CS 591 K1: Data Stream Processing and Analytics

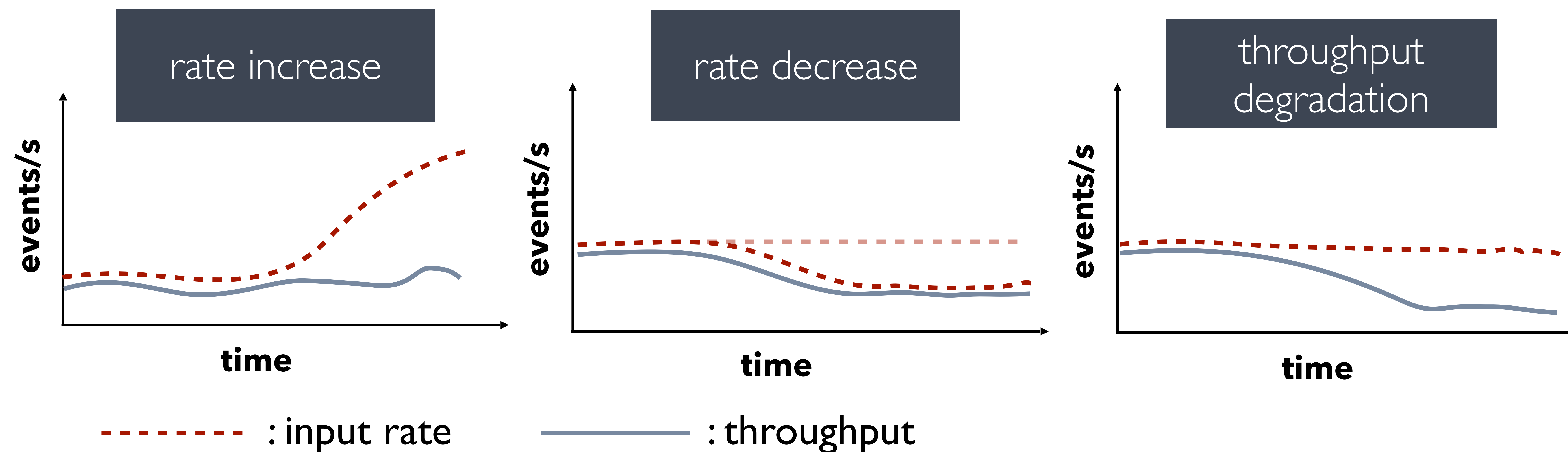
Spring 2020

4/02: Elasticity policies and state migration

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Streaming applications are long-running

- Workload will change
- Conditions might change
- State is accumulated over time



Control: When and how much to adapt?

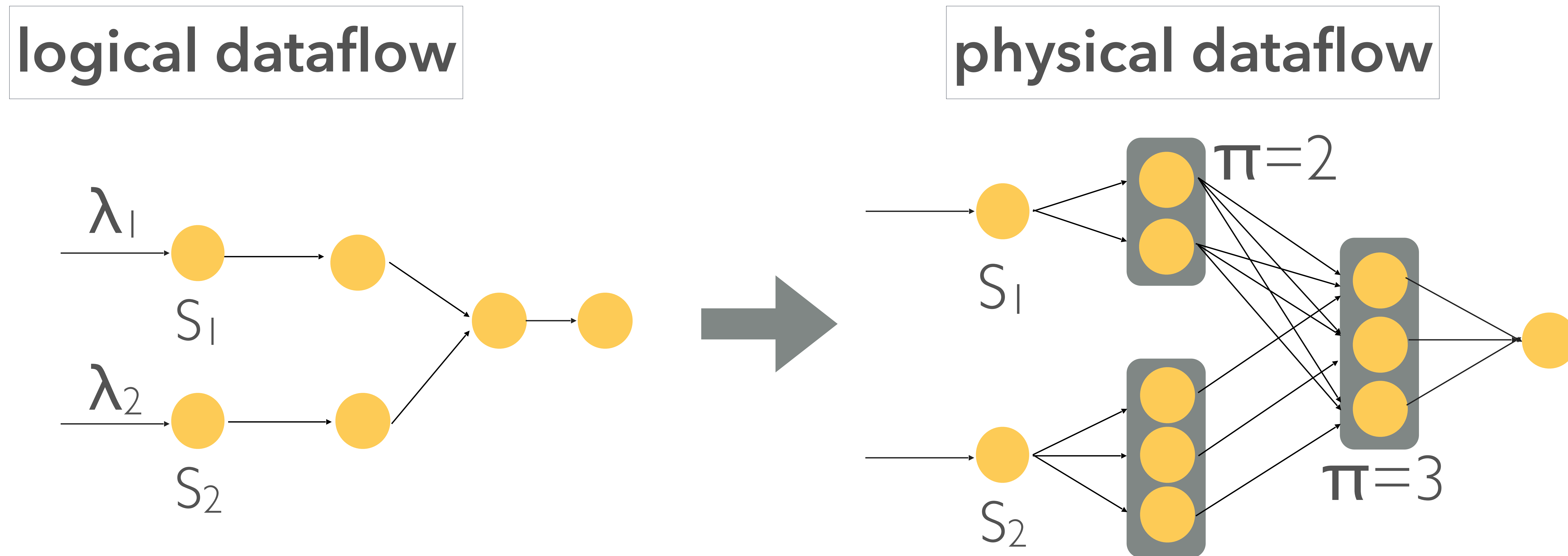
- Detect environment changes: external workload and system performance
- Identify bottleneck operators, straggler workers, skew
- Enumerate scaling actions, predict their effects, and decide which and when to apply

Mechanism: How to apply the re-configuration?

- Allocate new resources, spawn new processes or release unused resources, safely terminate processes
- Adjust dataflow channels and network connections
- Re-partition and migrate state in a consistent manner
- Block and unblock computations to ensure result correctness

