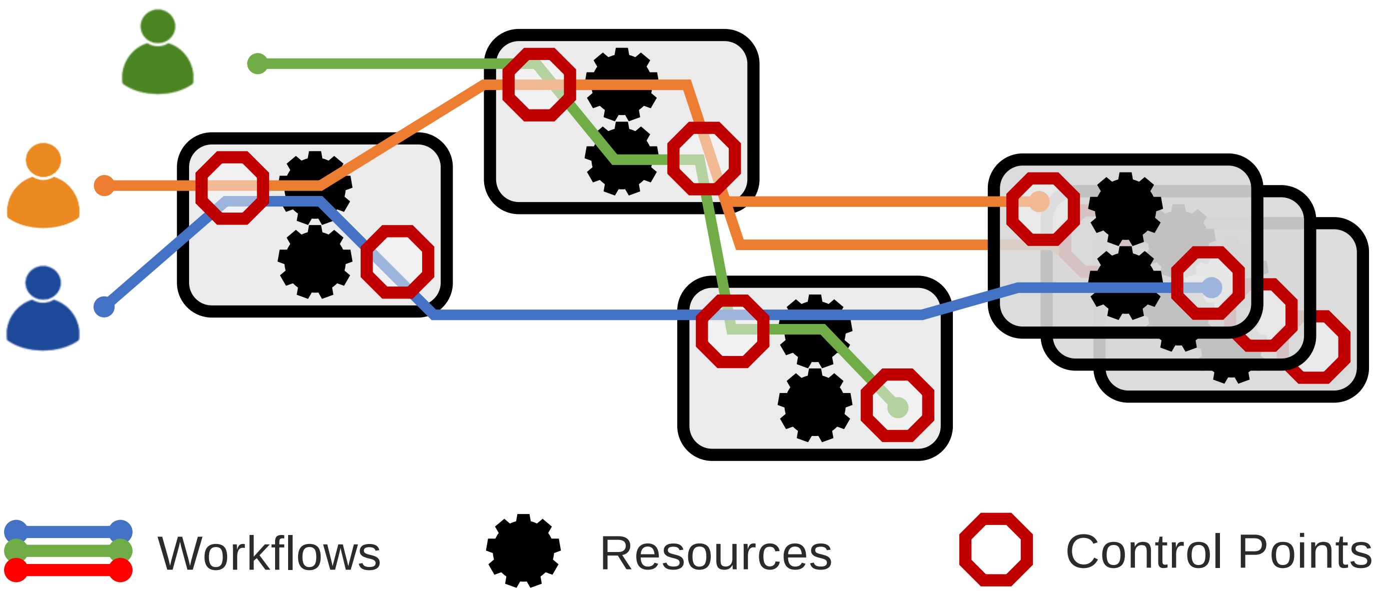


Goal: enforce resource management decisions

Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

- token bucket
- priority queue



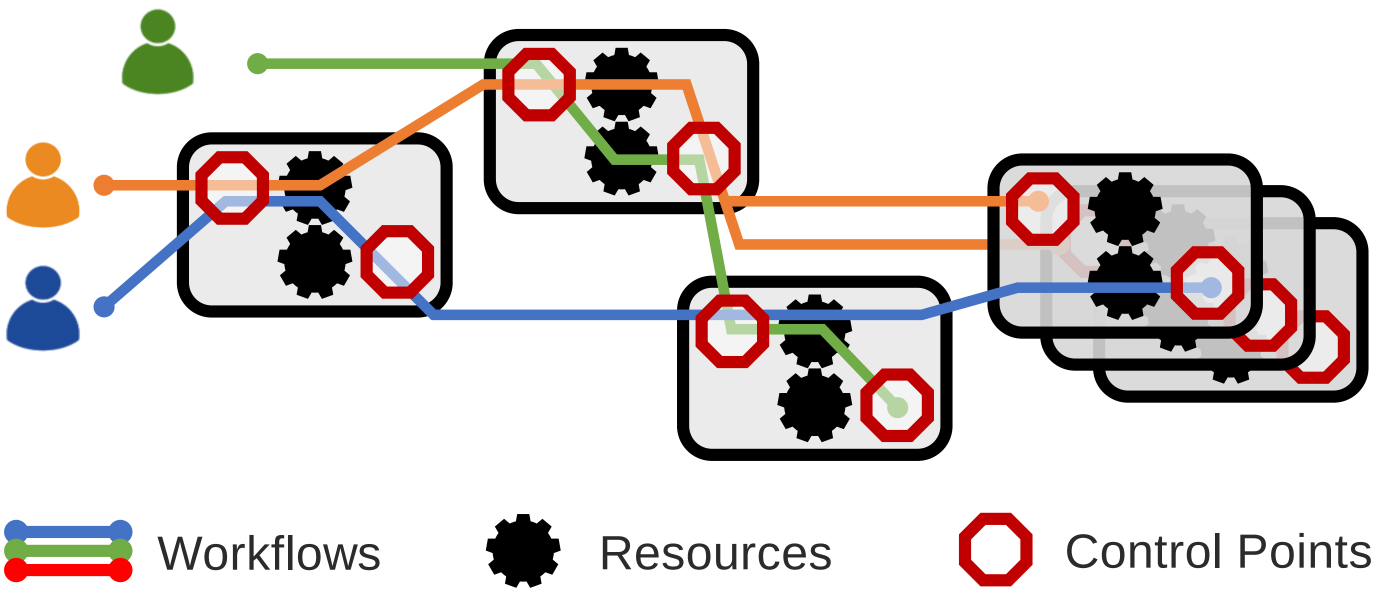
Goal: enforce resource management decisions