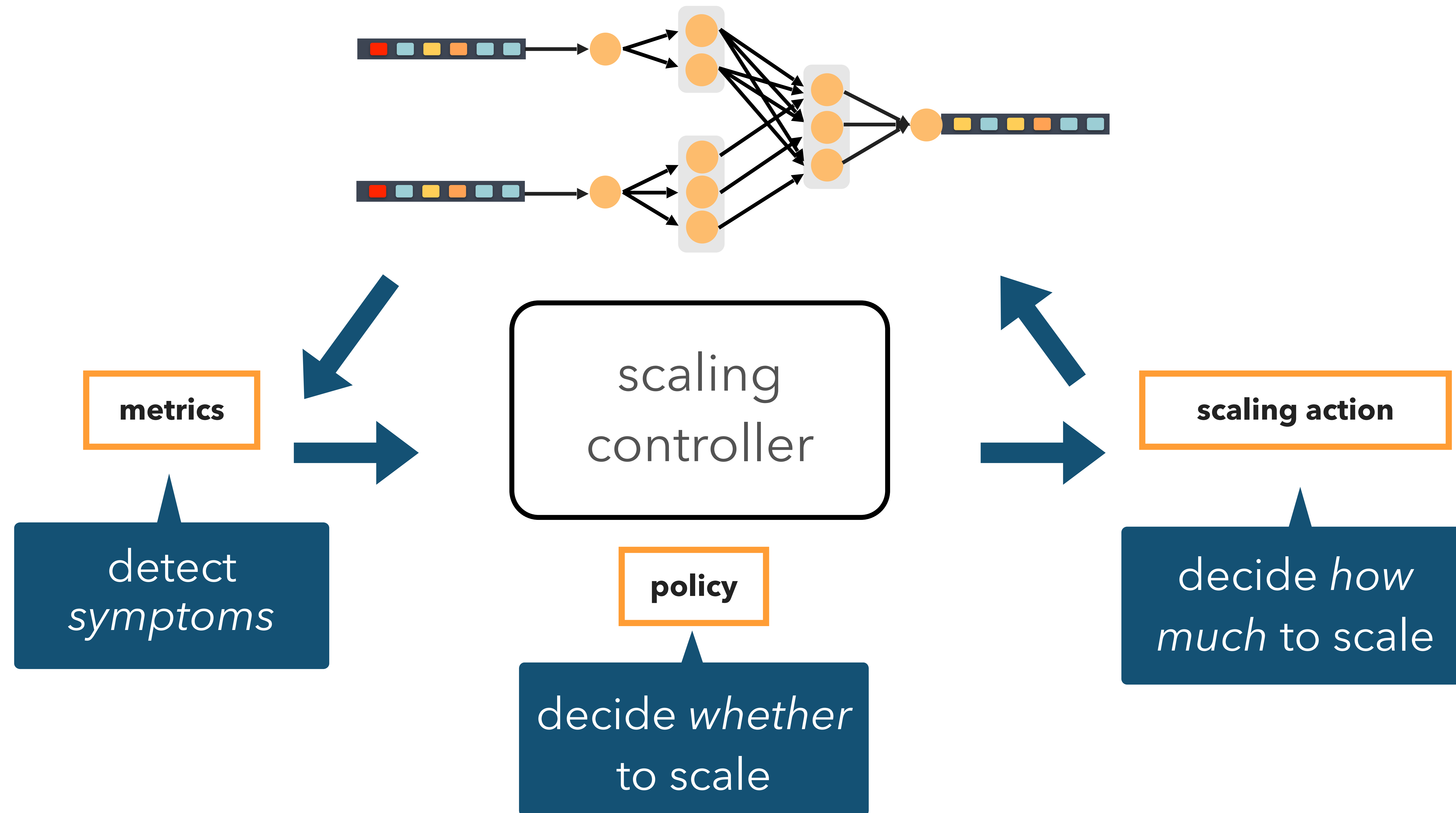
Automatic Scaling Control

The automatic scaling problem



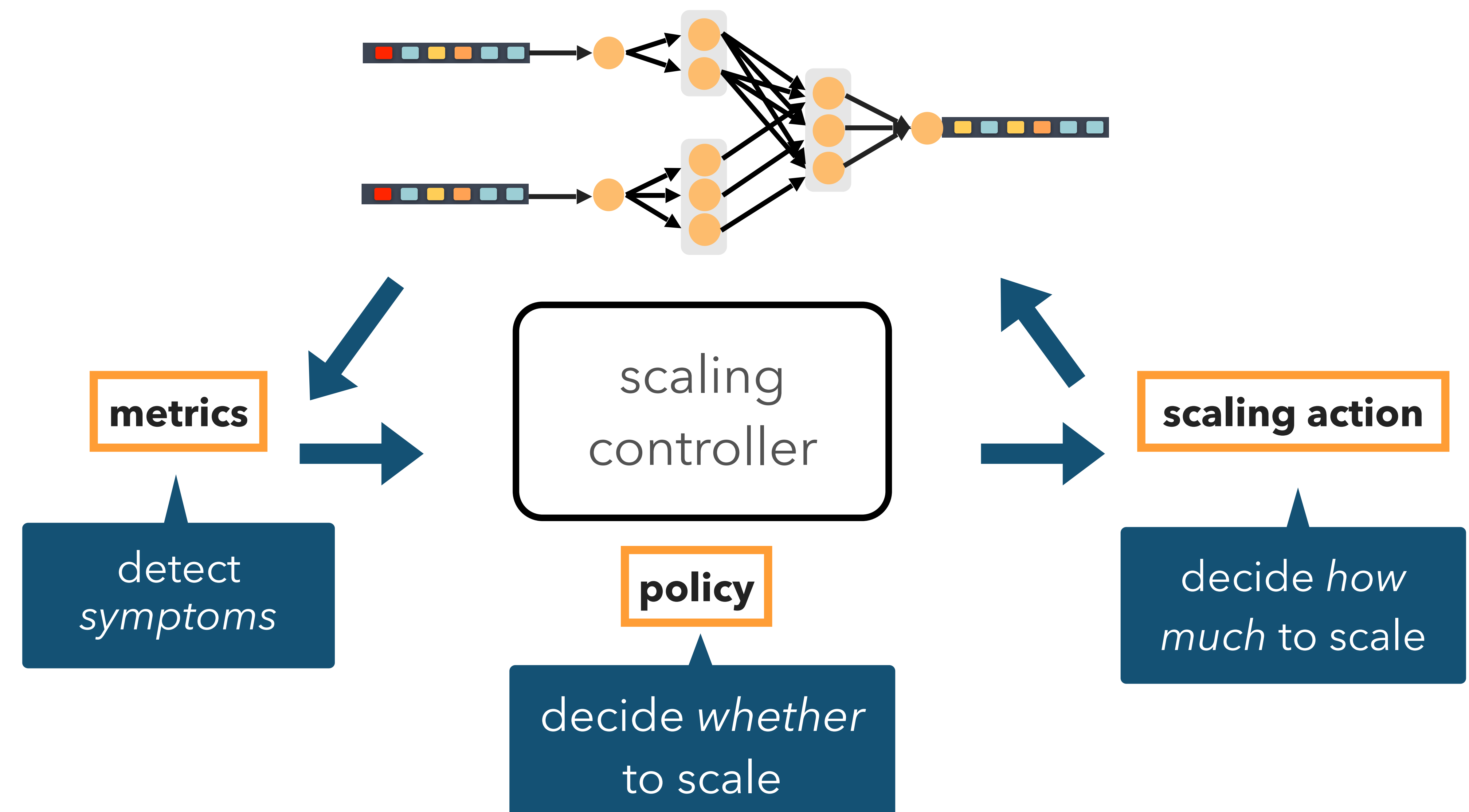
Given a logical dataflow with sources $S_1, S_2, \ldots S_n$ and rates $\lambda_1, \lambda_2, \ldots \lambda_n$ identify the **minimum parallelism** $\boldsymbol{\pi_i}$ per operator i, such that the physical dataflow can sustain all source rates.

Automatic scaling overview



Automatic scaling requirements

- **Accuracy**
  - no over/under-provisioning
- **Stability**
  - no oscillations
- **Performance**
  - fast convergence



Scaling approaches

Metrics

- service time and waiting time per tuple and per task
- total time spent processing a tuple and all its derived results
- CPU utilization, congestion, back pressure, throughput

Policy

- Queuing theory models: for latency objectives
- Control theory models: e.g., PID controller
- Rule-based models, e.g. if CPU utilization > 70% => scale out
- Analytical dataflow-based models

Action

- Speculative: small changes at one operator at a time
- Predictive: at-once for all operators

Queuing theory models

- **Metrics**
  - **service** time and **waiting** time **per tuple** and per task
  - total time spent processing a tuple and all its **derived results**
- **Policy**
  - each operator as a single-server queuing system
  - generalized Jackson networks
- **Action**
  - predictive, at-once for all operators

Queuing theory models

- **Metrics**
  - **service** time and **waiting** time **per tuple** and per task
  - total time spent processing a tuple and all its **derived results**
- **Policy**
  - each operator as a single-server queuing system
  - generalized Jackson networks
- **Action**
  - predictive, at-once for all operators

Too fine-grained, **impractical** for high-rate streams

Sampling **degrades** accuracy

Queuing theory models

- **Metrics**
  - **service** time and **waiting** time **per tuple** and per task
  - total time spent processing a tuple and all its **derived results**
- **Policy**
  - each operator as a single-server queuing system
  - generalized Jackson networks
- **Action**
  - predictive, at-once for all operators

Too fine-grained, **impractical** for high-rate streams

Sampling **degrades** accuracy

Simplified models make **strong assumptions**

Unsuitable for complex operators, e.g. sliding windows, joins

Control theory models

- **Metrics**
  - input and output signals
  - delay of tuples that have just entered the system
- **Policy**
  - dataflow as a black-box
  - SISO models - MIMO too complex
- **Action**
  - predictive, dataflow-wide

Control theory models

- **Metrics**
  - input and output signals
  - delay of tuples that have just entered the system
- **Policy**
  - dataflow as a black-box
  - SISO models - MIMO too complex
- **Action**
  - predictive, dataflow-wide

The output signal *is* the **delay** time

Control theory models

- **Metrics**
  - input and output signals
  - delay of tuples that have just entered the system
- **Policy**
  - dataflow as a black-box
  - SISO models - MIMO too complex
- **Action**
  - predictive, dataflow-wide

The output signal *is* the **delay** time

Performance depends on **parameter selection**, e.g. poles placement, sampling period, damping

Cannot identify individual **bottlenecks** neither model 2-input operators

Heuristic models

- **Metrics**
  - externally observed coarse-grained and aggregates
  - CPU utilization, throughput, back-pressure signal
- **Policy**
  - rule-based
  - *If CPU utilization > 70% and back-pressure then scale up*
- **Action**
  - speculative, one operator at-a-time

Heuristic models

- **Metrics**
  - externally observed coarse-grained and aggregates
  - CPU utilization, throughput, back-pressure signal
- **Policy**
  - rule-based
  - *If CPU utilization > 70% and back-pressure then scale up*
- **Action**
  - speculative, one operator at-a-time

Noisy, sensitive to interference, misleading

Easy-to-obtain

Heuristic models

- **Metrics**
  - externally observed coarse-grained and aggregates
  - CPU utilization, throughput, back-pressure signal
- **Policy**
  - rule-based
  - *If CPU utilization > 70% and back-pressure then scale up*
- **Action**
  - speculative, one operator at-a-time

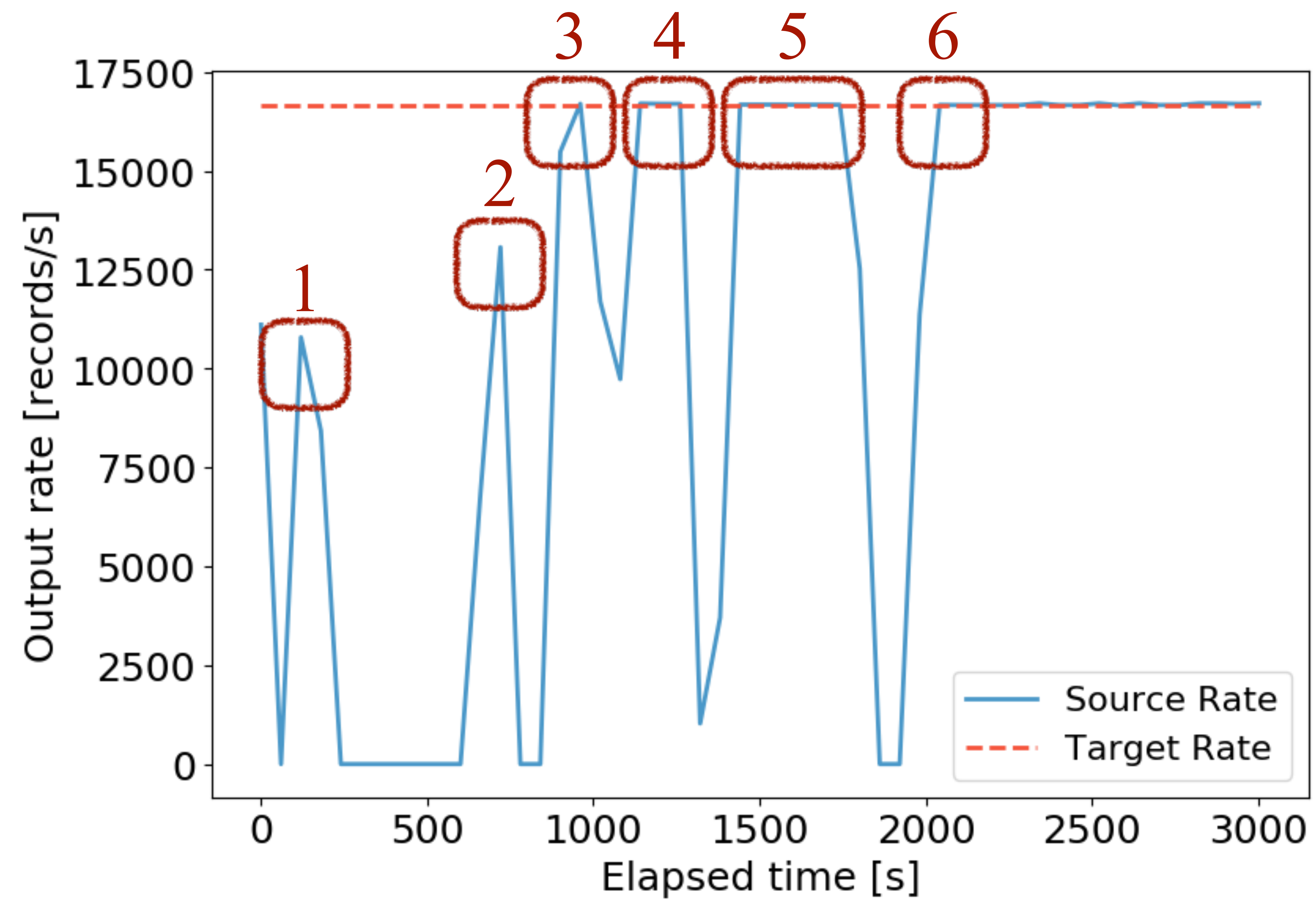
Noisy, sensitive to interference, misleading

Easy-to-obtain

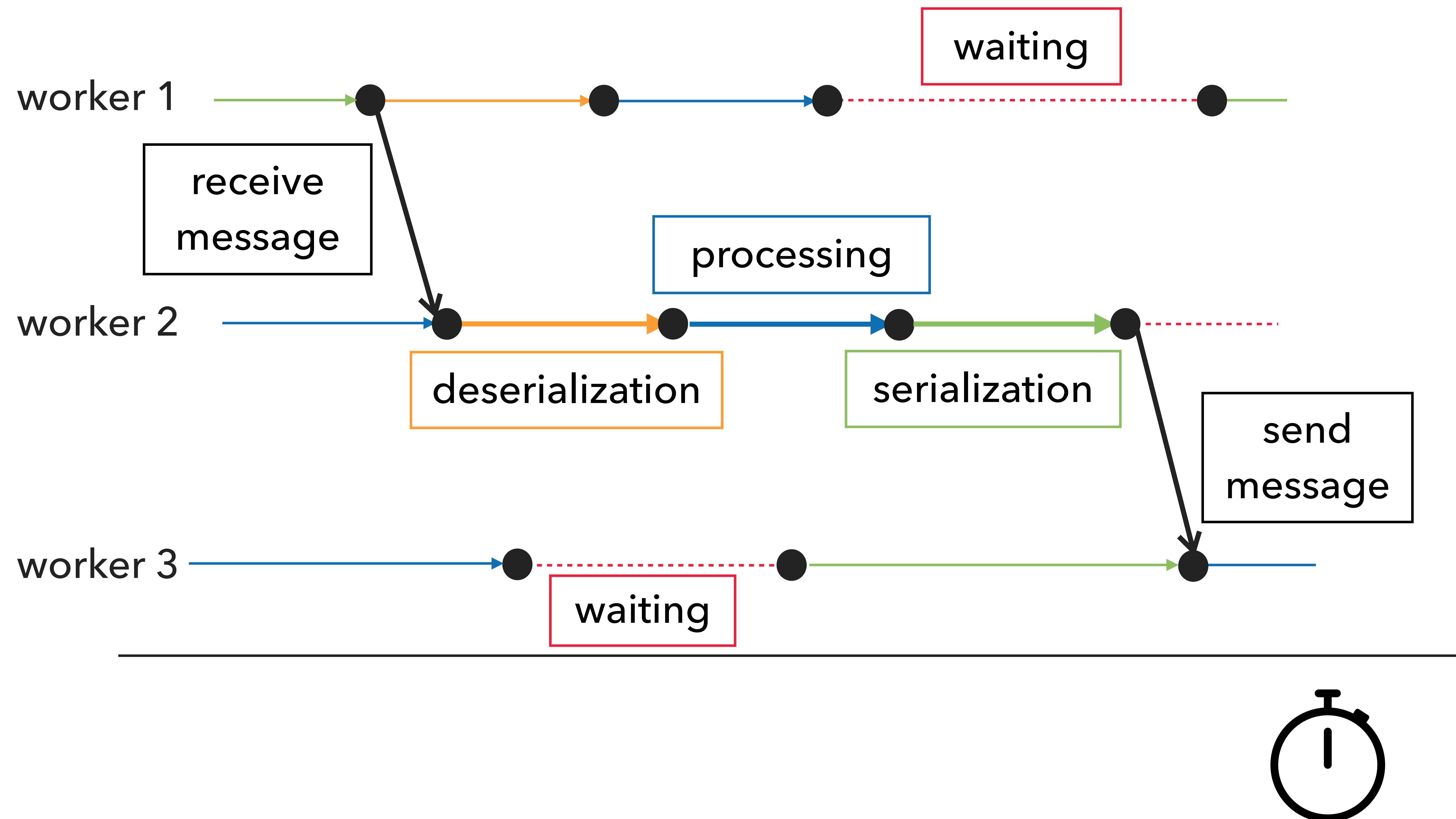
Sensitive to thresholds and require **manual tuning**

Oscillations, slow convergence, black-listing

effect of Dhalion's scaling actions
in an initially under-provisioned wordcount dataflow



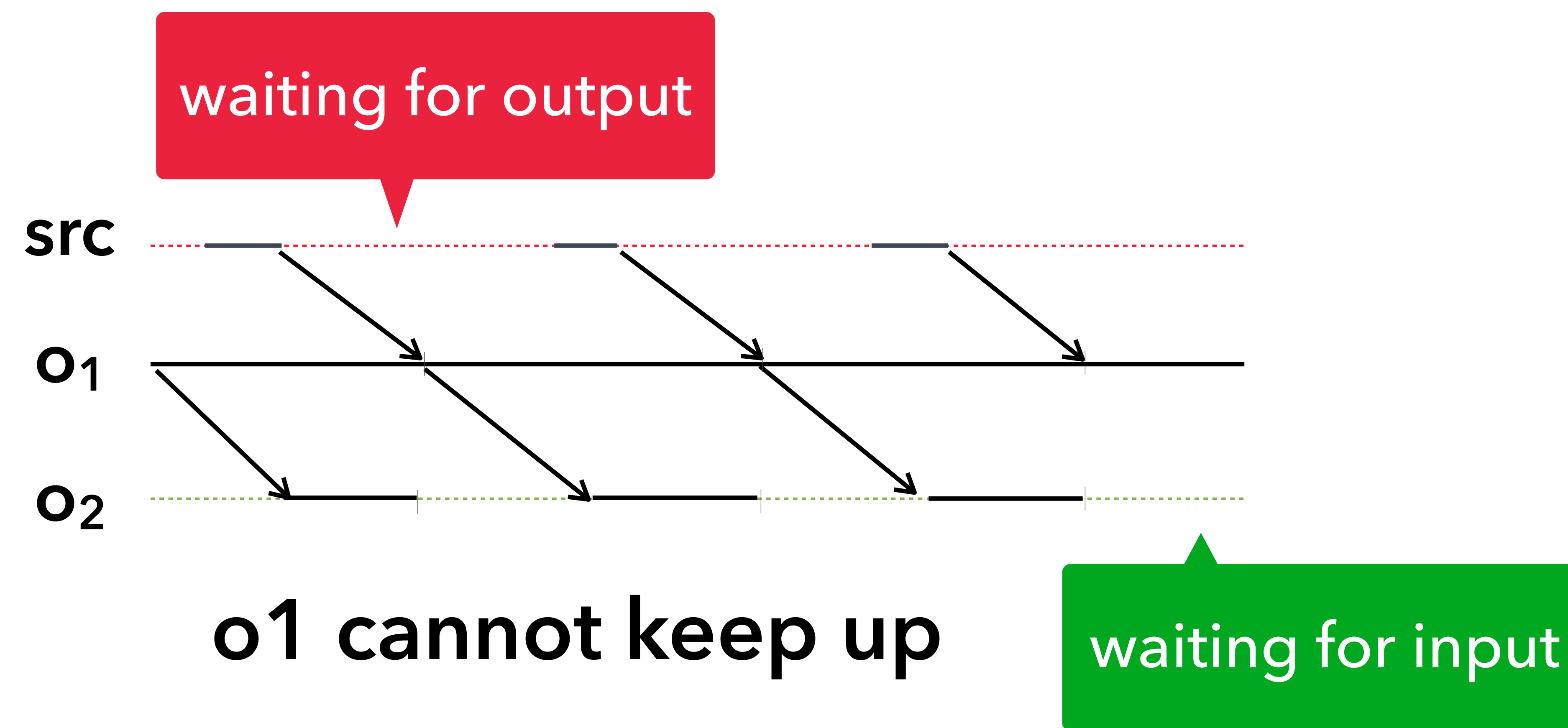
Dataflow worker activities



Which operator is the bottleneck?

What if we scale o_1 x 4?