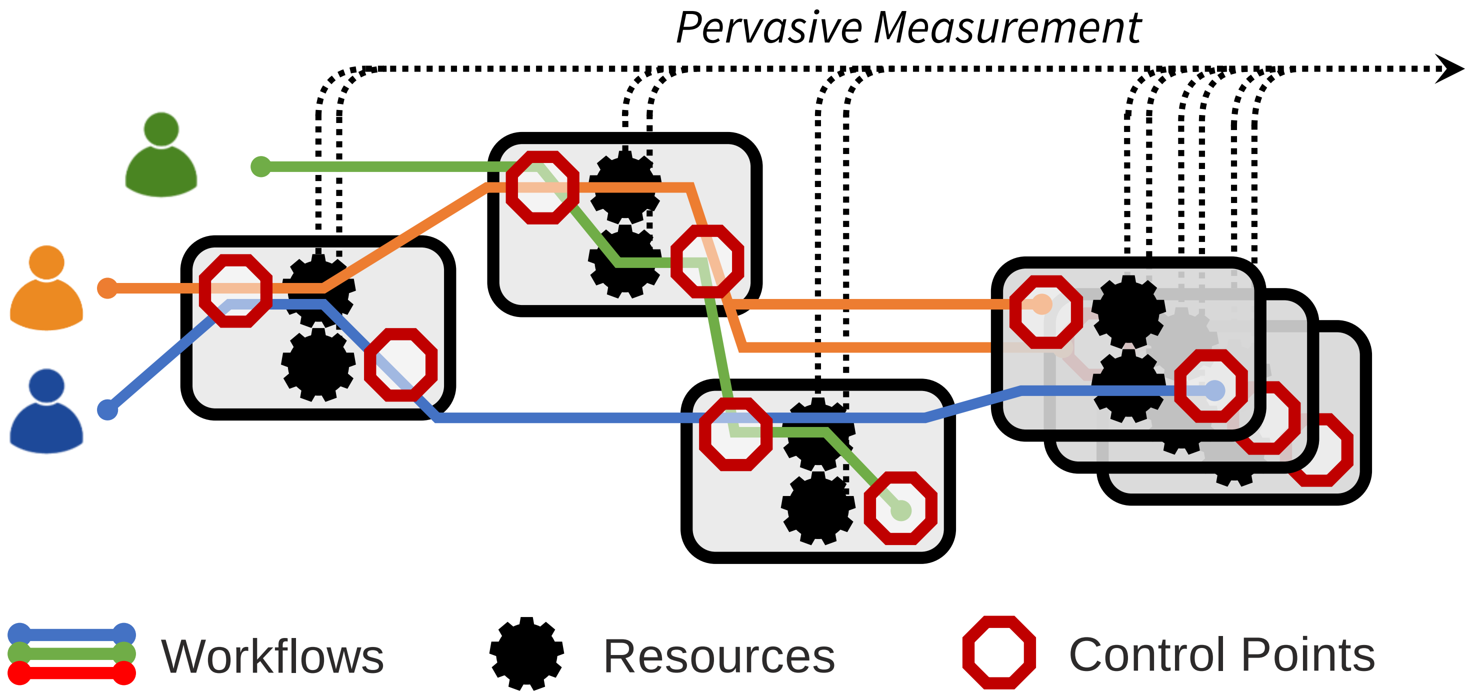
Decoupled from resources

Rate-limits workflows, agnostic to underlying implementation e.g.,

token bucket

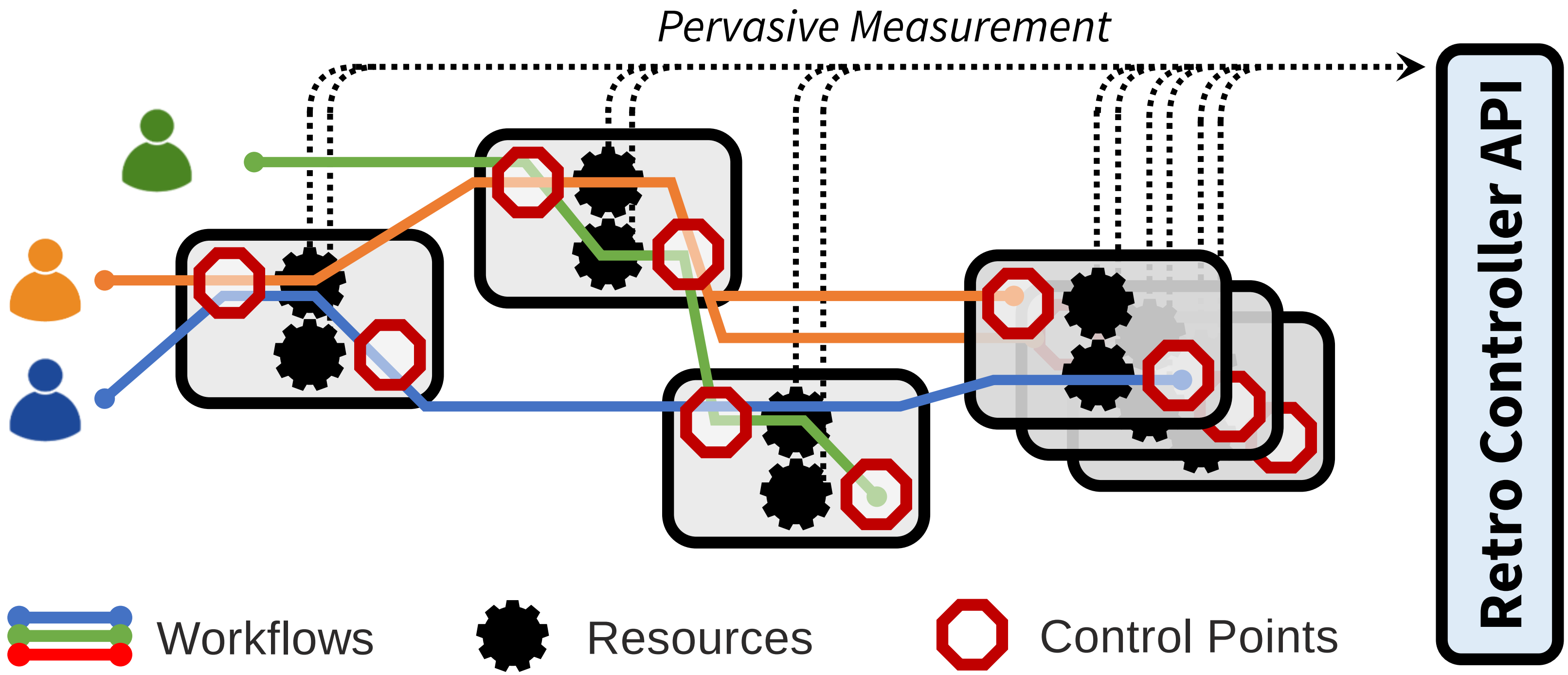
priority queue


Workflows
Resources
Control Points



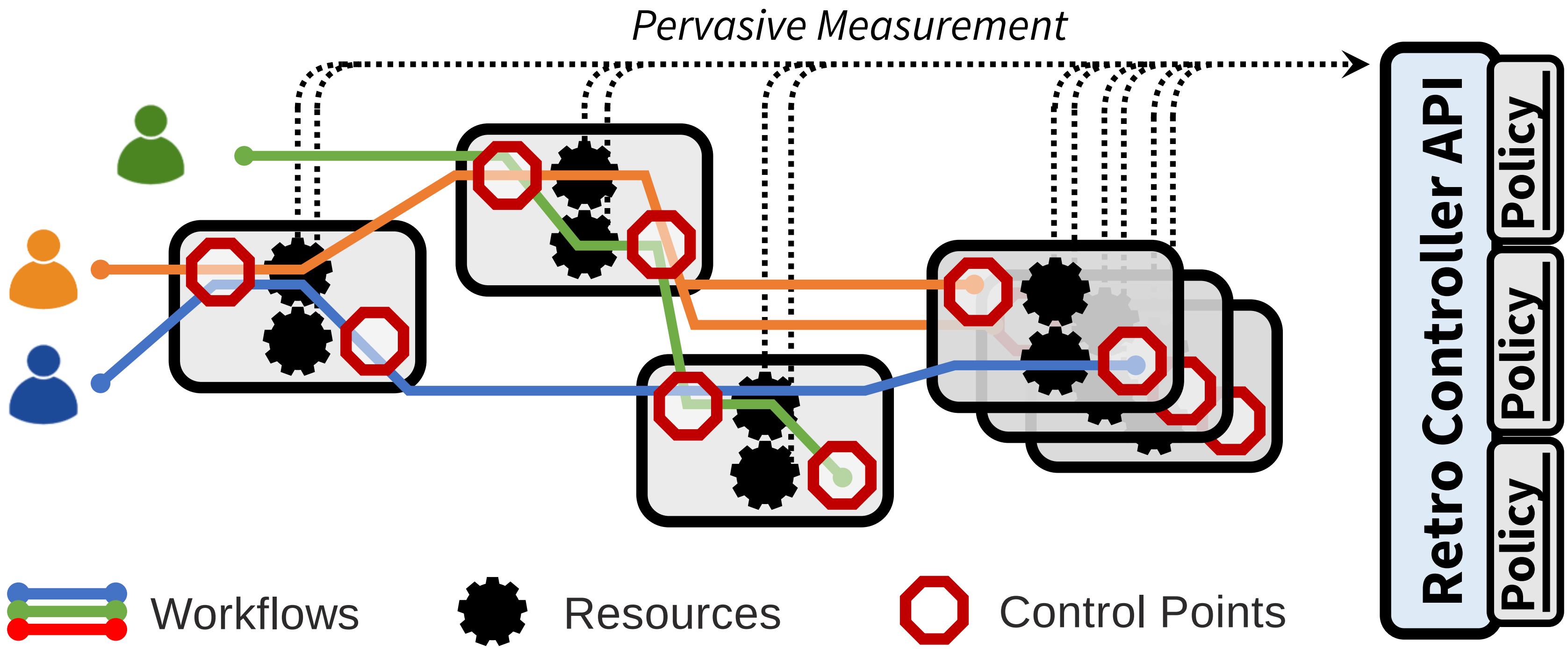
1. Pervasive Measurement

   Aggregated locally then reported centrally once per second



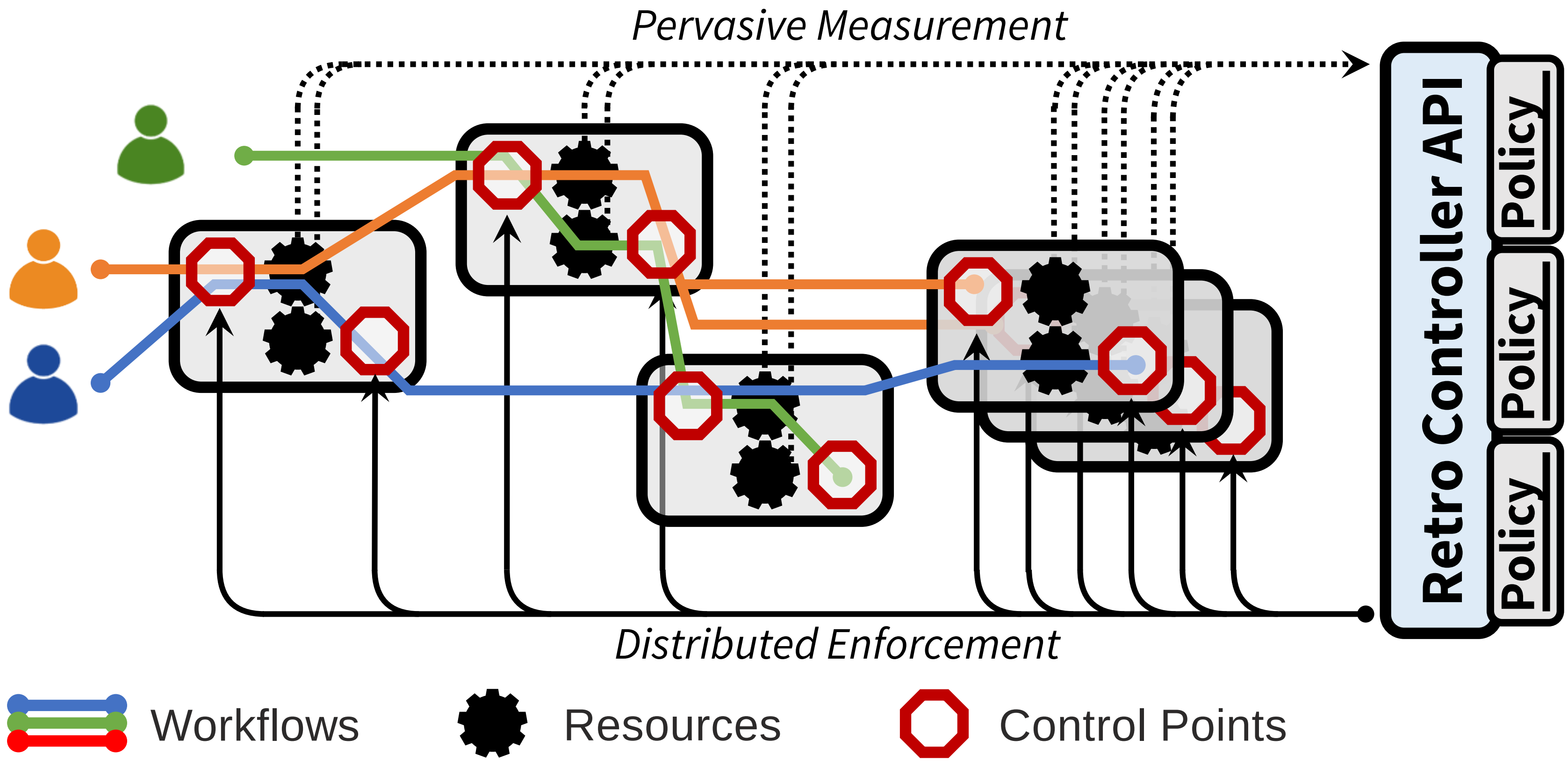
1. Pervasive Measurement
   Aggregated locally then reported centrally once per second

2. Centralized Controller
   Global, abstracted view of the system



1. Pervasive Measurement
   Aggregated locally then reported centrally once per second

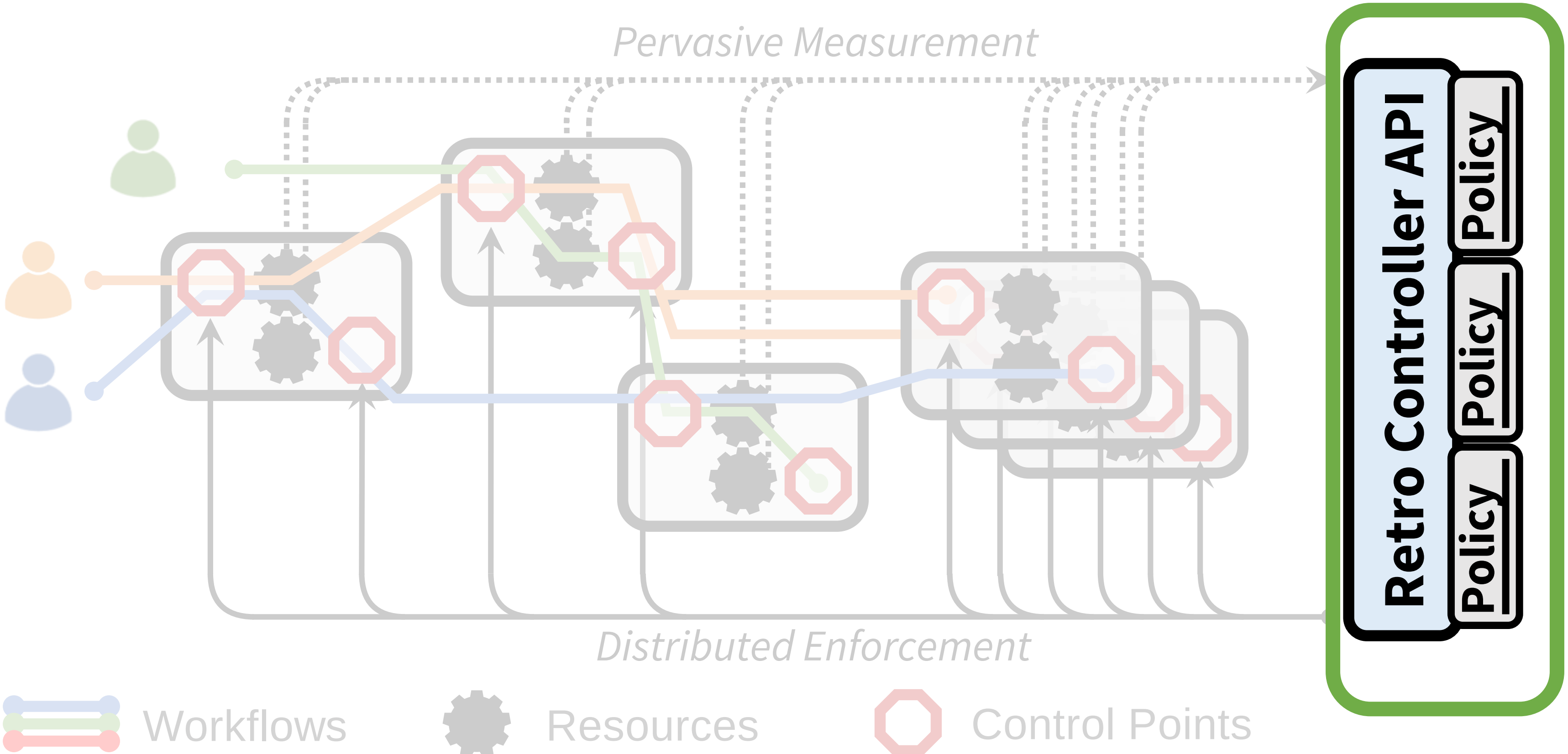
2. Centralized Controller
   Global, abstracted view of the system
   Policies run in continuous control loop



1. Pervasive Measurement
   Aggregated locally then reported centrally once per second

2. Centralized Controller
   Global, abstracted view of the system
   Policies run in continuous control loop

3. Distributed Enforcement
   Co-ordinates enforcement using distributed token bucket



“Control Plane” for resource management

Global, abstracted view of the system

- Easier to write
- Reusable

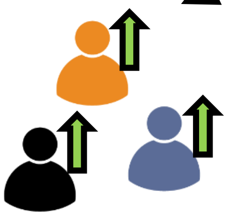
Policy

Example: LatencySLO

Policy

Example: LatencySLO

H High Priority Workflows



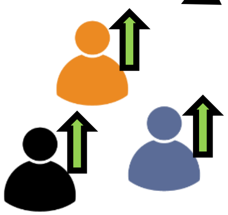
“200ms average request latency”

Policy

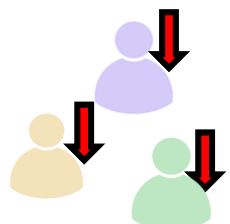
Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows



“200ms average request latency”



(use spare capacity)

Policy

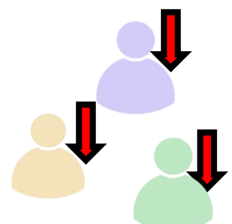
Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

“200ms average request latency”

monitor latencies



(use spare capacity)

Policy

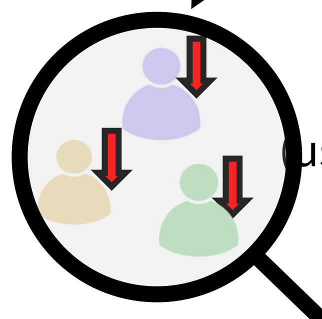
Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

"200ms average request latency"

monitor latencies



(use spare capacity)

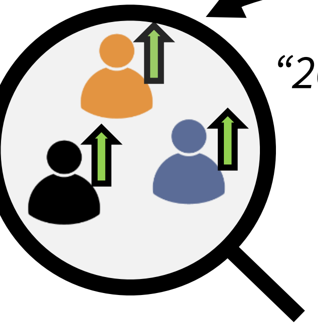
attribute interference

Policy

Example: LatencySLO

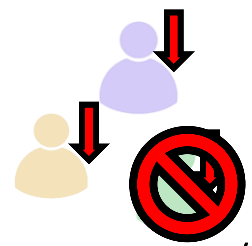
H High Priority Workflows

L Low Priority Workflows



"200ms average request latency"

monitor latencies



(use spare capacity)

throttle interfering workflows

Policy

Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]


foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σ_rsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)


foreach lopri in L            // normalize interference
  interference[lopri] /= Σ_k interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 – α * (miss[W] – 1) * interference[lopri]
  else                // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Policy

Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σ_rsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σ_k interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 – α * (miss[W] – 1) * interference[lopri]
  else                // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri)
```

Policy Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

```
foreach rsrc in resources()    // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L             // calculate low priority workflow interference
  interference[lopri] = Σ_rsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

```
foreach lopri in L             // normalize interference
  interference[lopri] /= Σ_k interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 – α * (miss[W] – 1) * interference[lopri]
  else                 // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Policy

Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

Weight low priority workflows by their interference with *W*

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σrsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σk interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 − α * (miss[W] − 1) * interference[lopri]
  else                        // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Policy

Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

Weight low priority workflows by their interference with *W*

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = -latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σrsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

Throttle low priority workflows proportionally to their weight

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σk interference[k]

foreach lopri in L
  if miss[W] > 1        // throttle
    scalefactor = 1 - α * (miss[W] - 1) * interference[lopri]
  else                  // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Policy

Example: LatencySLO

H High Priority Workflows L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))
```

Weight low priority workflows by their interference with *W*

```
foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σrsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σk interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
```

Throttle low priority workflows proportionally to their weight

```
  else                // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Policy

Example: LatencySLO

H High Priority Workflows   L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

Weight low priority workflows by their interference with *W*

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σrsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σk interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 - α * (miss[W] - 1) * interference[lopri]
  else                // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Throttle low priority workflows proportionally to their weight

Policy

Example: LatencySLO

H High Priority Workflows

L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

Weight low priority workflows by their interference with *W*

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σ_rsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

Throttle low priority workflows proportionally to their weight

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σ_k interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 – α * (miss[W] – 1) * interference[lopri]
  else                    // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Policy

Example: LatencySLO

H High Priority Workflows L Low Priority Workflows

Select the high priority workflow *W* with worst performance

```
foreach candidate in H
  miss[candidate] = latency(candidate) / guarantee[candidate]
W = candidate in H with max miss[candidate]
```

Weight low priority workflows by their interference with *W*

```
foreach rsrc in resources()   // calculate importance of each resource for hipri
  importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))

foreach lopri in L            // calculate low priority workflow interference
  interference[lopri] = Σ_rsrc importance[rsrc] * load(lopri, rsrc) / load(rsrc)
```

Throttle low priority workflows proportionally to their weight

```
foreach lopri in L            // normalize interference
  interference[lopri] /= Σ_k interference[k]

foreach lopri in L
  if miss[W] > 1  // throttle
    scalefactor = 1 – α * (miss[W] – 1) * interference[lopri]
  else                // release
    scalefactor = 1 + β

  foreach cpoint in controlpoints()  // apply new rates
    set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

Other types of policy…

Other types of policy…

Policy Bottleneck Fairness

- Detect most overloaded resource
- Fair-share resource between tenants using it

Other types of policy…

Policy Bottleneck Fairness

- Detect most overloaded resource
- Fair-share resource between tenants using it

Policy Dominant Resource Fairness

- Estimate demands and capacities from measurements

Other types of policy…

Policy Bottleneck Fairness

- Detect most overloaded resource
- Fair-share resource between tenants using it

Policy Dominant Resource Fairness

- Estimate demands and capacities from measurements

Concise

Any resources can be bottleneck (policy doesn’t care)

Not system specific

Retro Evaluation

Instrumentation

Instrumentation

- Retro implementation in Java
  - Instrumentation Library
  - Central controller implementation

Instrumentation

Retro implementation in Java

Instrumentation Library

Central controller implementation

To enable Retro

- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Instrumentation

Retro implementation in Java

Instrumentation Library

Central controller implementation

To enable Retro

- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Overheads

Instrumentation

Retro implementation in Java

Instrumentation Library

Central controller implementation

To enable Retro

- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Overheads

Resource instrumentation automatic using AspectJ

Instrumentation

Retro implementation in Java

- Instrumentation Library
- Central controller implementation

To enable Retro

- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Overheads

- Resource instrumentation automatic using AspectJ
- Overall 50-200 lines per system to modify RPCs

Instrumentation

Retro implementation in Java

Instrumentation Library

Central controller implementation

To enable Retro

- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Overheads

Resource instrumentation automatic using AspectJ

Overall 50-200 lines per system to modify RPCs

Retro overhead: max 1-2% on throughput, latency

Experiments

Experiments

Systems

YARN

MapReduce

ZooKeeper

HBase

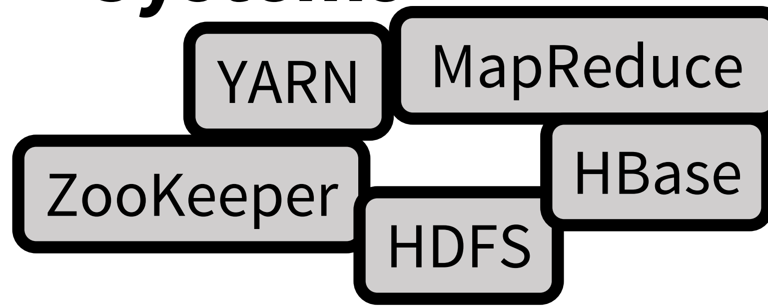
HDFS

Experiments

Workflows

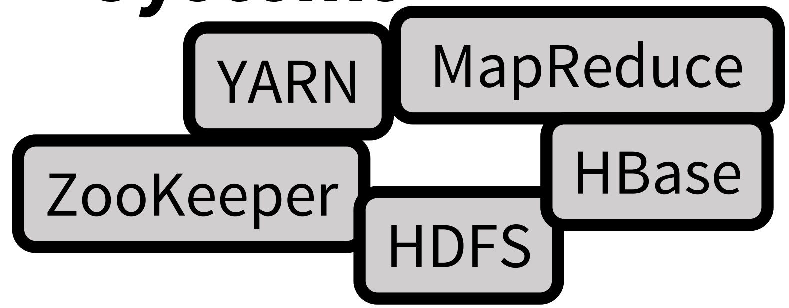
MapReduce Jobs (HiBench)
HBase (YCSB)
HDFS clients
Background Data Replication
Background Heartbeats

Systems



Experiments

Systems



Workflows

MapReduce Jobs (HiBench)
HBase (YCSB)
HDFS clients
Background Data Replication
Background Heartbeats

Resources

CPU, Disk, Network (All systems)
Locks, Queues (HDFS, HBase)

Experiments

Workflows

MapReduce Jobs (HiBench)
HBase (YCSB)
HDFS clients
Background Data Replication
Background Heartbeats

Resources

CPU, Disk, Network (All systems)
Locks, Queues (HDFS, HBase)

Systems

YARN
MapReduce
ZooKeeper
HBase
HDFS

Policies

Bottleneck Fairness
LatencySLO
Dominant Resource Fairness

Experiments

Workflows

MapReduce Jobs (HiBench)
HBase (YCSB)
HDFS clients
Background Data Replication
Background Heartbeats

Systems

YARN
MapReduce
ZooKeeper
HBase
HDFS

Resources

CPU, Disk, Network (All systems)
Locks, Queues (HDFS, HBase)

Policies

Bottleneck Fairness
LatencySLO
Dominant Resource Fairness

Policies for a mixture of systems, workflows, and resources

Results on clusters up to 200 nodes

See paper for full experiment results

Experiments

Workflows

MapReduce Jobs (HiBench)
HBase (YCSB)
HDFS clients
Background Data Replication
Background Heartbeats

Systems

YARN
MapReduce
ZooKeeper
HBase
HDFS

Policies

LatencySLO
Bottleneck Fairness
Dominant Resource Fairness

Resources

CPU, Disk, Network (All systems)
Locks, Queues (HDFS, HBase)

Policies for a mixture of systems, workflows, and resources

Results on clusters up to 200 nodes

See paper for full experiment results

This talk LatencySLO policy results

Experiments

Systems

YARN

MapReduce

ZooKeeper

HBase

HDFS

Workflows

MapReduce Jobs (HiBench)
HBase (YCSB)
HDFS clients
Background Data Replication
Background Heartbeats

Policies

LatencySLO

Bottleneck Fairness

Dominant Resource Fairness

Resources

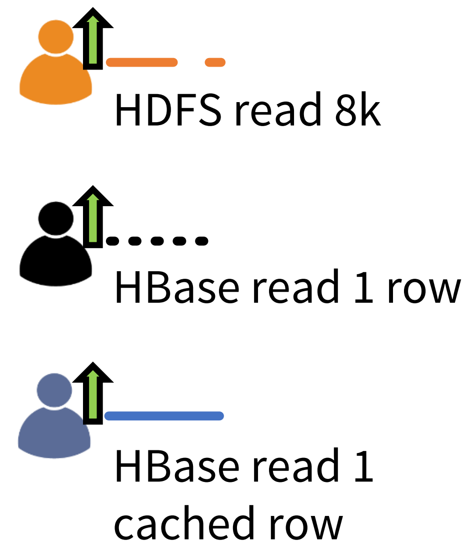
CPU, Disk, Network (All systems)
Locks, Queues (HDFS, HBase)

Policies for a mixture of systems, workflows, and resources

Results on clusters up to 200 nodes

See paper for full experiment results

This talk LatencySLO policy results


HDFS read 8k
HBase read 1 row
HBase read 1 cached row

HBase Table Scan

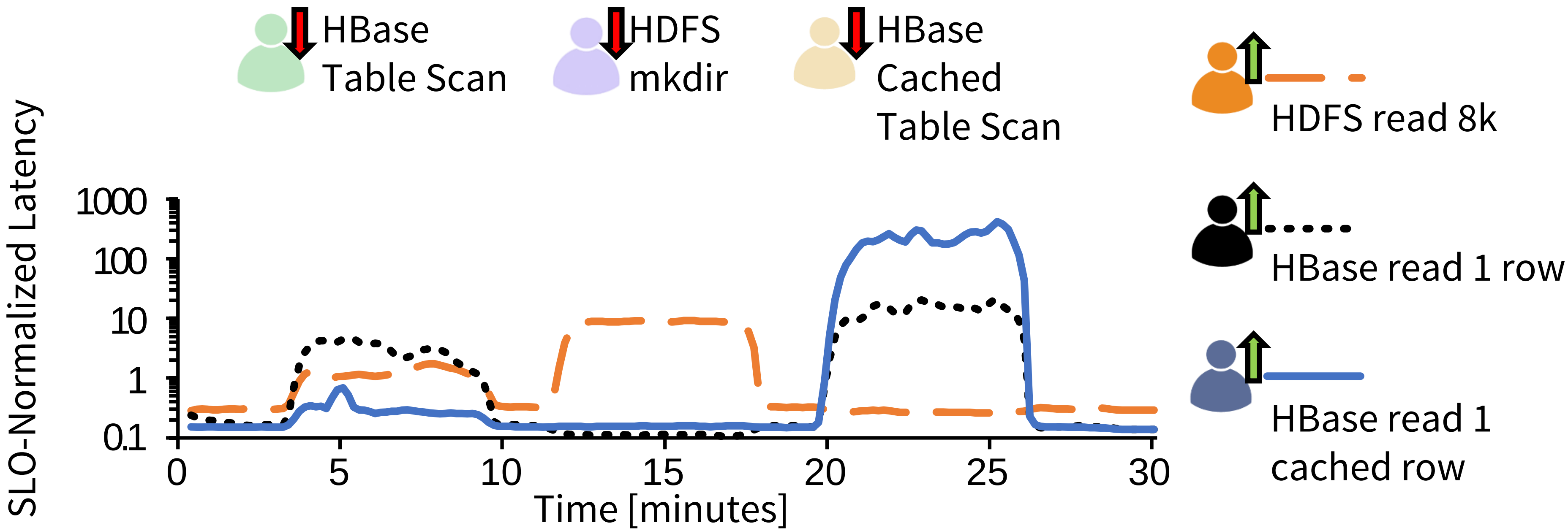
HDFS mkdir

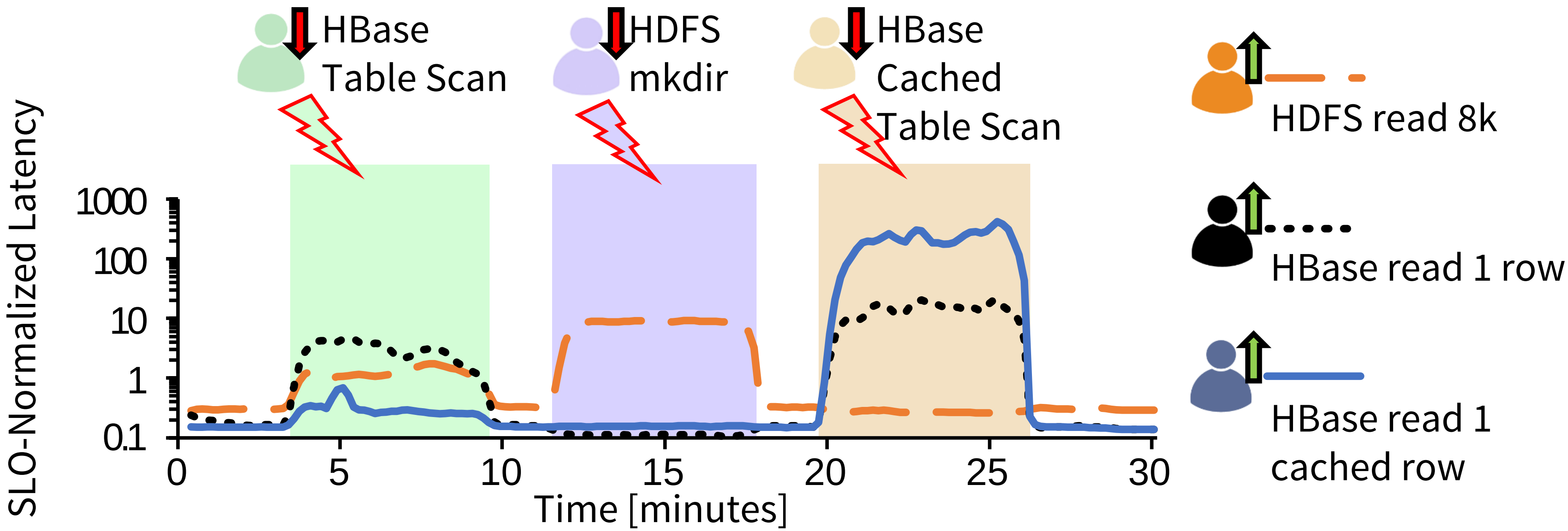
HBase Cached Table Scan

HDFS read 8k

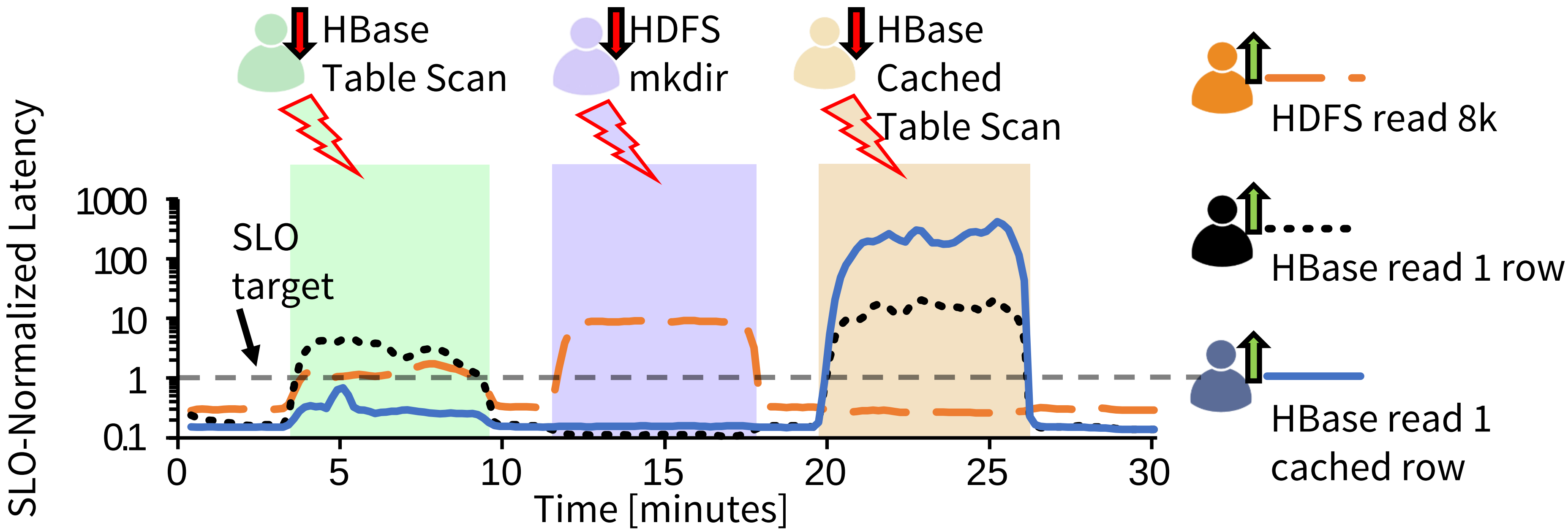
HBase read 1 row

HBase read 1 cached row

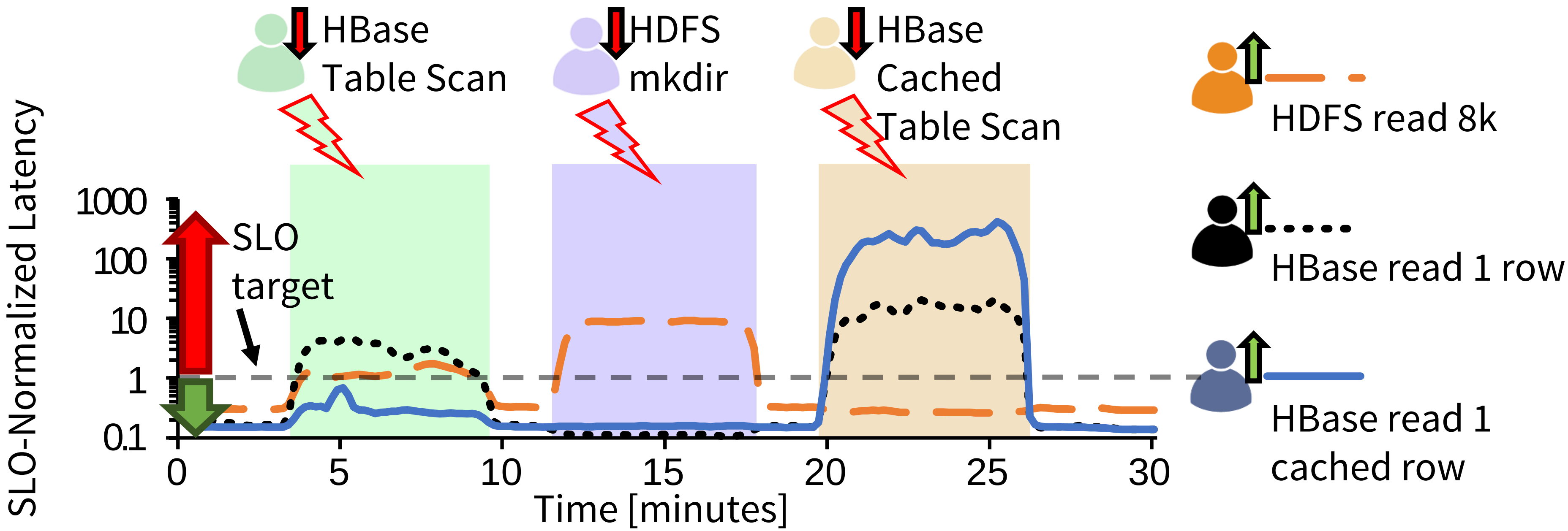

HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]