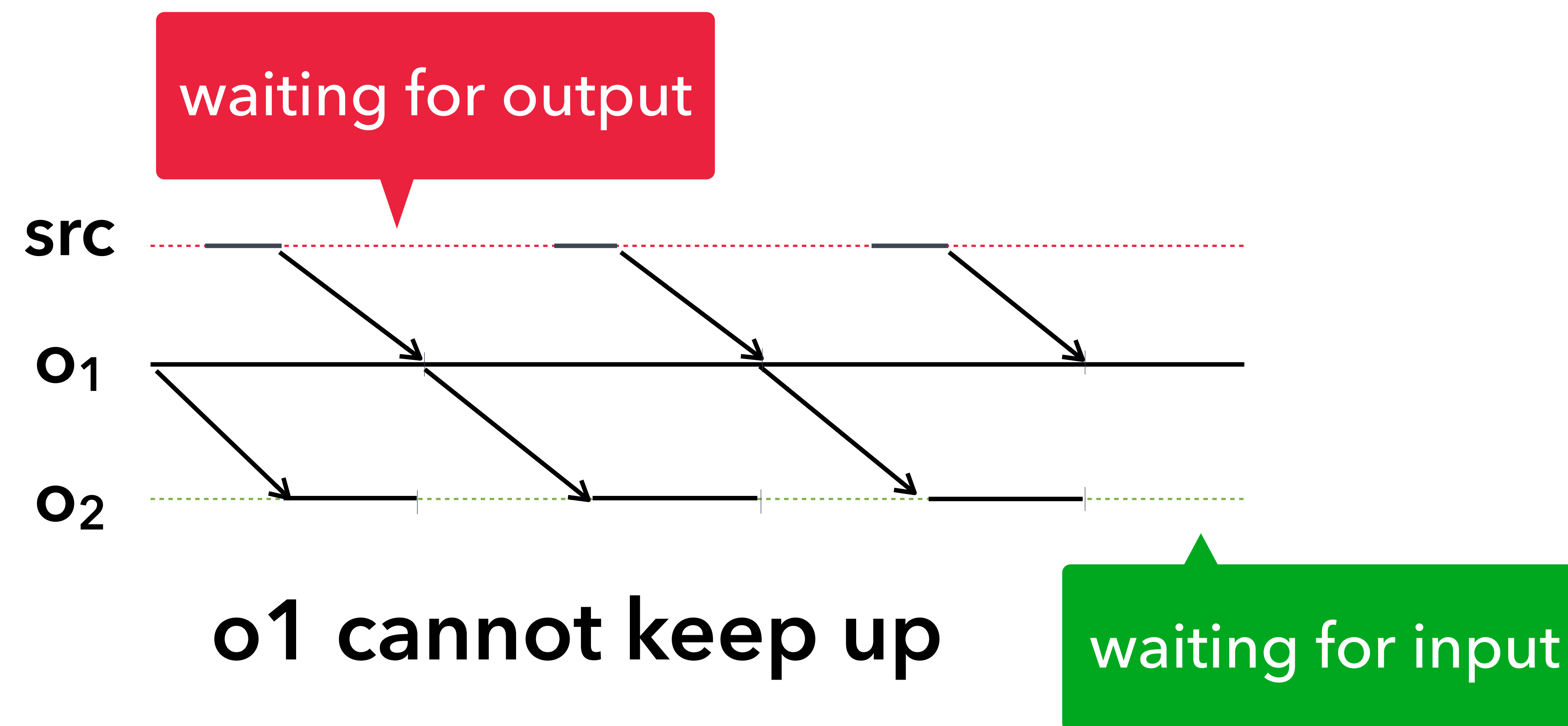
How much to scale o_2?



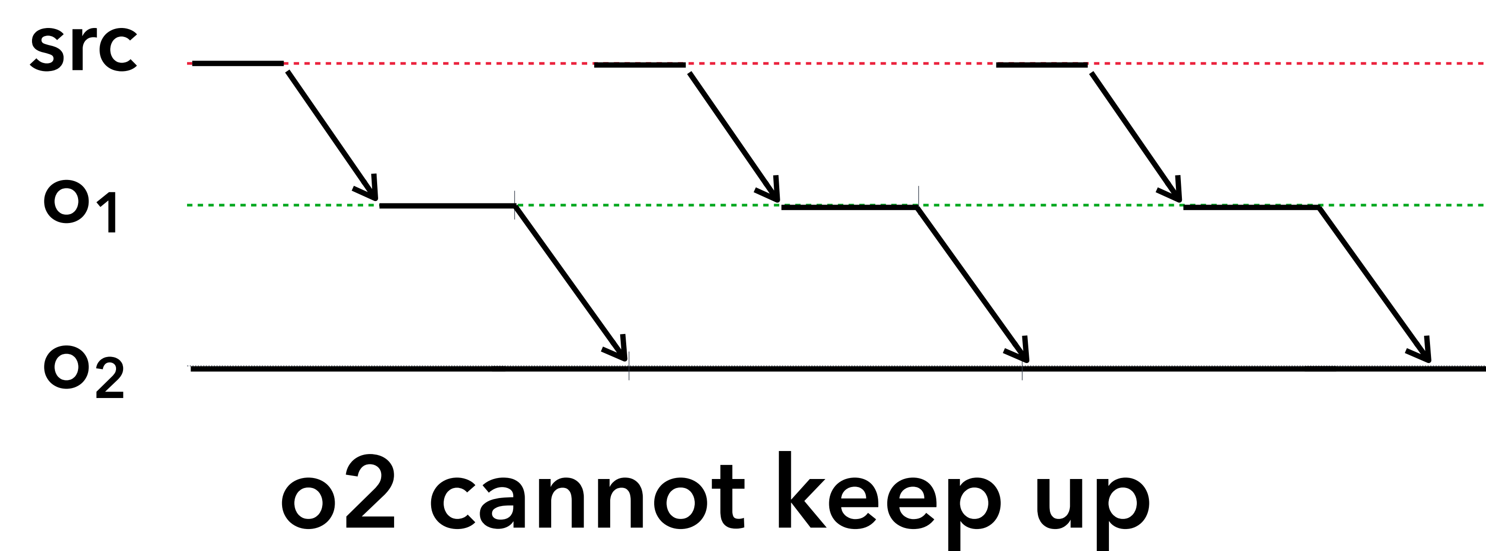
Which operator is the bottleneck?

What if we scale o_1 x 4?

How much to scale o_2?



o1 cannot keep up



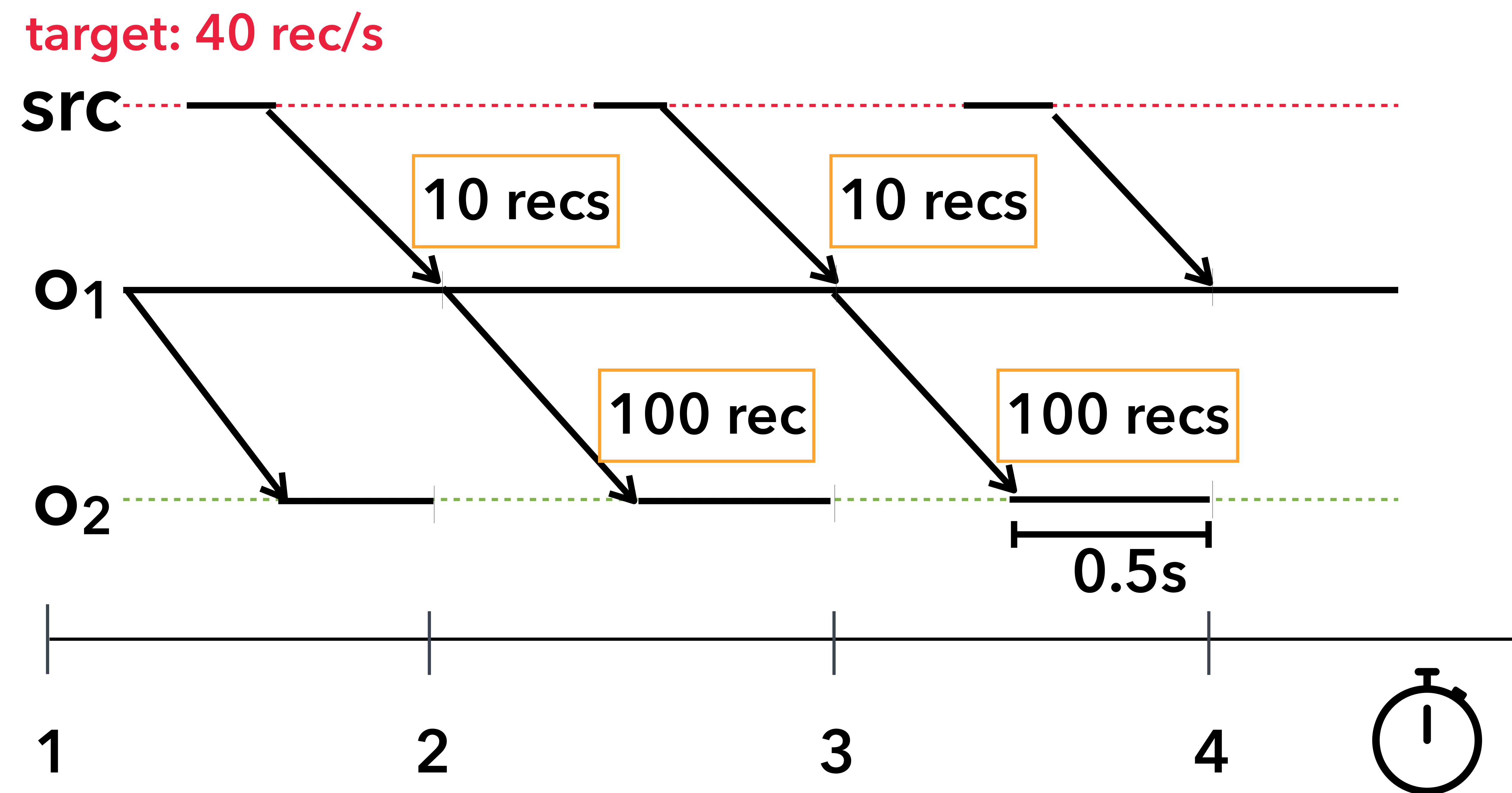
o2 cannot keep up

Which operator is the bottleneck?

What if we scale o_1 x 4?