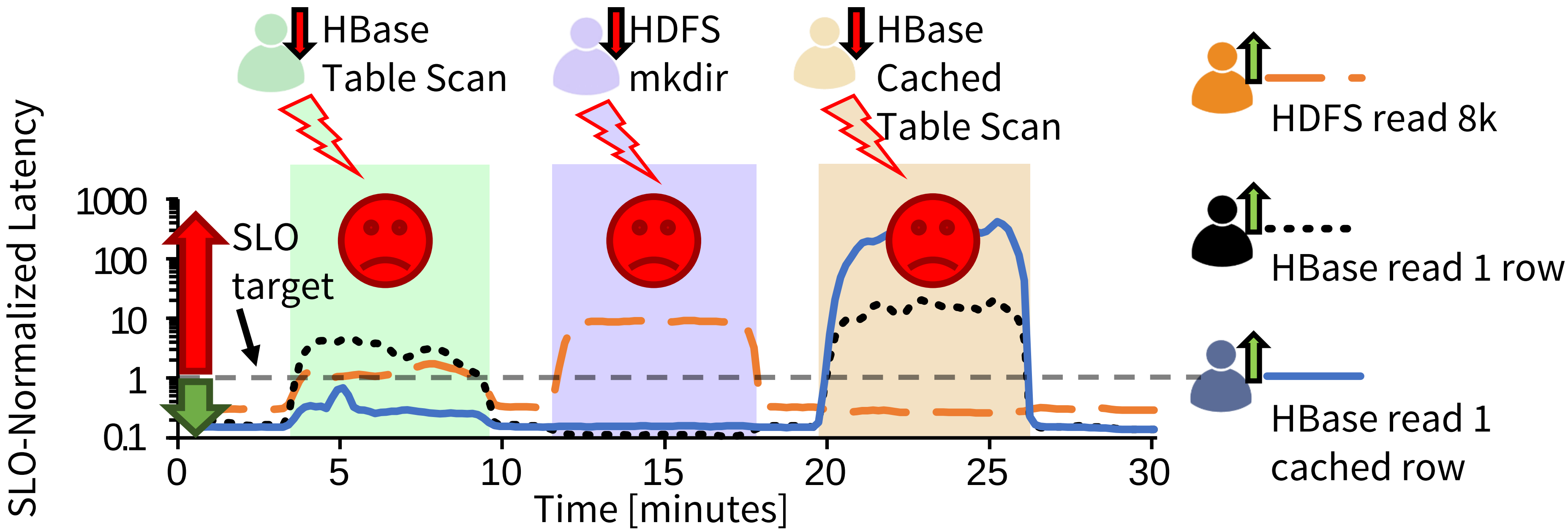


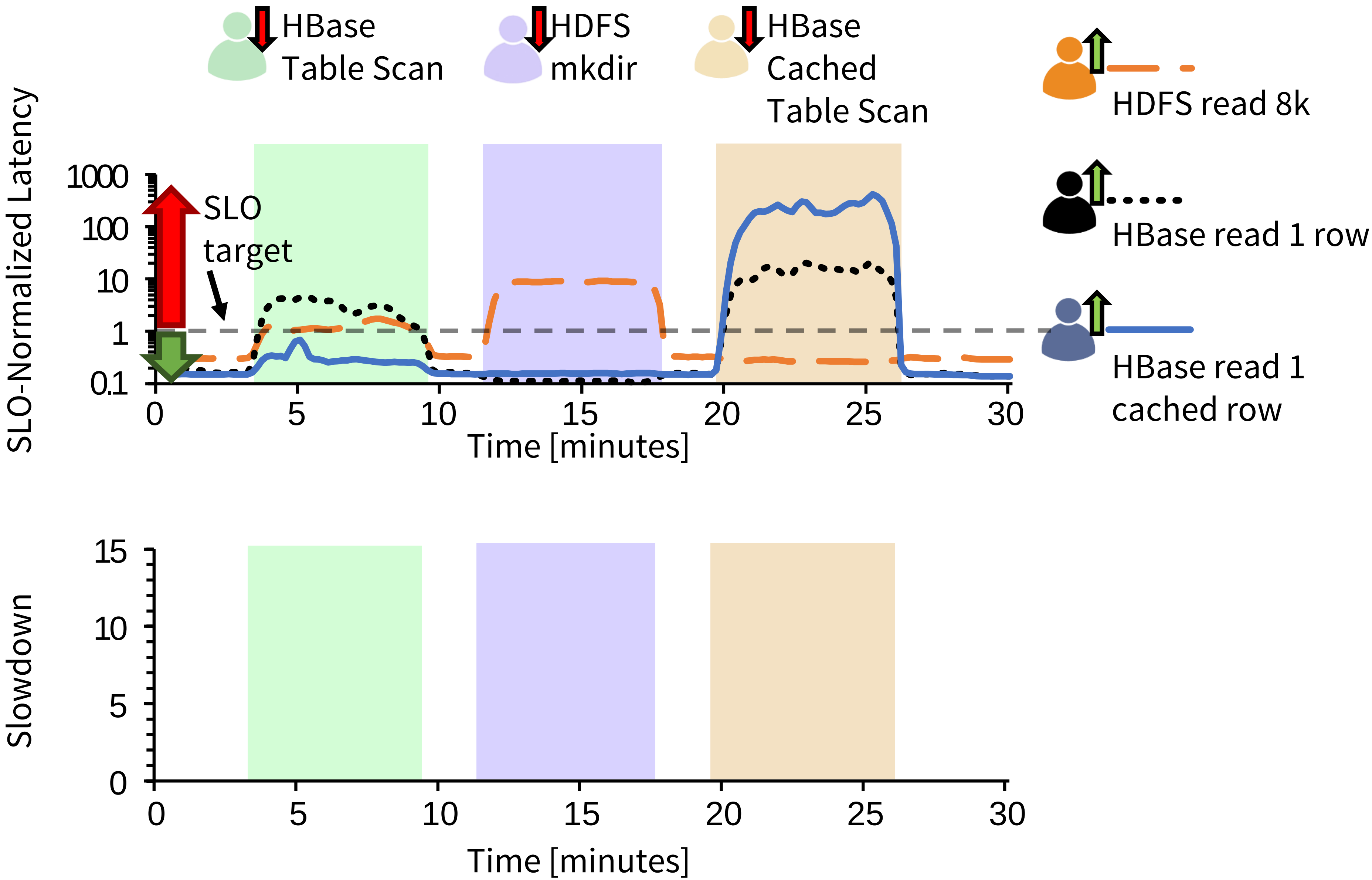
HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]

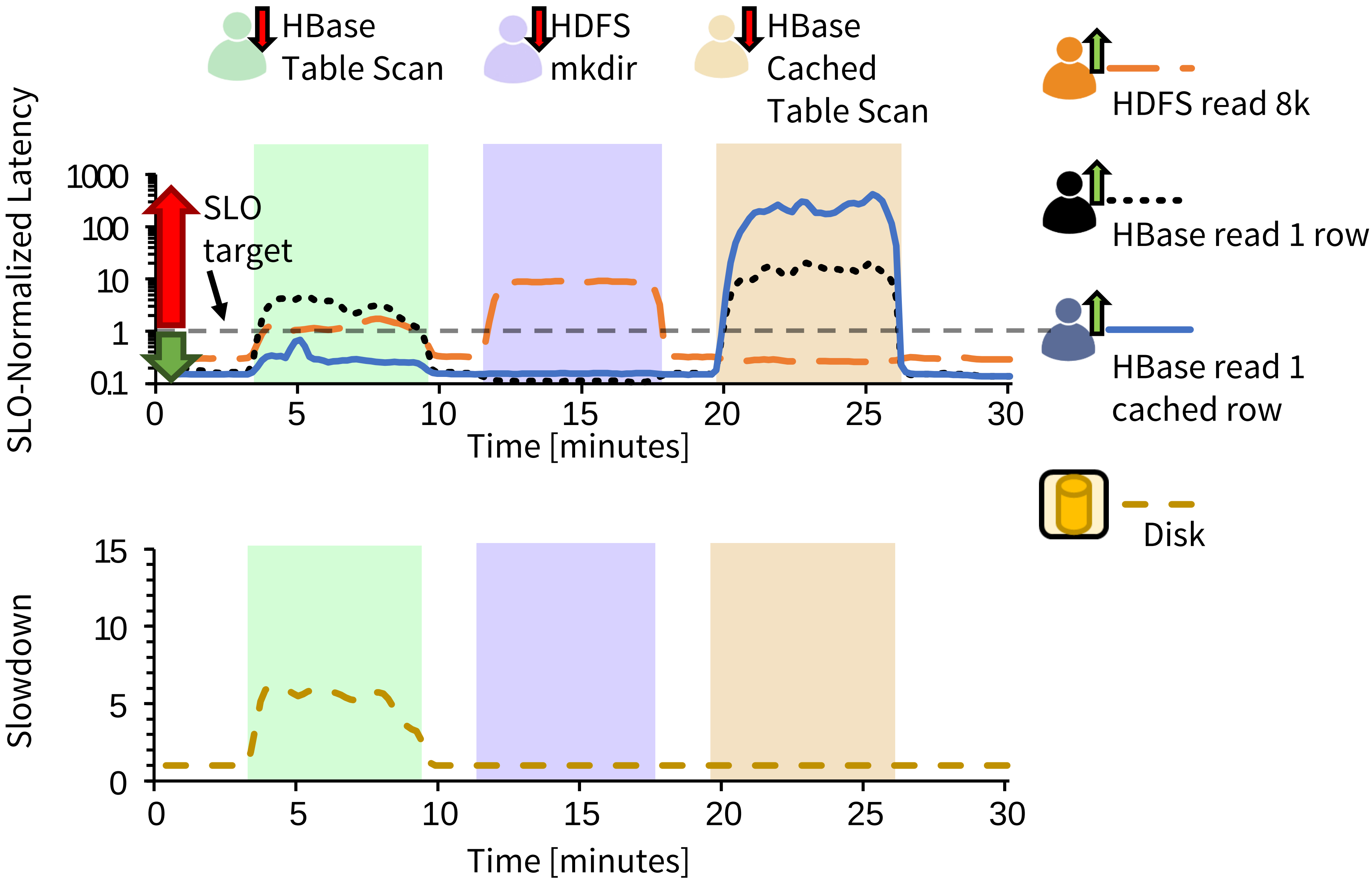


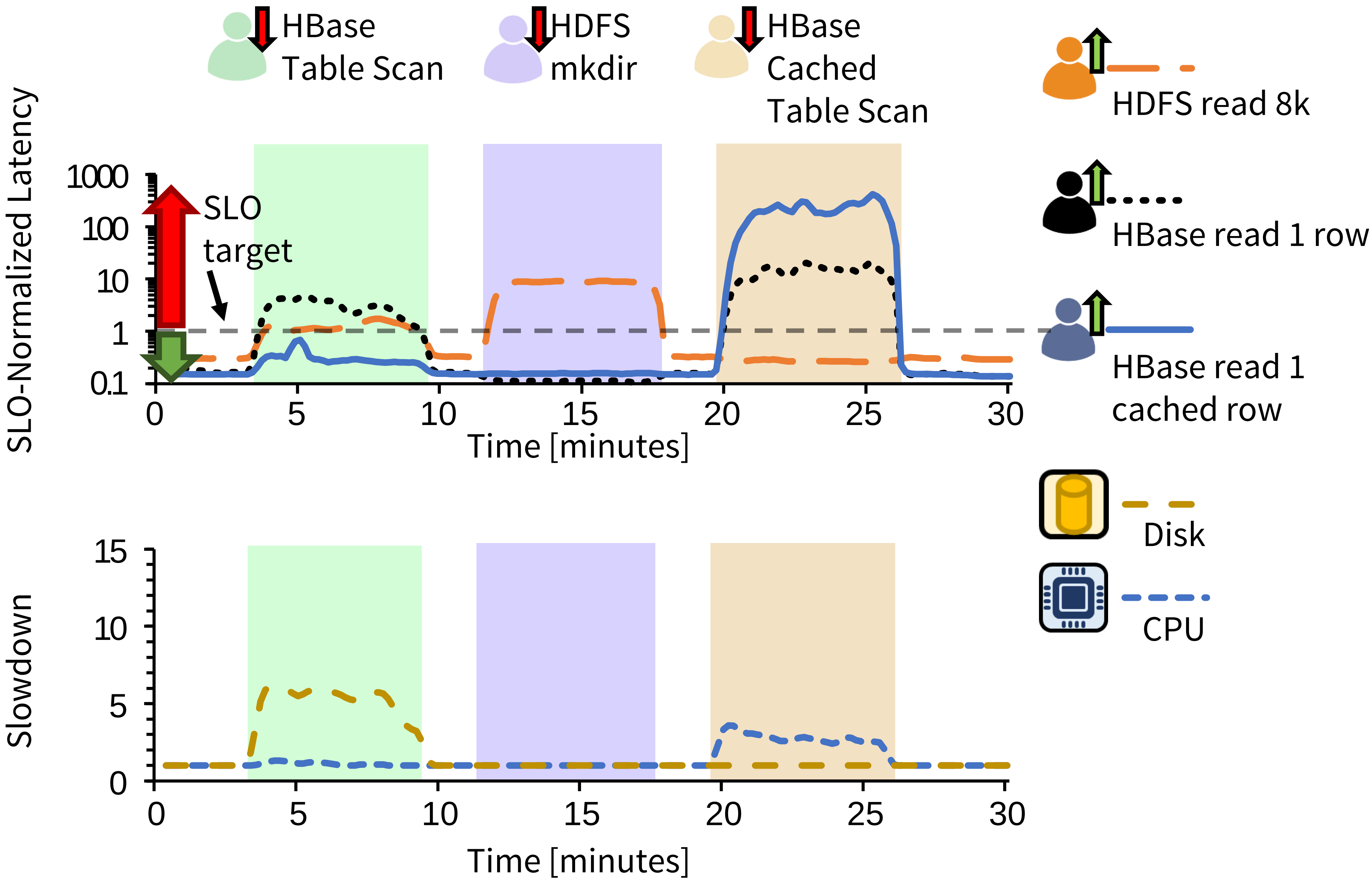
HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO
target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]

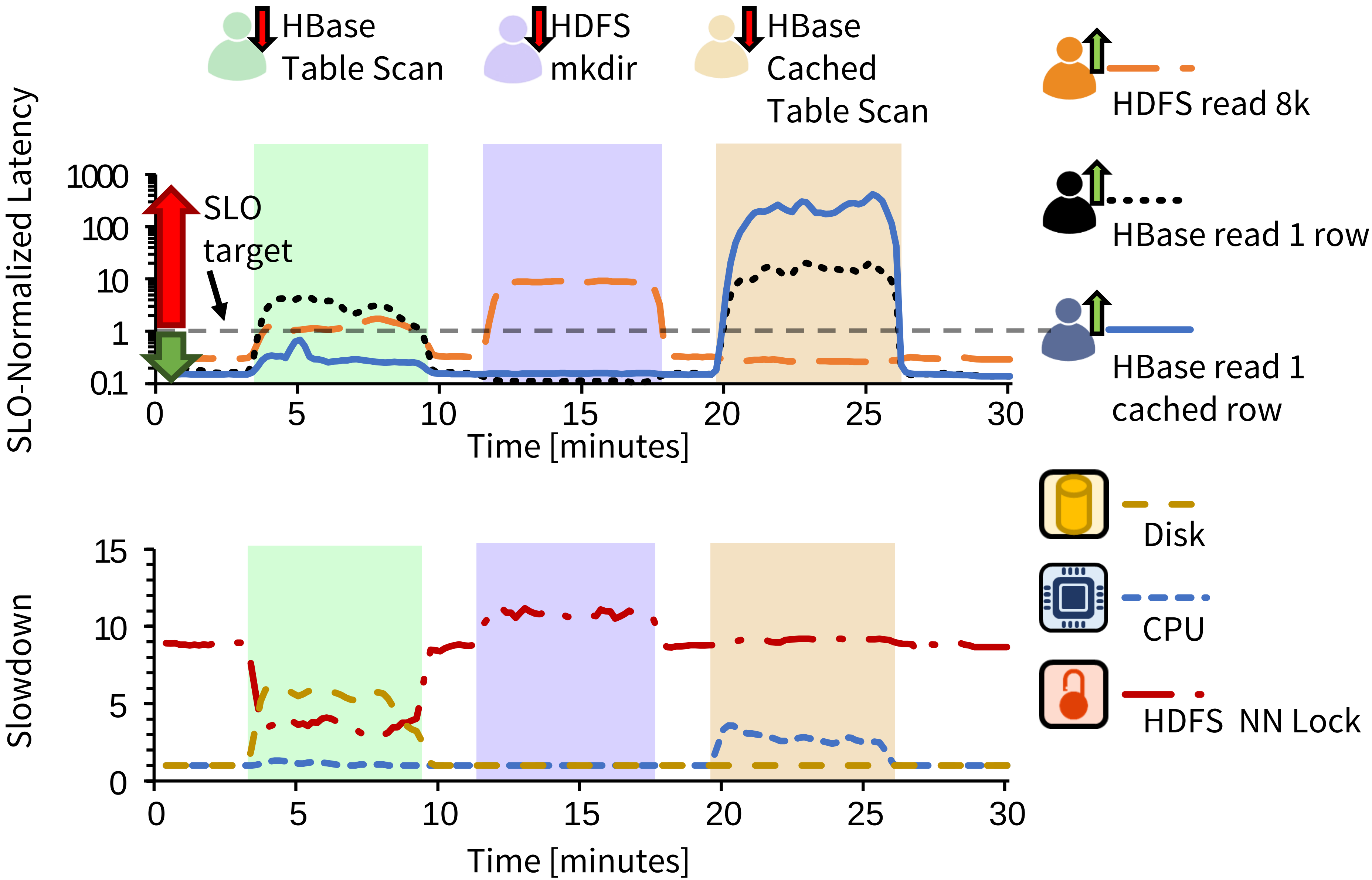

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]


HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]

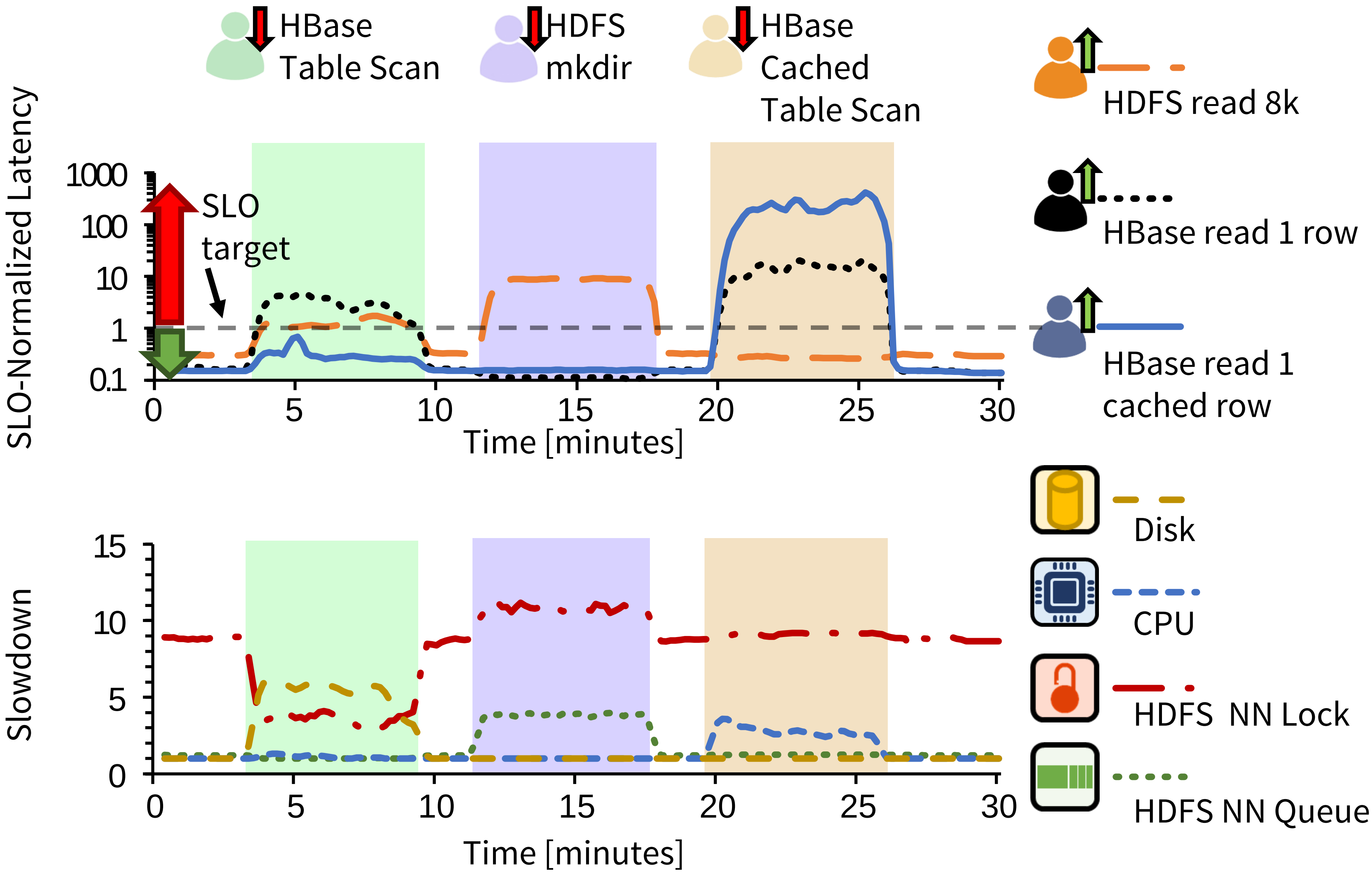

HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO
target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
Slowdown
15
10
5
0
0
5
10
15
20
25
30
Time [minutes]

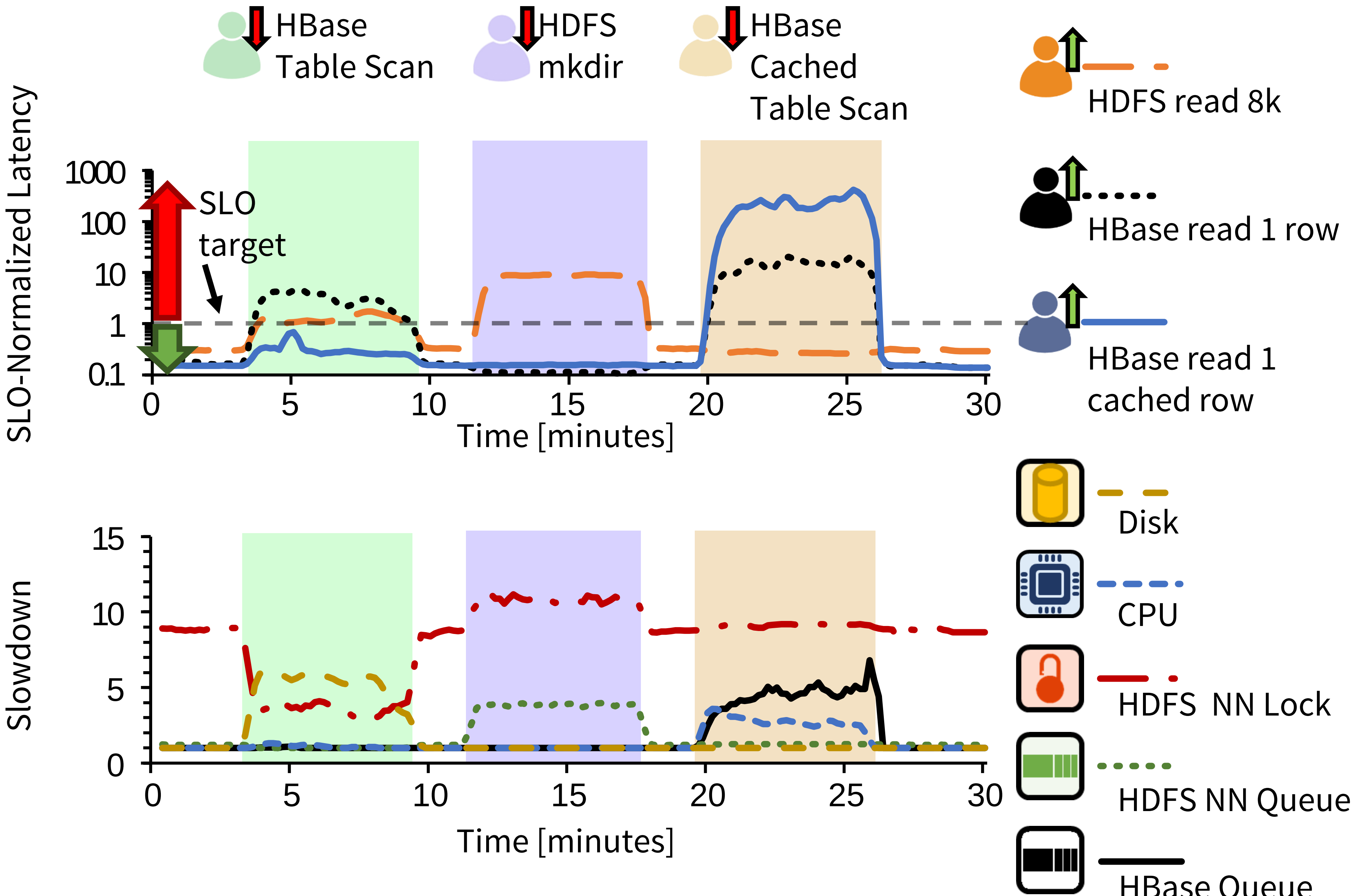

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
Disk
SLO-Normalized Latency
1000
100
10
1
0.1
SLO target
0
5
10
15
20
25
30
Time [minutes]
Slowdown
15
10
5
0
0
5
10
15
20
25
30
Time [minutes]


HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO-Normalized Latency
1000
100
10
1
0.1
SLO
target
0
5
10
15
20
25
30
Time [minutes]
Disk
CPU
Slowdown
15
10
5
0
0
5
10
15
20
25
30
Time [minutes]

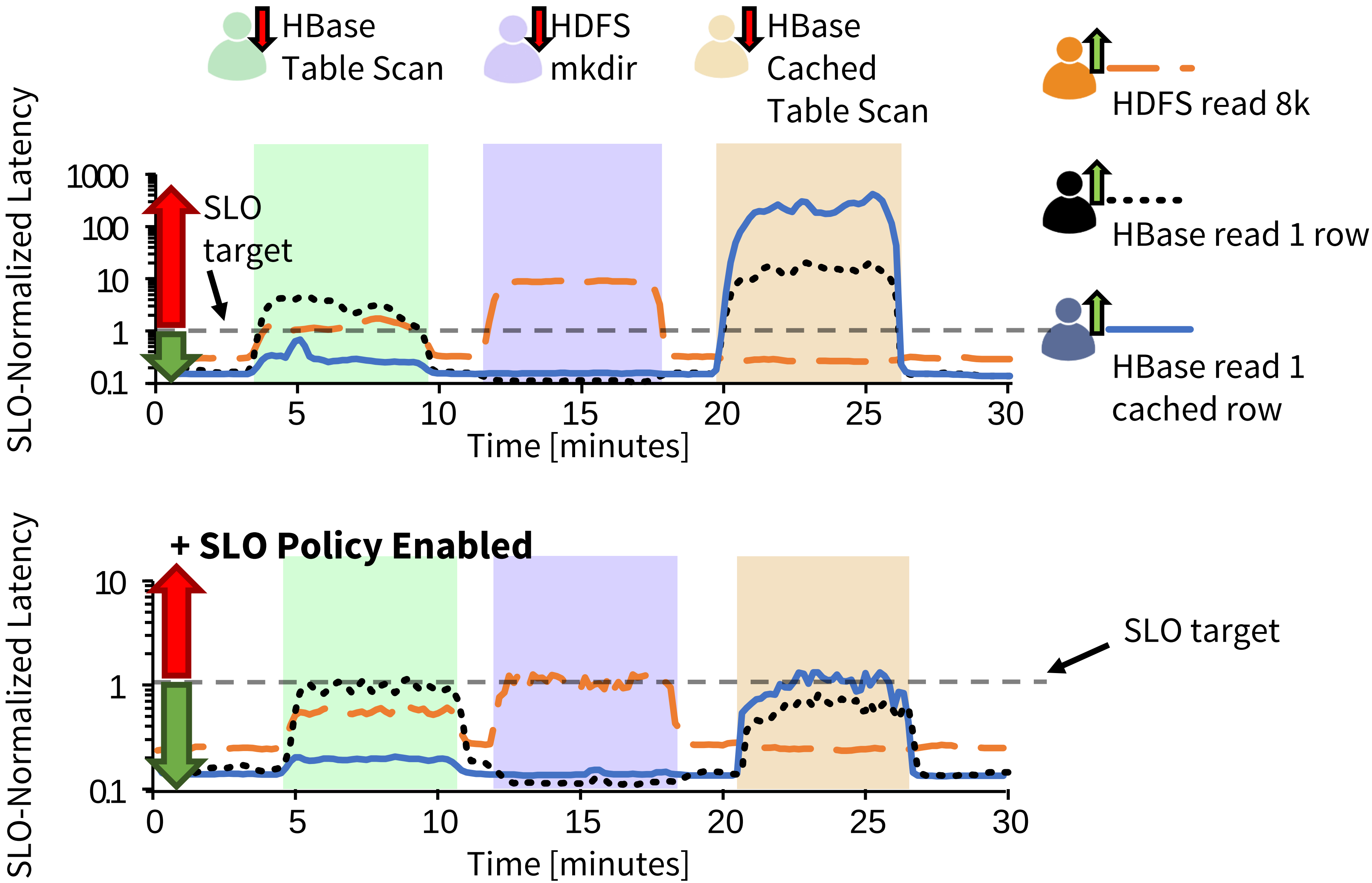


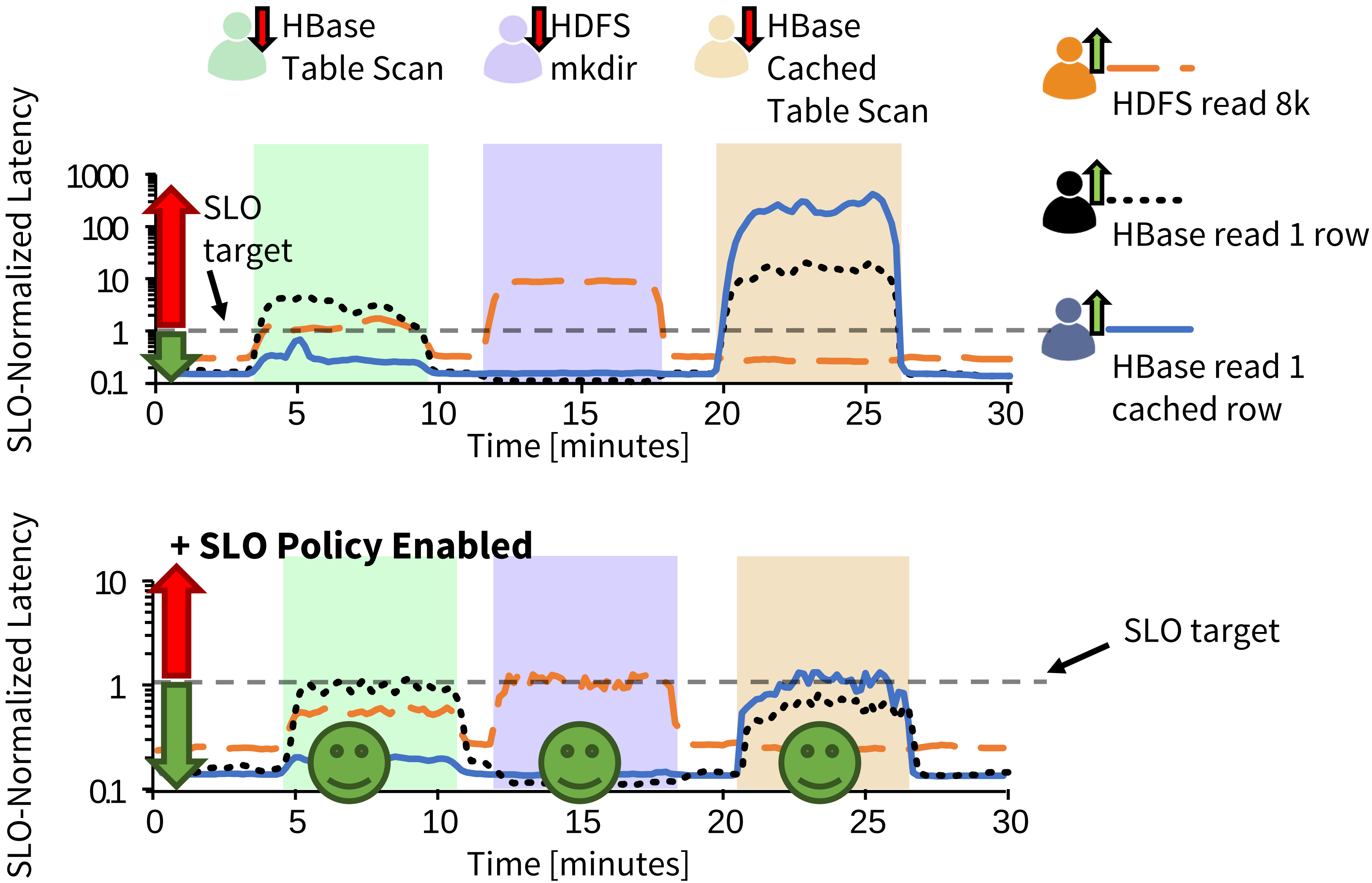
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
Disk
CPU
HDFS NN Lock
Slowdown
15
10
5
0
Time [minutes]