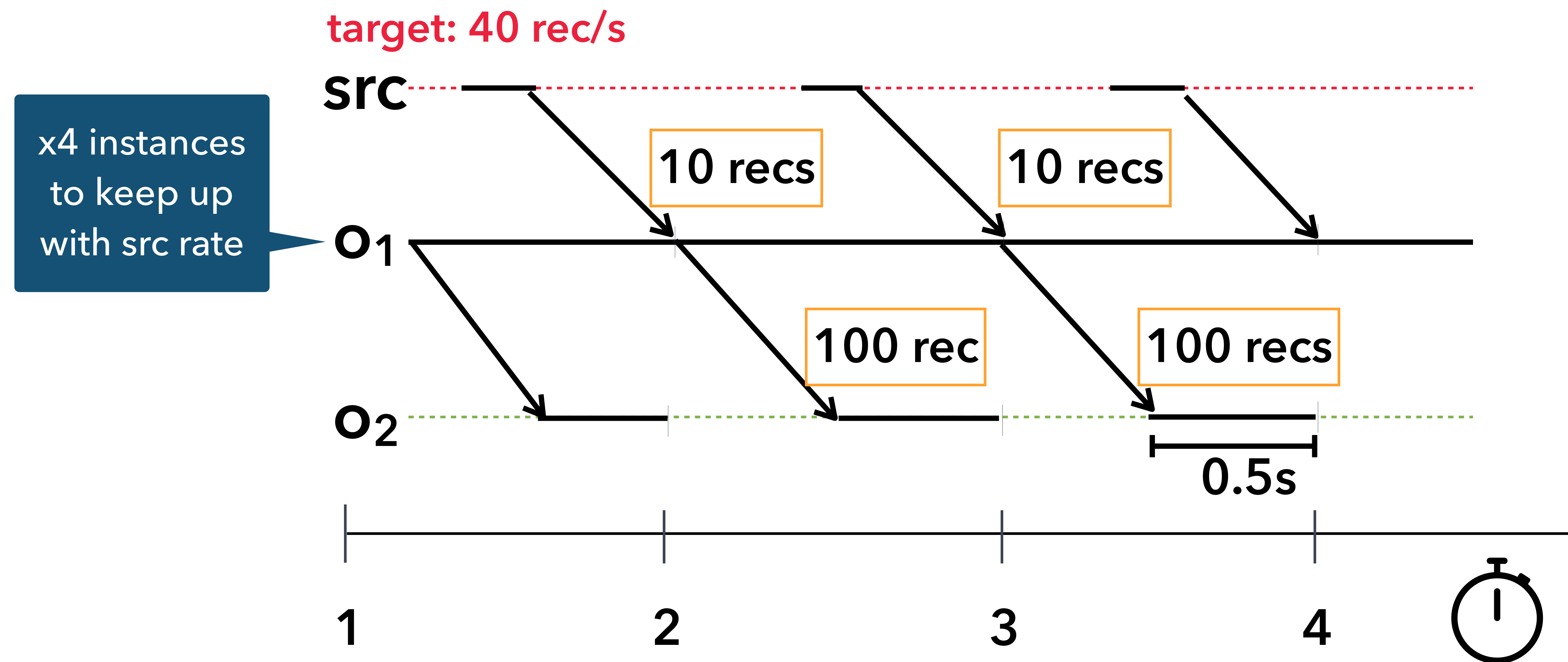
How much to scale o_2?

<u>Intuition</u>: use the dataflow graph to extract **operator dependencies** and **system instrumentation** to collect accurate, representative metrics.



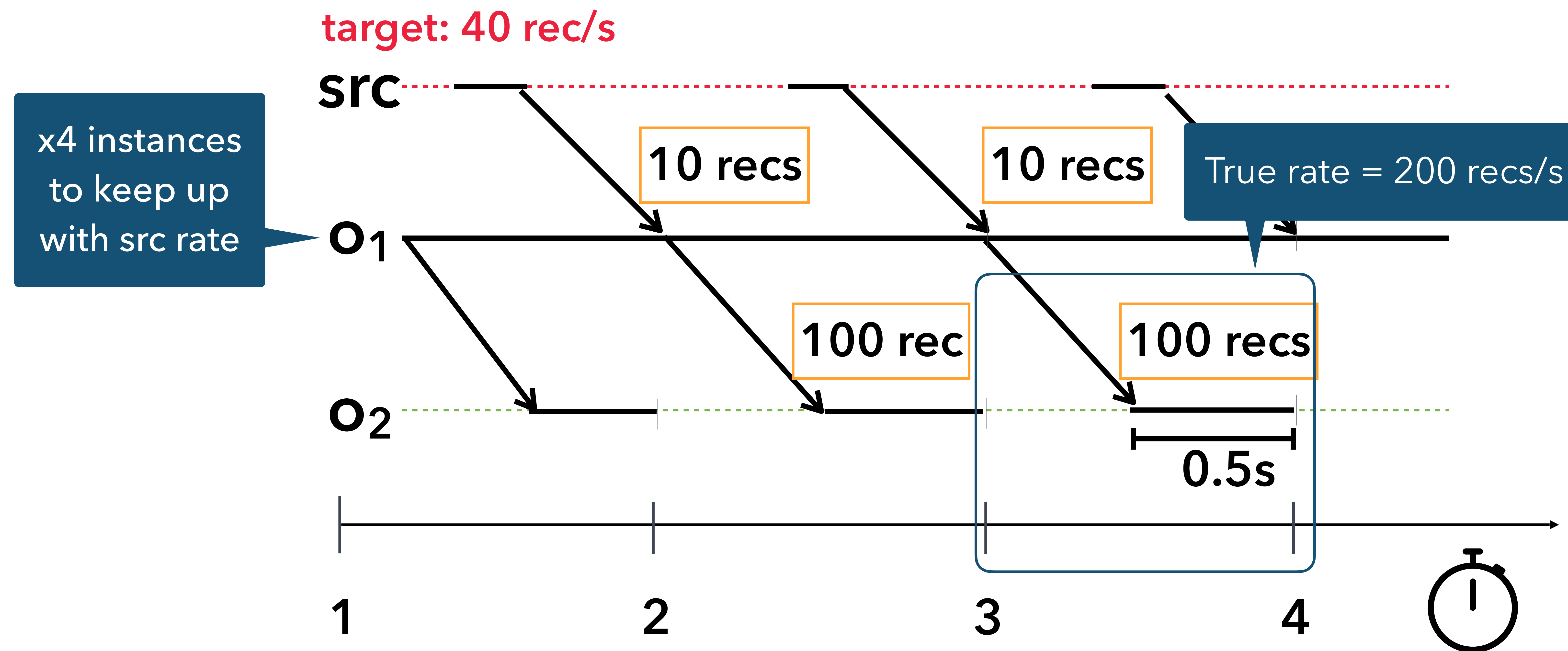
Vasiliki Kalavri | Boston University 2020

Intuition: use the dataflow graph to extract **operator dependencies** and **system instrumentation** to collect accurate, representative metrics.