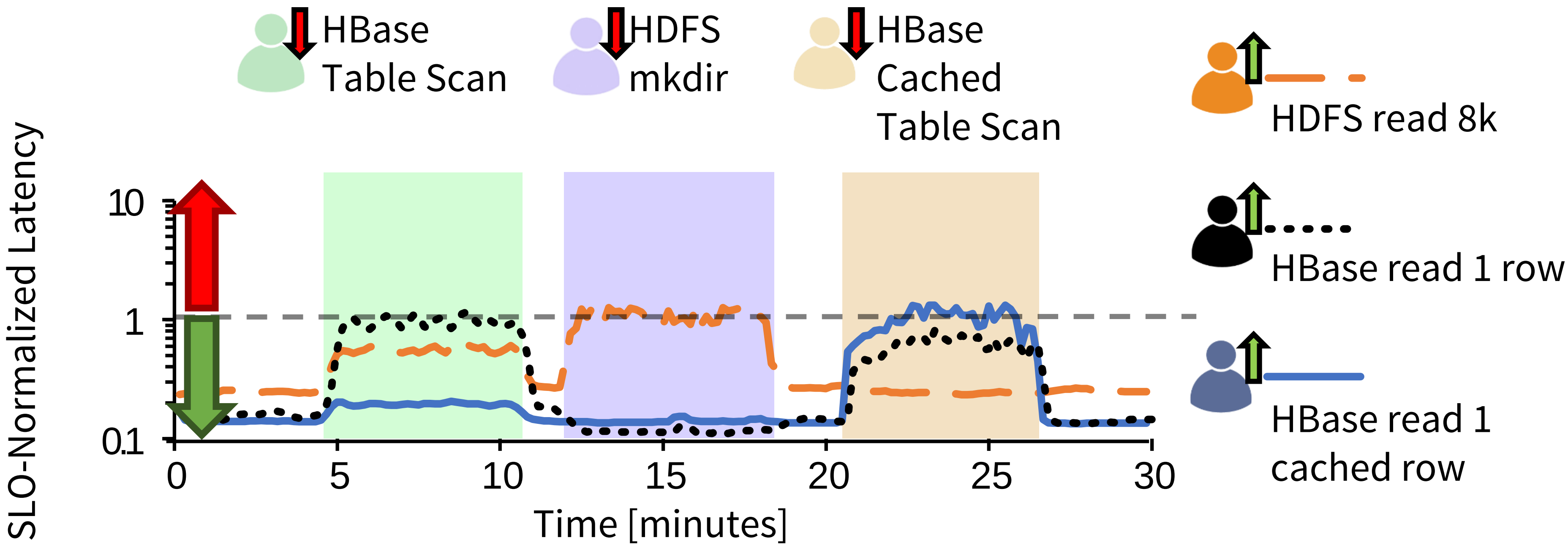

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
Disk
CPU
HDFS NN Lock
HDFS NN Queue
Slowdown
15
10
5
0
0
5
10
15
20
25
30
Time [minutes]



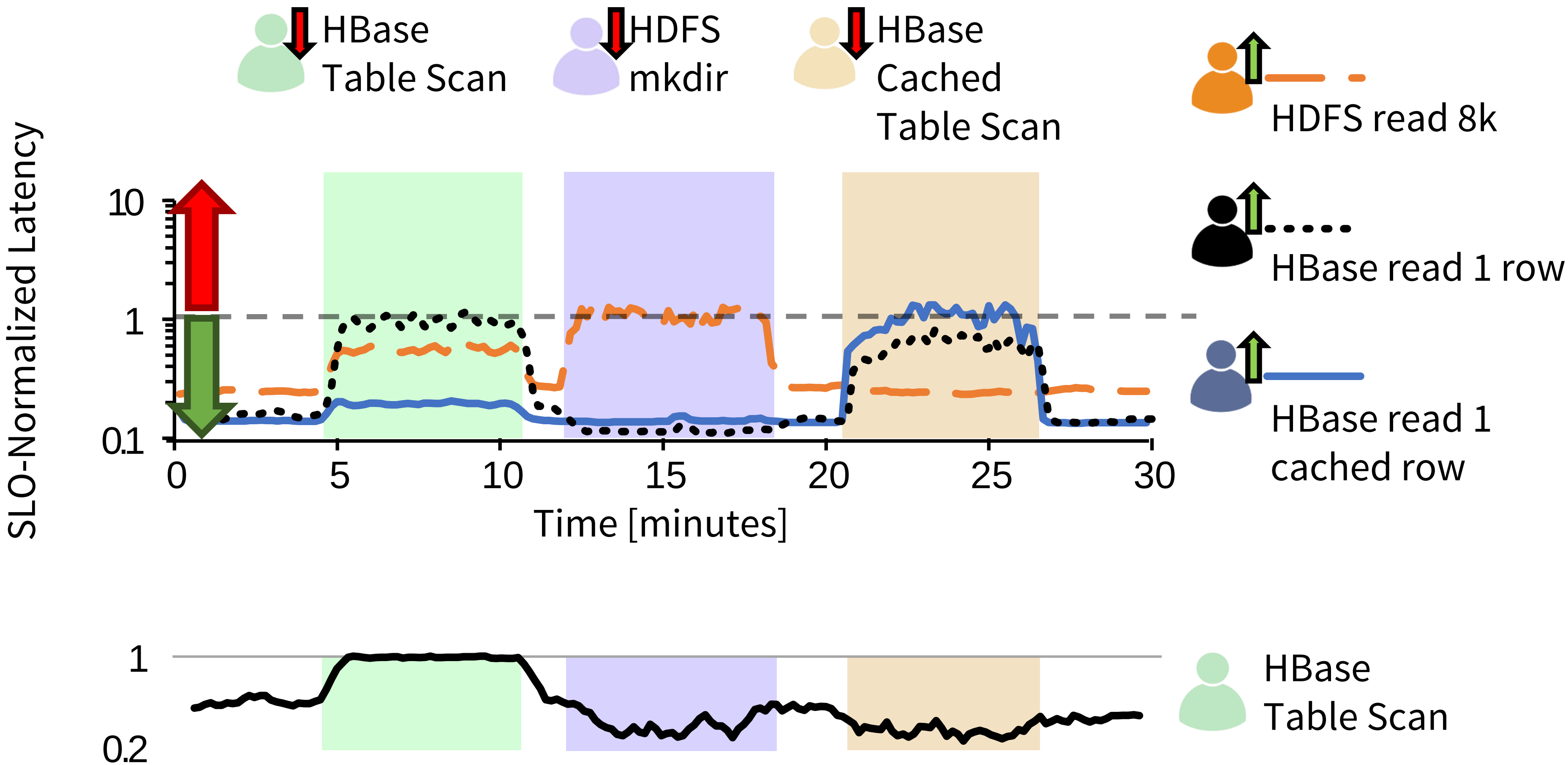
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
Disk
CPU
HDFS NN Lock
HDFS NN Queue
HBase Queue
Slowdown
15
10
5
0
0
5
10
15
20
25
30
Time [minutes]

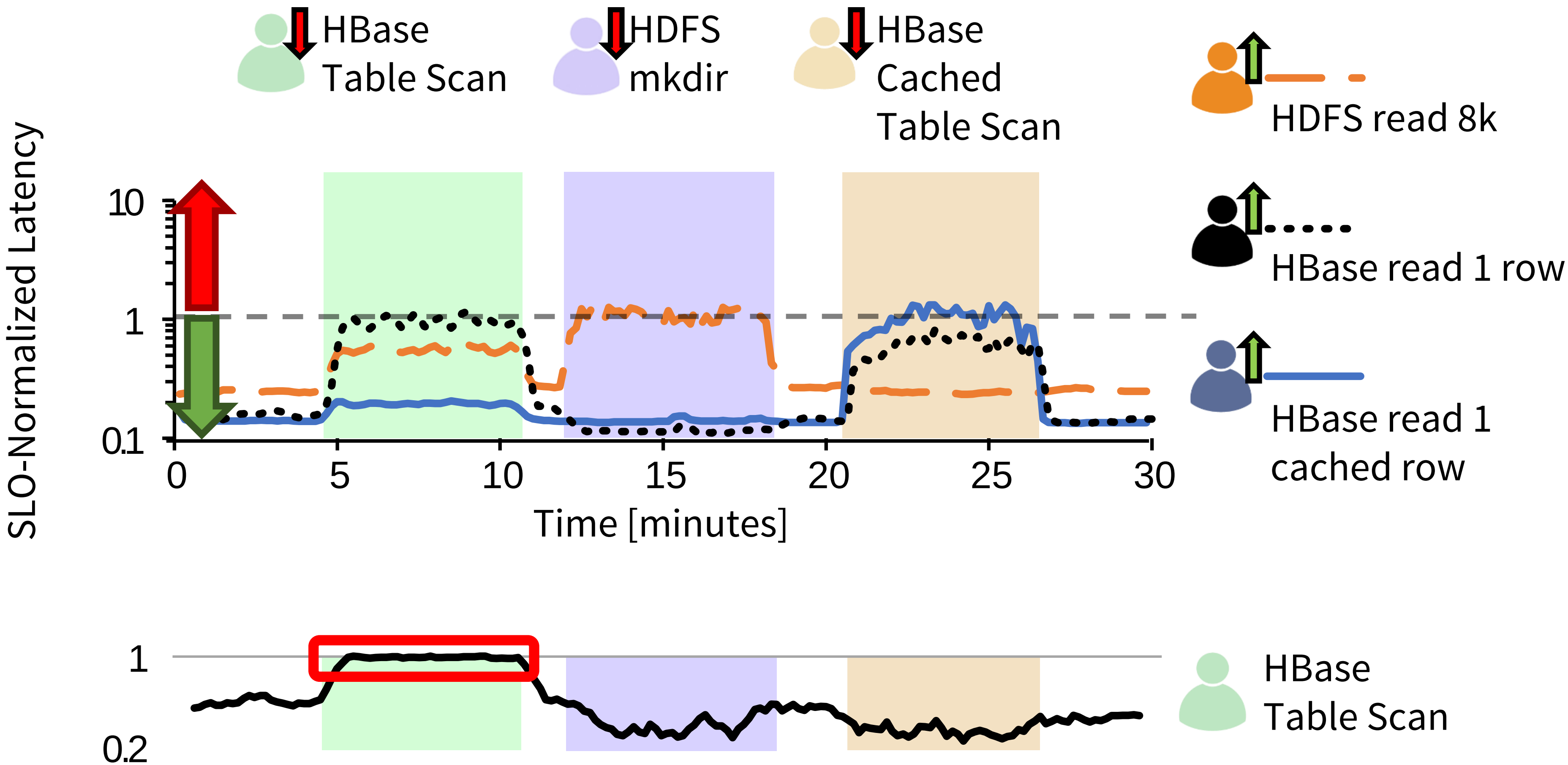

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO target
SLO-Normalized Latency
1000
100
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
+ SLO Policy Enabled
SLO target
SLO-Normalized Latency
10
1
0.1
Time [minutes]

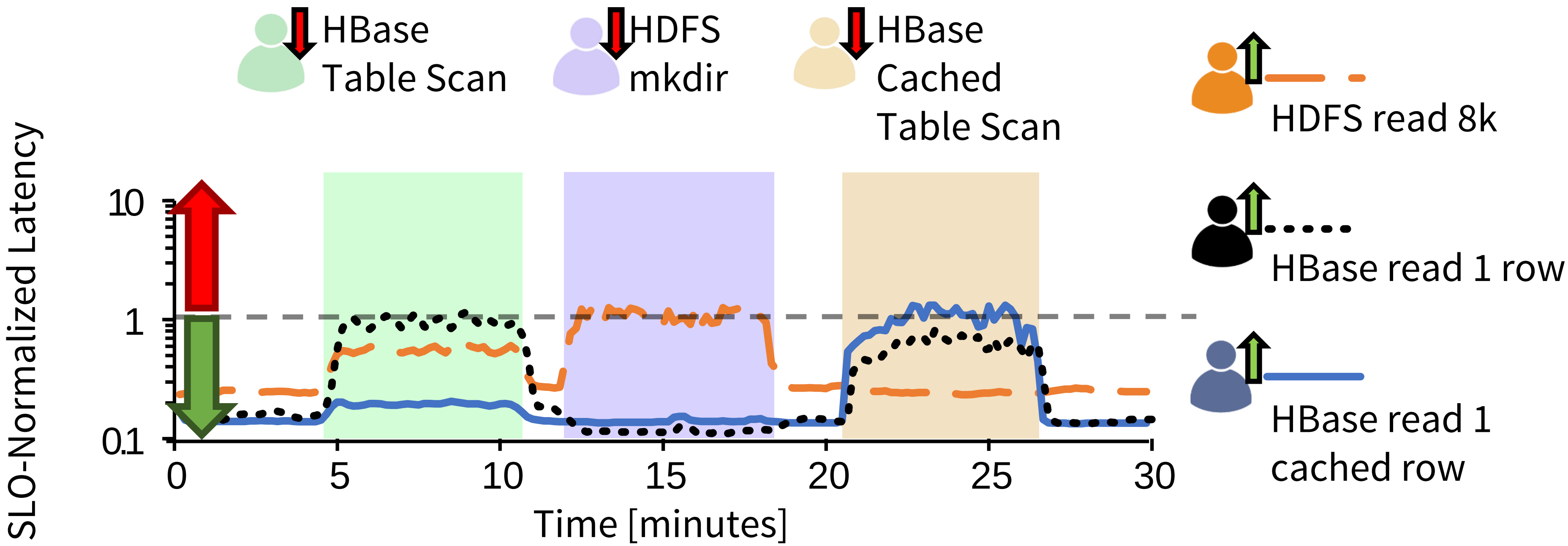

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
1000
100
10
1
0.1
SLO target
0
5
10
15
20
25
30
Time [minutes]
+ SLO Policy Enabled
SLO-Normalized Latency
10
1
0.1
SLO target
0
5
10
15
20
25
30
Time [minutes]



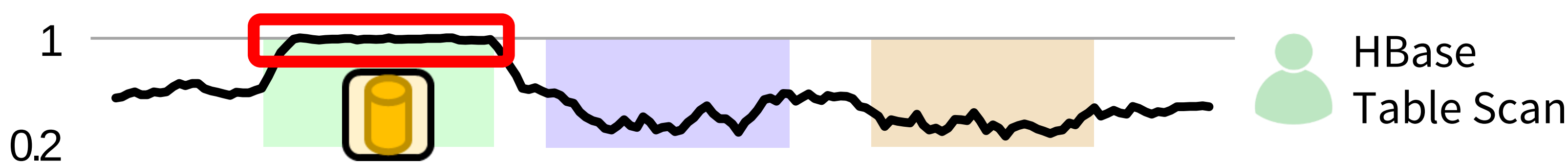
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]


HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
HBase
Table Scan

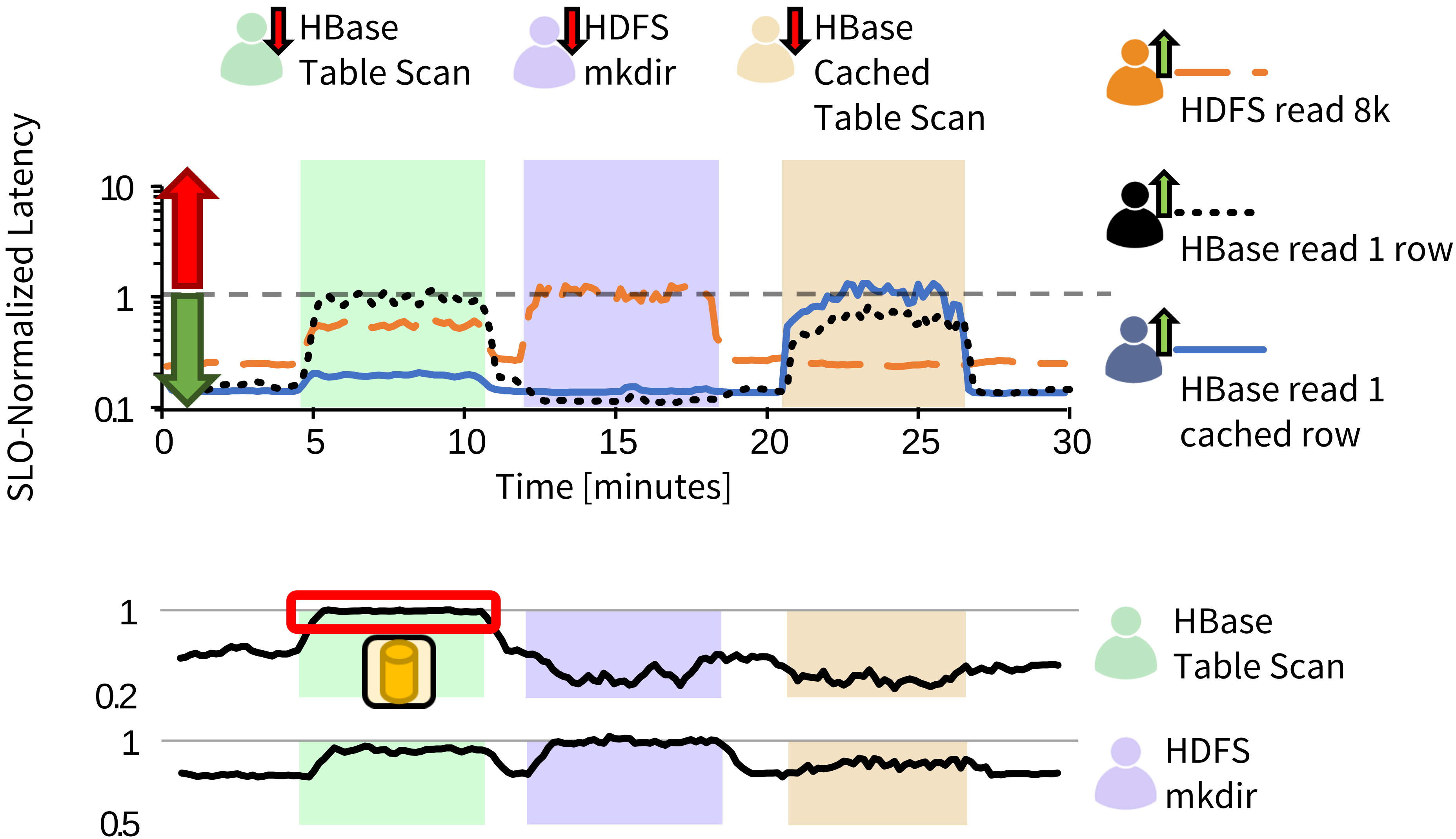

HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
HBase
Table Scan



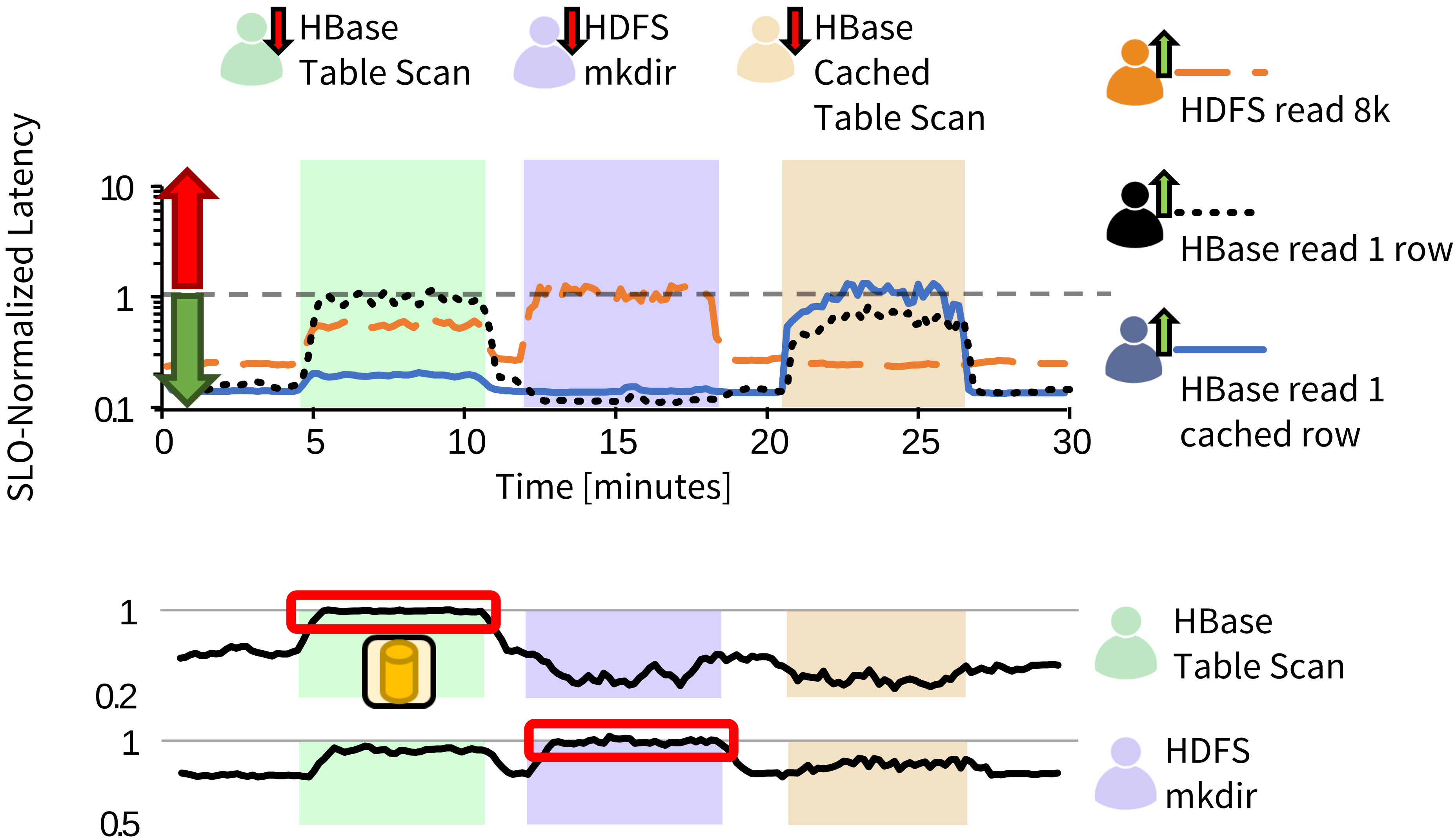
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]

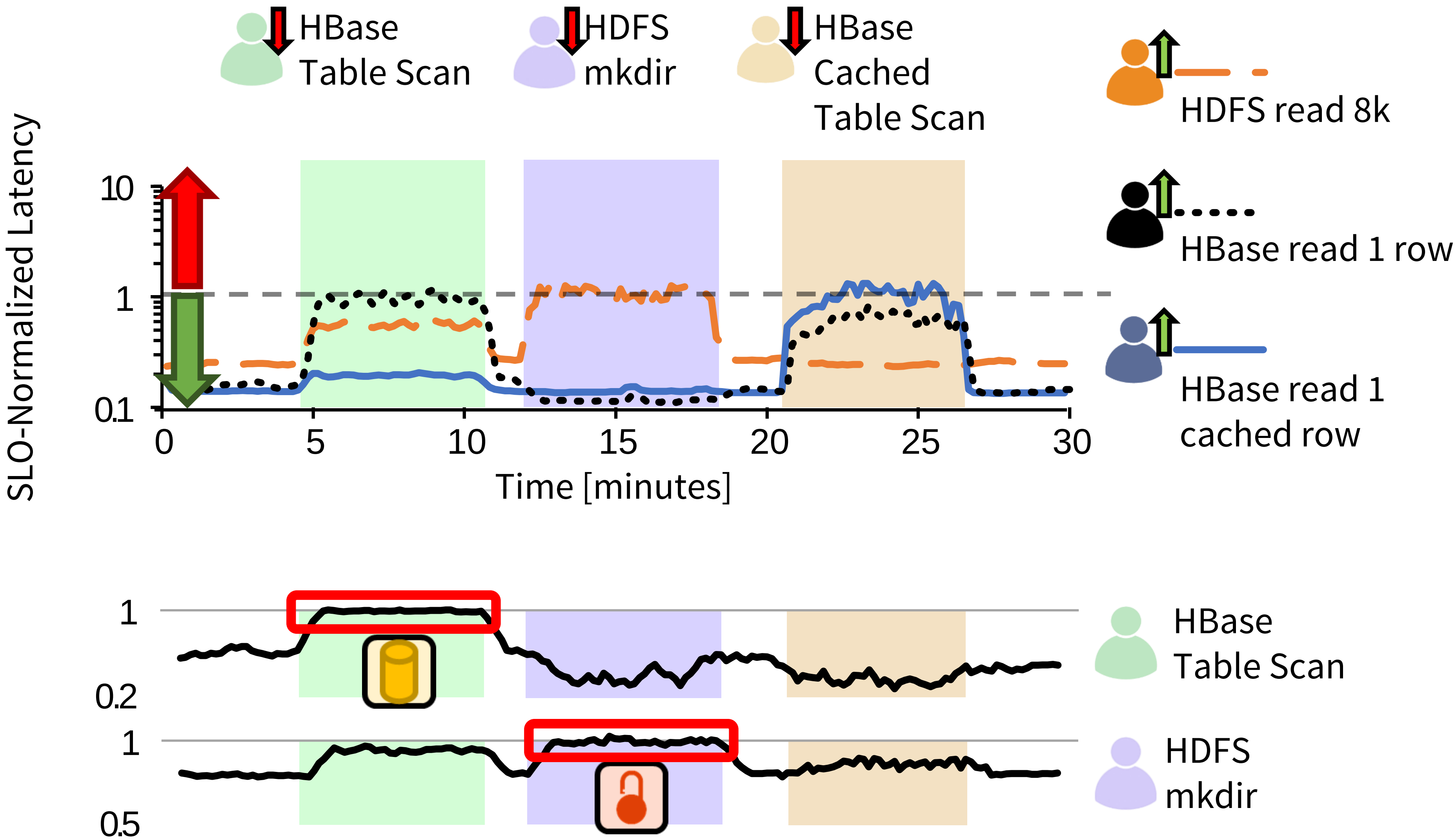


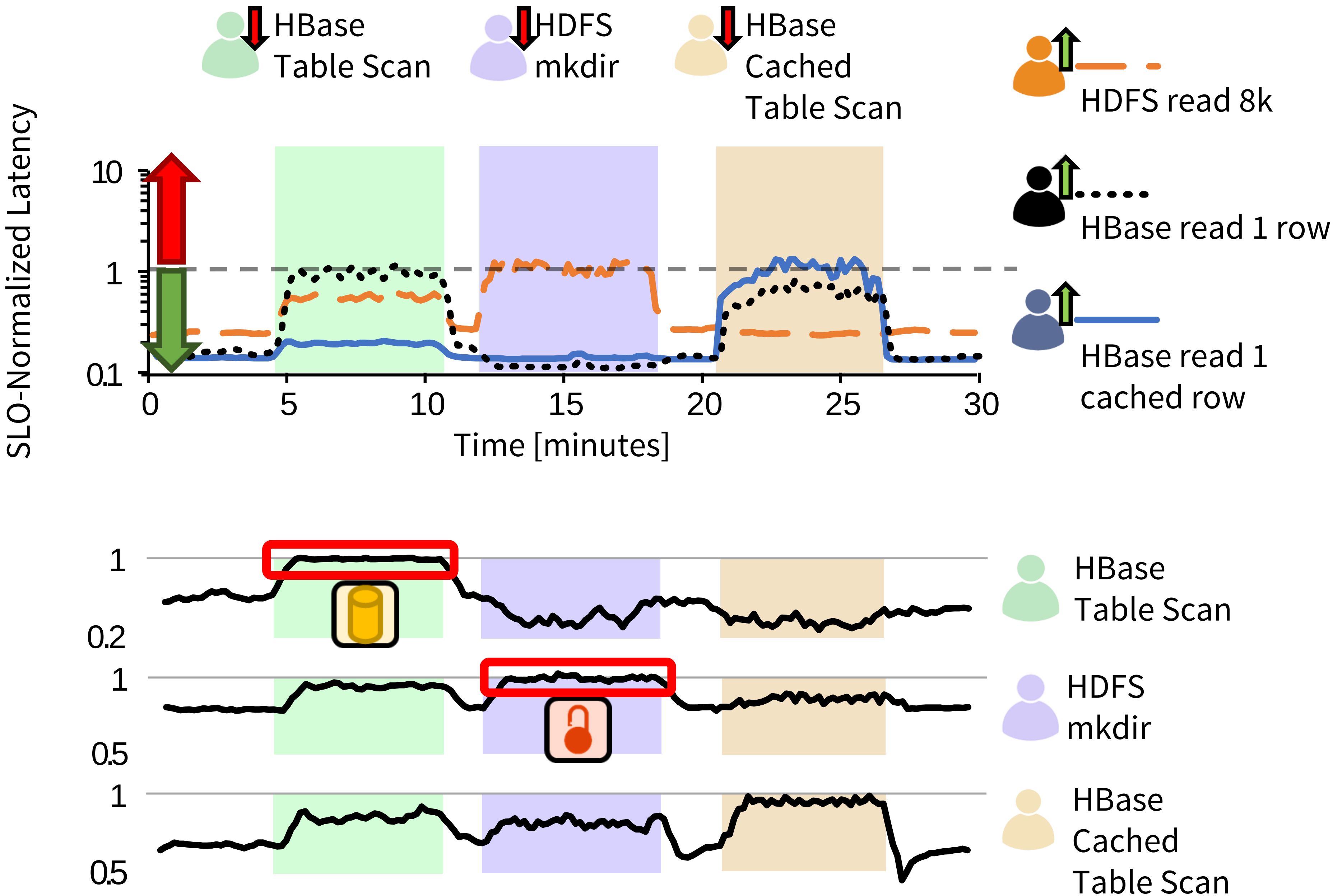
1
0.2
HBase Table Scan



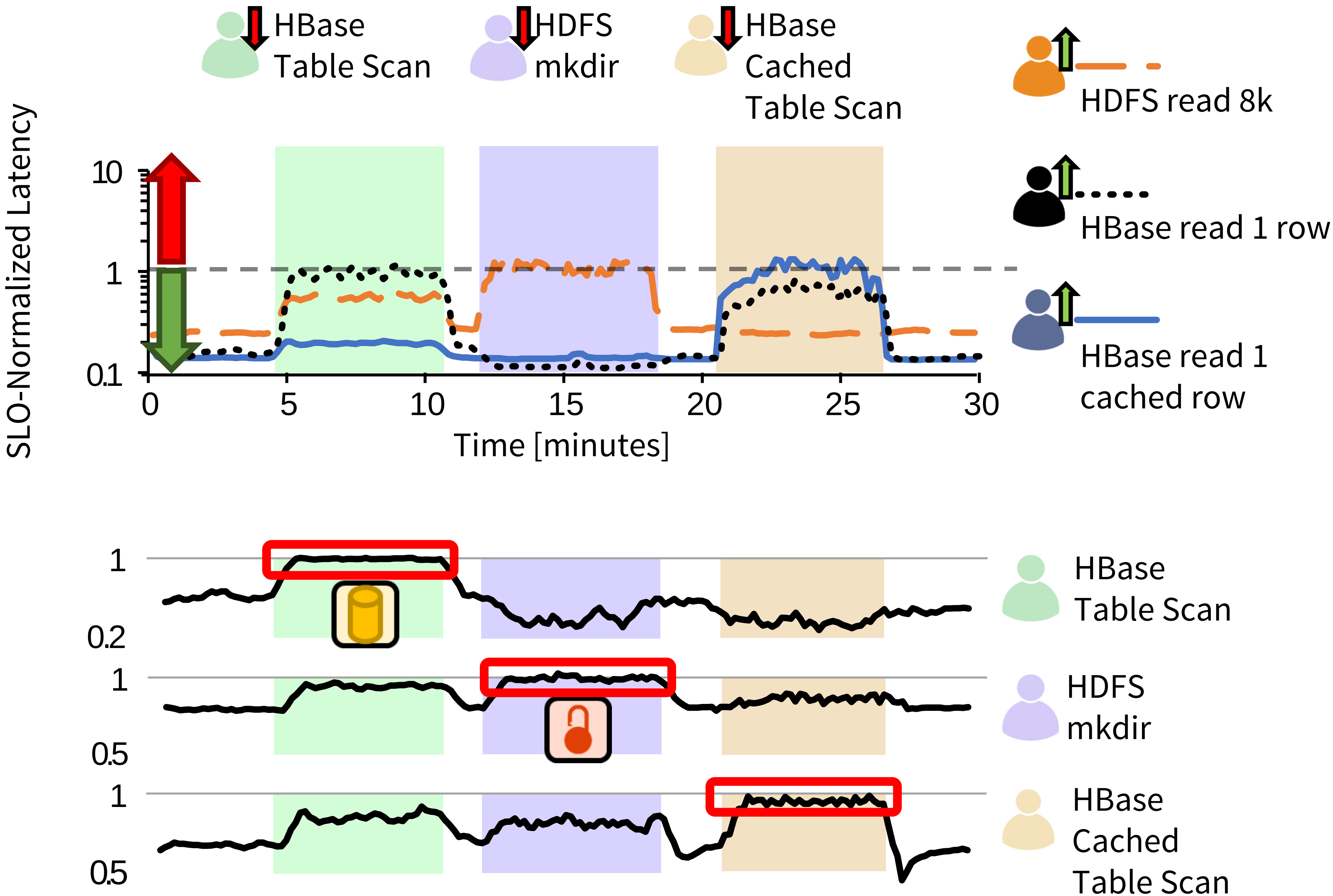
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
HBase Table Scan
1
0.5
HDFS mkdir