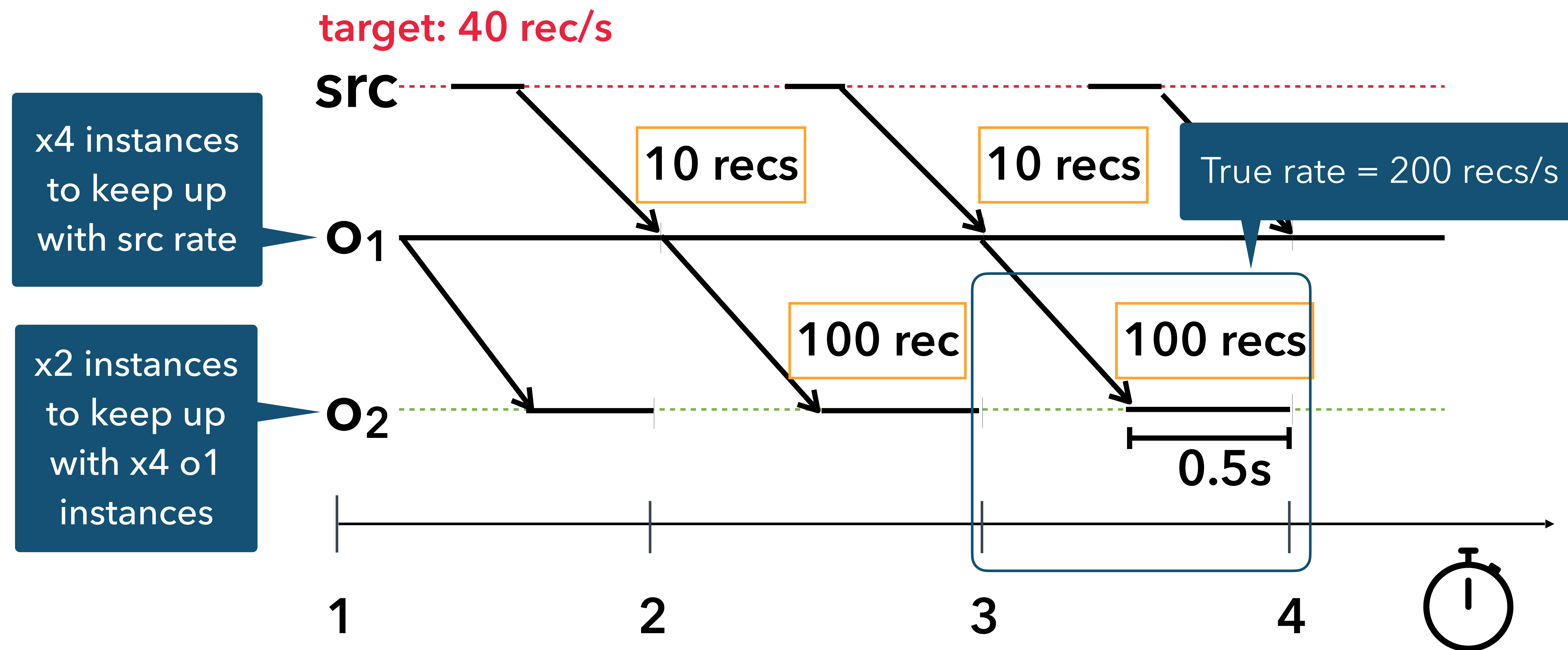


Vasiliki Kalavri | Boston University 2020

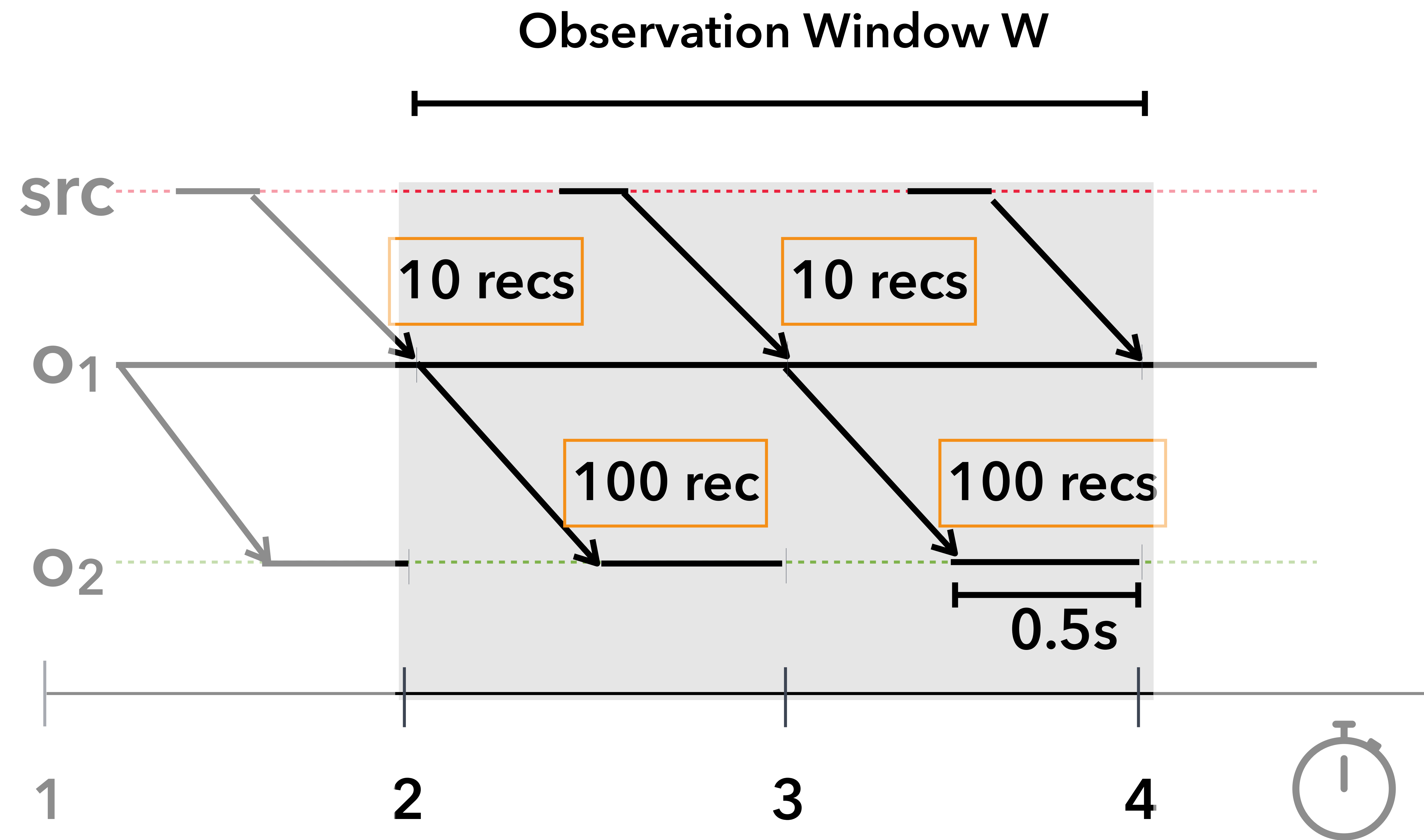
Intuition: use the dataflow graph to extract **operator dependencies** and **system instrumentation** to collect accurate, representative metrics.

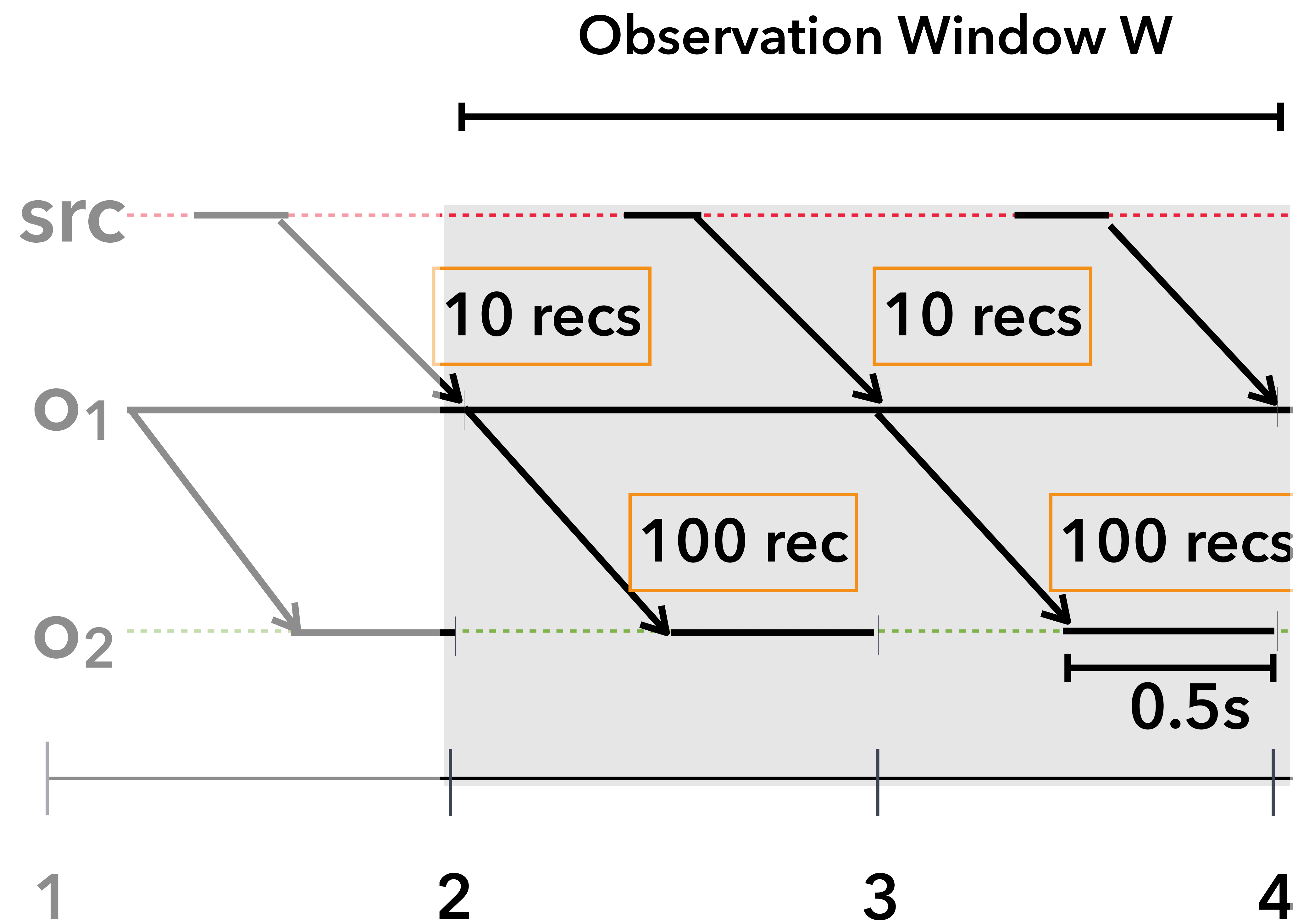


Intuition: use the dataflow graph to extract **operator dependencies** and **system instrumentation** to collect accurate, representative metrics.



Vasiliki Kalavri | Boston University 2020


Observation Window W
src
10 recs
10 recs
o1
100 rec
100 recs
o2
0.5s
1
2
3
4



Instrumentation Metrics

	o1	o2
Records processed R_{pcd}	20	200
Records pushed R_{psd}	200	-
Useful time W_u	2s	1s

The DS2 model

The DS2 model

- Collect metrics per configurable **observation** window **W**
  - **activity durations** per worker
  - records processed $\mathbf{R_{prc}}$ and records pushed to output $\mathbf{R_{psd}}$

The DS2 model

- Collect metrics per configurable **observation** window **W**
  - **activity durations** per worker
  - records processed $\mathbf{R_{prc}}$ and records pushed to output $\mathbf{R_{psd}}$
- Capture **dependencies** through the dataflow graph
  - assign an increasing **sequential id** to all operators in topological order, starting from the sources
  - represent as an **adjacency** matrix **A**
    - $A_{ij} = 1$ iff operator i is upstream neighbor of j

Useful time W_u

The time spent by an operator instance in **deserialization**, **processing**, and **serialization** activities.

- excludes any time spent waiting on input or on output
- amounts to the time an operator instance runs for if executed in an *ideal* setting
  - when there is no waiting the useful time is equal to the **observed time**

True processing / output rates

$$\lambda_p = \frac{R_{\text{prc}}}{W_u} \qquad \lambda_o = \frac{R_{\text{psd}}}{W_u}$$

Aggregated true processing / output rates

$$o_i[\lambda_p] = \sum_{k=1}^{k=p_i} \lambda_p^k \qquad o_i[\lambda_o] = \sum_{k=1}^{k=p_i} \lambda_o^k$$

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j : j < i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m$$

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m$$

captures upstream operators

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i}\right)^{-1} \right\rceil, n \leq i < m$$

captures upstream operators

Aggregated true output rate of operator o_j, when o_j itself and all upstream ops are deployed with optimal parallelism

Optimal parallelism per operator

$$\pi_i = \left[\sum_{\forall j : j < i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right], n \leq i < m$$

captures upstream operators

Aggregated true output rate of operator o_j, when o_j itself and all upstream ops are deployed with optimal parallelism

current parallelism of operator i

Recursively computed as:

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda_{\text{src}}^{j}, & 0 \leq j < n \\ \frac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum_{\forall u : u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases}$$

Recursively computed as:

True output rate of source j

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda_{\text{src}}^j, & 0 \le j < n \\ \frac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum_{\forall u: u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \le j < m \end{cases}$$

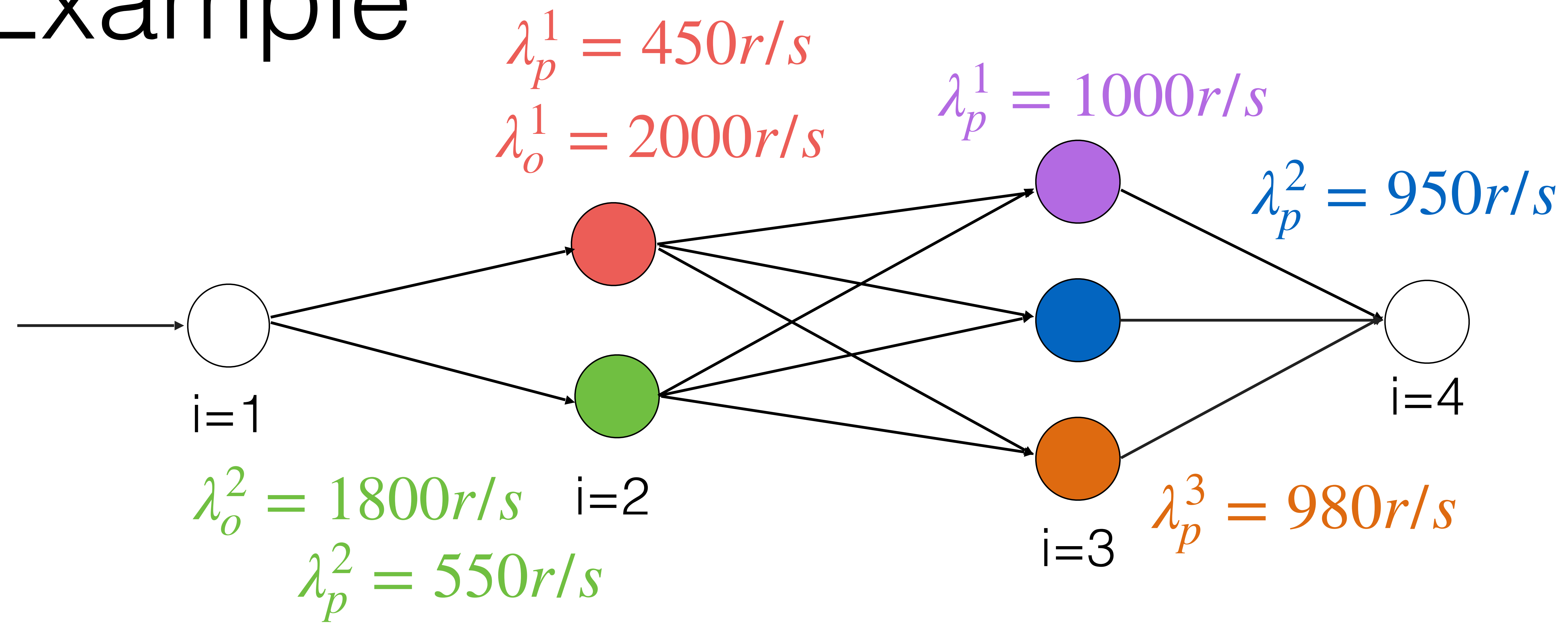
Recursively computed as:

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda_{\text{src}}^j, & 0 \leq j < n \\ \dfrac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum\limits_{\forall u: u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases}$$

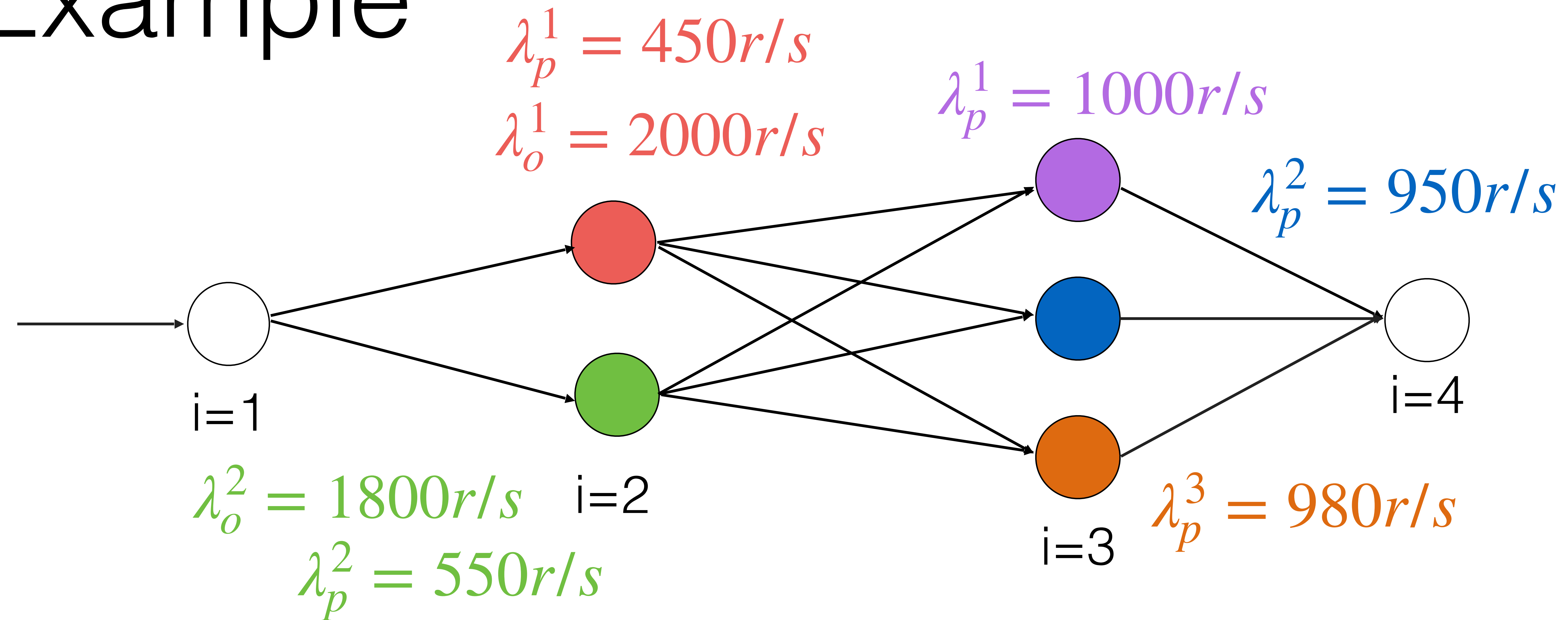
True output rate of source j

It can be computed **for all operators** by traversing the dataflow from left to right **once**