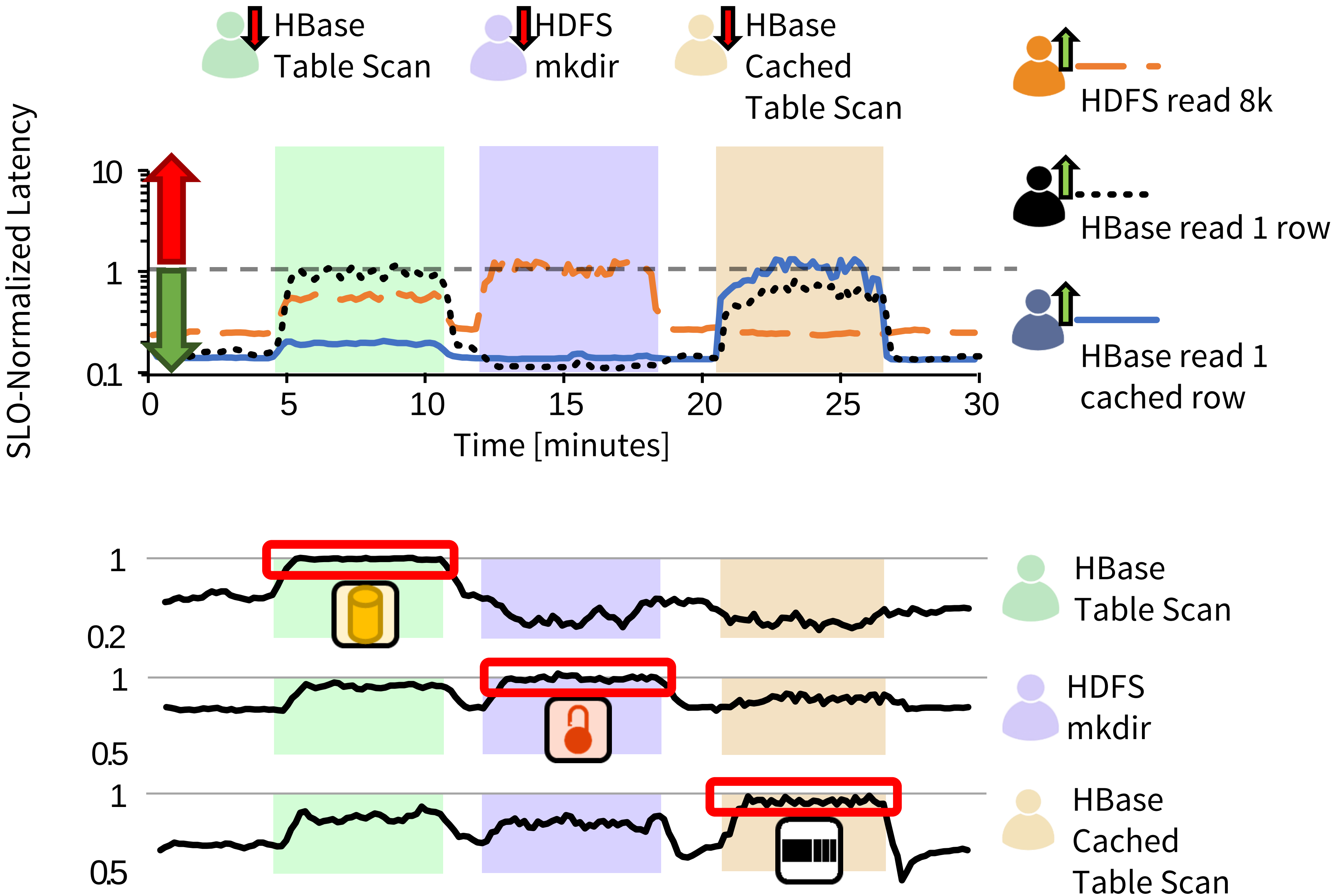

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
1
0.5
HBase Table Scan
HDFS mkdir


HBase
Table Scan
HDFS
mkdir
HBase
Cached
Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1
cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
1
0.5
HBase
Table Scan
HDFS
mkdir



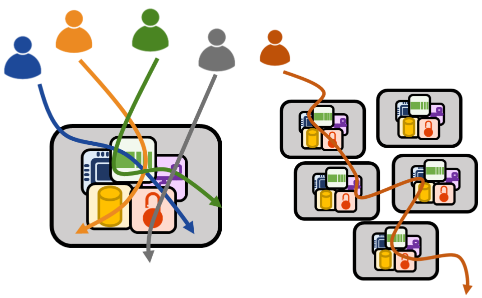
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
1
0.5
1
0.5
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan


HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
1
0.5
1
0.5
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan

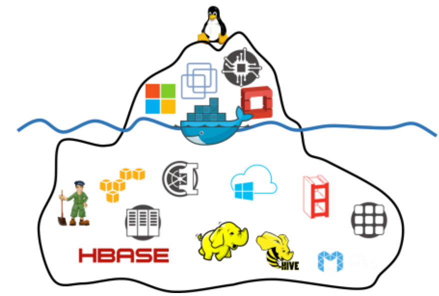

HBase Table Scan
HDFS mkdir
HBase Cached Table Scan
HDFS read 8k
HBase read 1 row
HBase read 1 cached row
SLO-Normalized Latency
10
1
0.1
0
5
10
15
20
25
30
Time [minutes]
1
0.2
1
0.5
1
0.5
HBase Table Scan
HDFS mkdir
HBase Cached Table Scan

Retro Conclusion

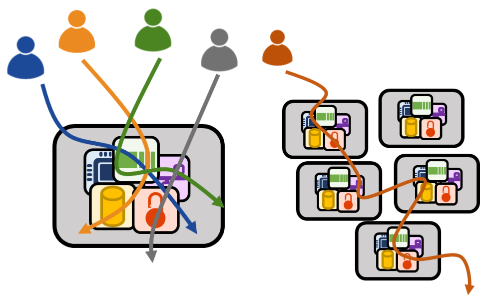
Retro Conclusion



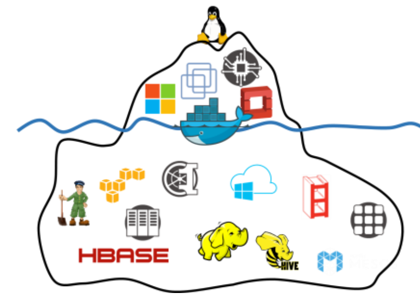
Resource management for *shared* distributed systems



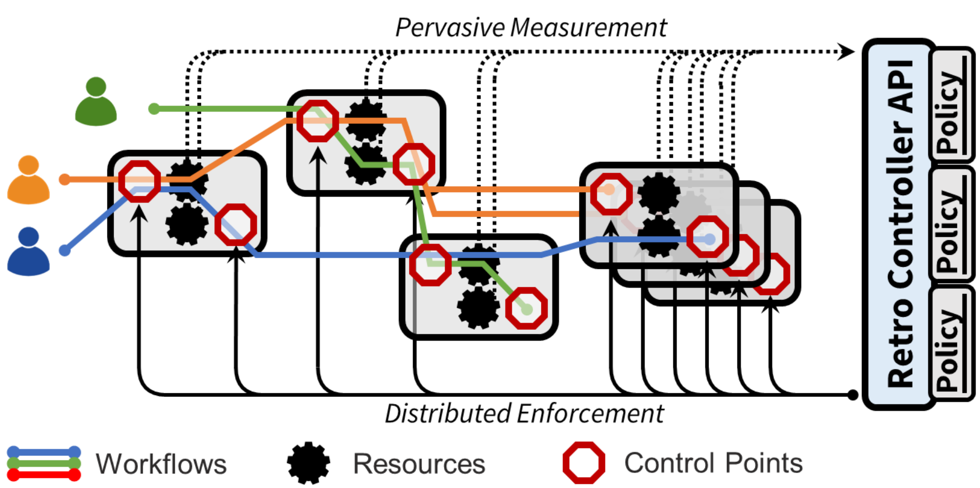
Retro Conclusion



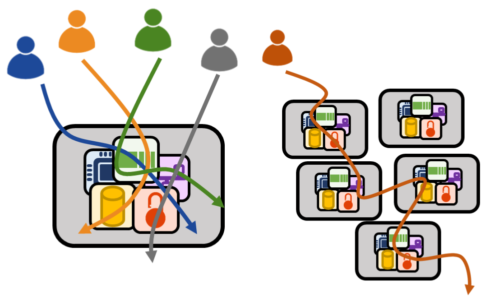
Resource management for *shared* distributed systems



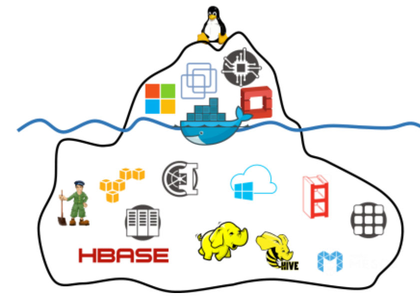
Centralized resource management



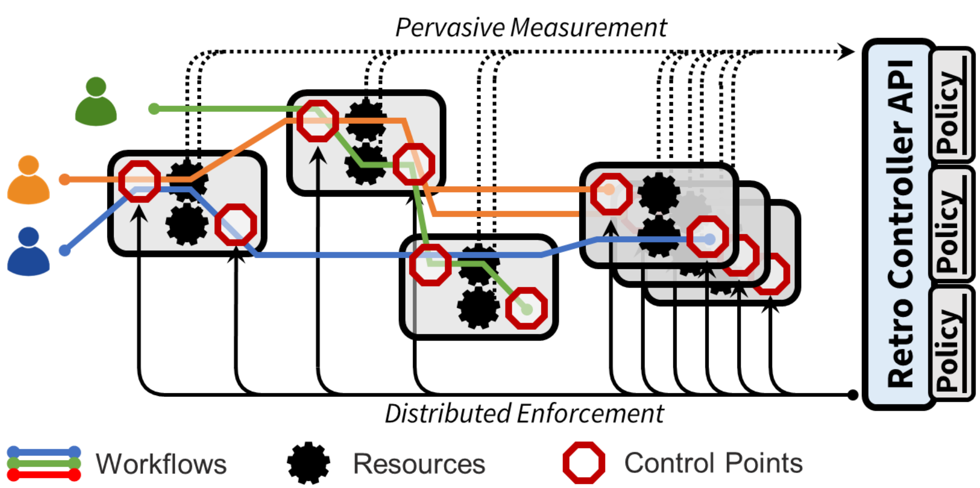
Retro Conclusion



Resource management for *shared* distributed systems

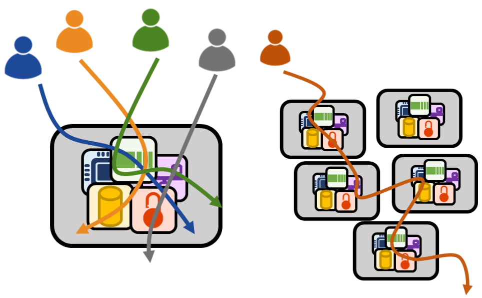


Centralized resource management

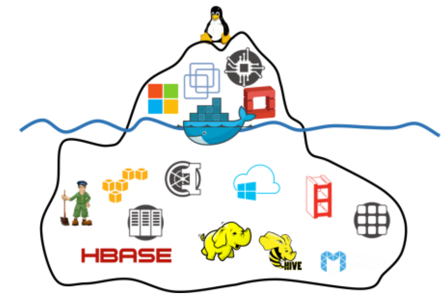


Comprehensive: resources, processes, tenants, background tasks

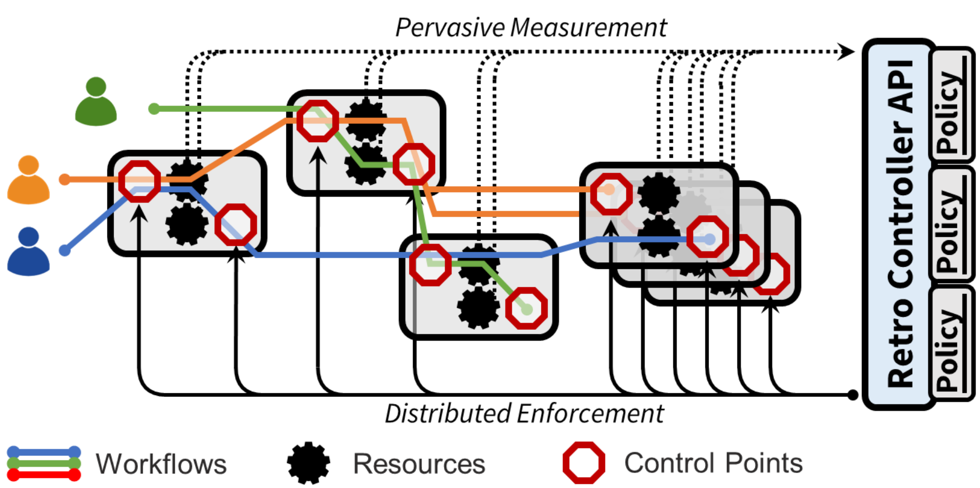
Retro Conclusion



Resource management for *shared* distributed systems



Centralized resource management

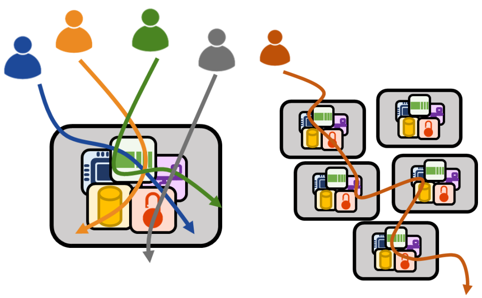


Comprehensive: resources, processes, tenants, background tasks

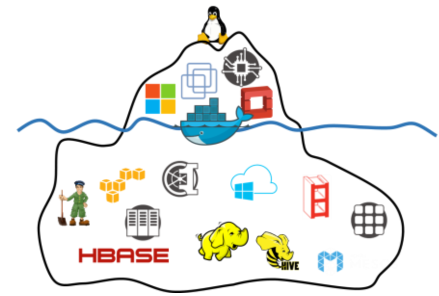
Abstractions for writing *concise*, *general-purpose* policies:

- Workflows
- Resources (slowdown, load)
- Control points

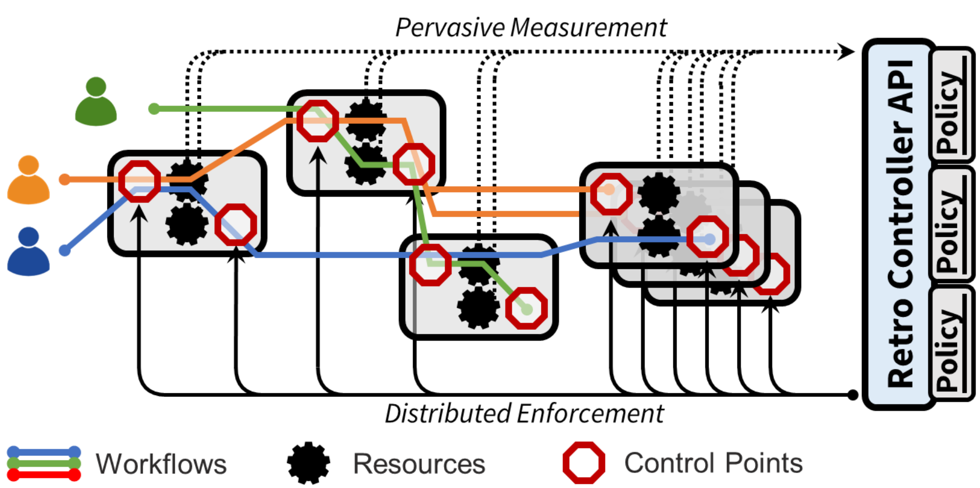
Retro Conclusion



Resource management for *shared* distributed systems



Centralized resource management



Comprehensive: resources, processes, tenants, background tasks

Abstractions for writing *concise, general-purpose* policies:

- Workflows
- Resources (slowdown, load)
- Control points

BROWN UNIVERSITY

http://cs.brown.edu/~jcmace

Microsoft Research