Example



Example

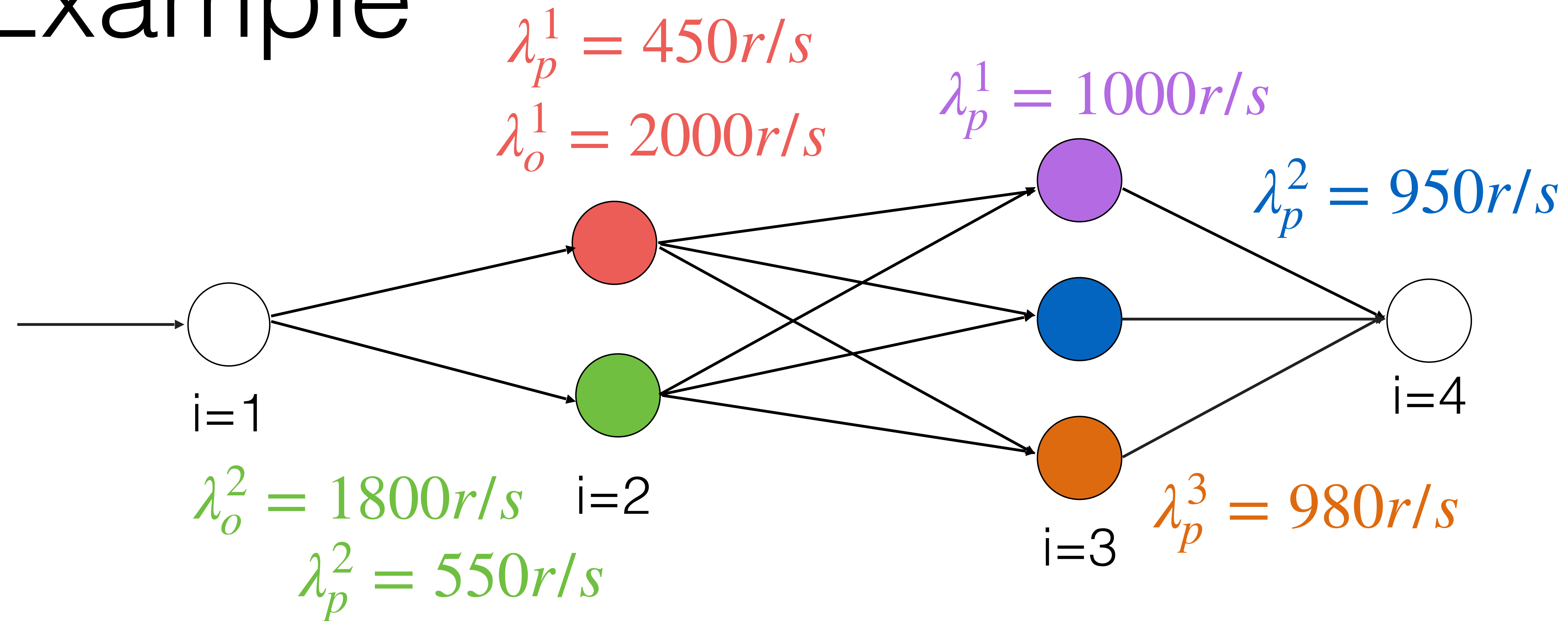


$o_1[\lambda_p] = 0$
$o_1[\lambda_o] = 2000$ r/s

$o_2[\lambda_p] = 1000$ r/s
$o_2[\lambda_o] = 3800$ r/s

$o_3[\lambda_p] = 2930$ r/s
$o_3[\lambda_o] = 600$ r/s

Example



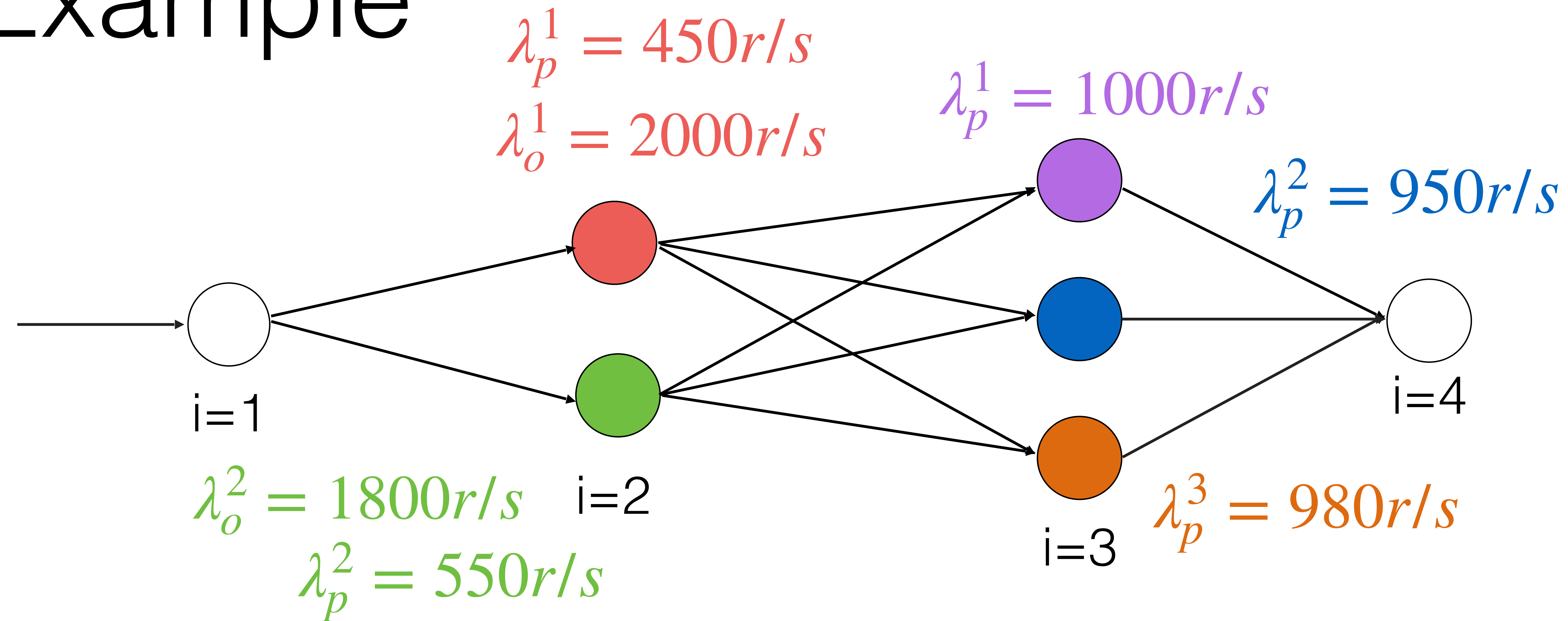
$o_1[\lambda_p] = 0$
$o_1[\lambda_o] = 2000$ r/s

$o_2[\lambda_p] = 1000$ r/s
$o_2[\lambda_o] = 3800$ r/s

$o_3[\lambda_p] = 2930$ r/s
$o_3[\lambda_o] = 600$ r/s

$$\pi_2 = o_1[\lambda_o^*] * \frac{p_2}{o_2[\lambda_p]} = 2000 * \frac{2}{1000} = 4$$

Example



$o_1[\lambda_p] = 0$
$o_1[\lambda_o] = 2000$ r/s

$o_2[\lambda_p] = 1000$ r/s
$o_2[\lambda_o] = 3800$ r/s

$o_3[\lambda_p] = 2930$ r/s
$o_3[\lambda_o] = 600$ r/s

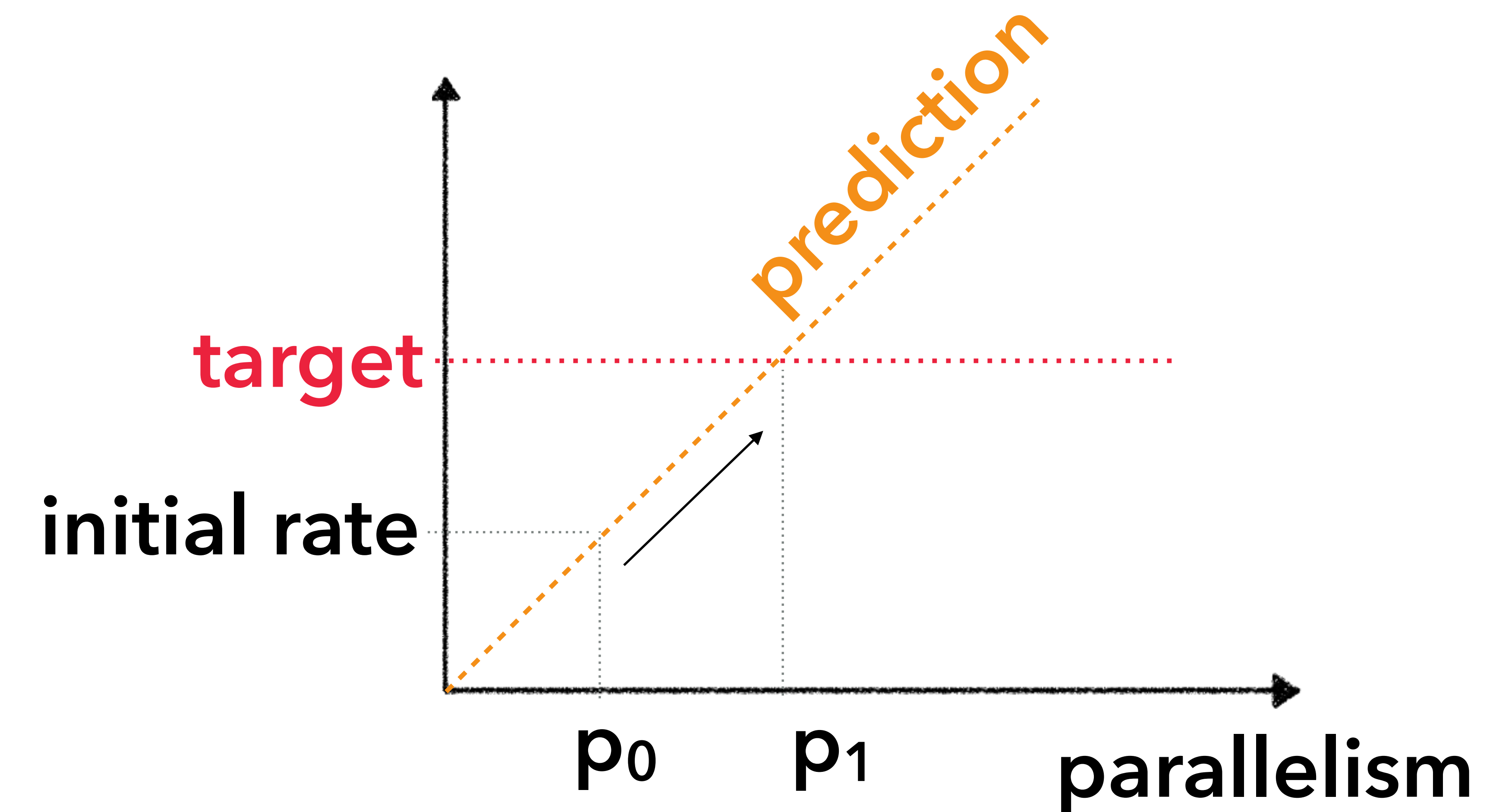
$$\pi_2 = o_1[\lambda_o^*] * \frac{p_2}{o_2[\lambda_p]} = 2000 * \frac{2}{1000} = 4$$

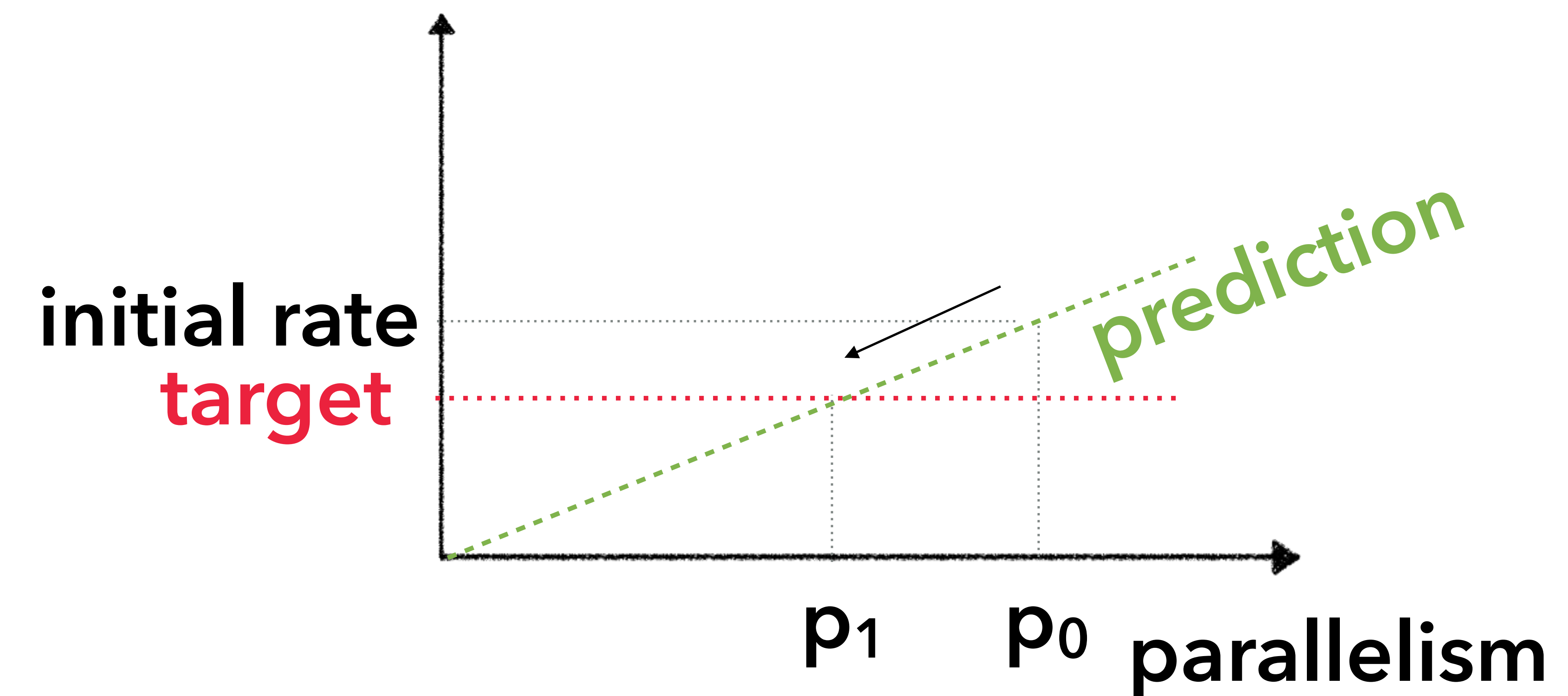
$$\pi_3 = o_2[\lambda_o^*] * \frac{p_3}{o_3[\lambda_p]} = 7600 * \frac{3}{2930} \approx 7.78 \rightarrow 8$$

DS2 model properties

If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down

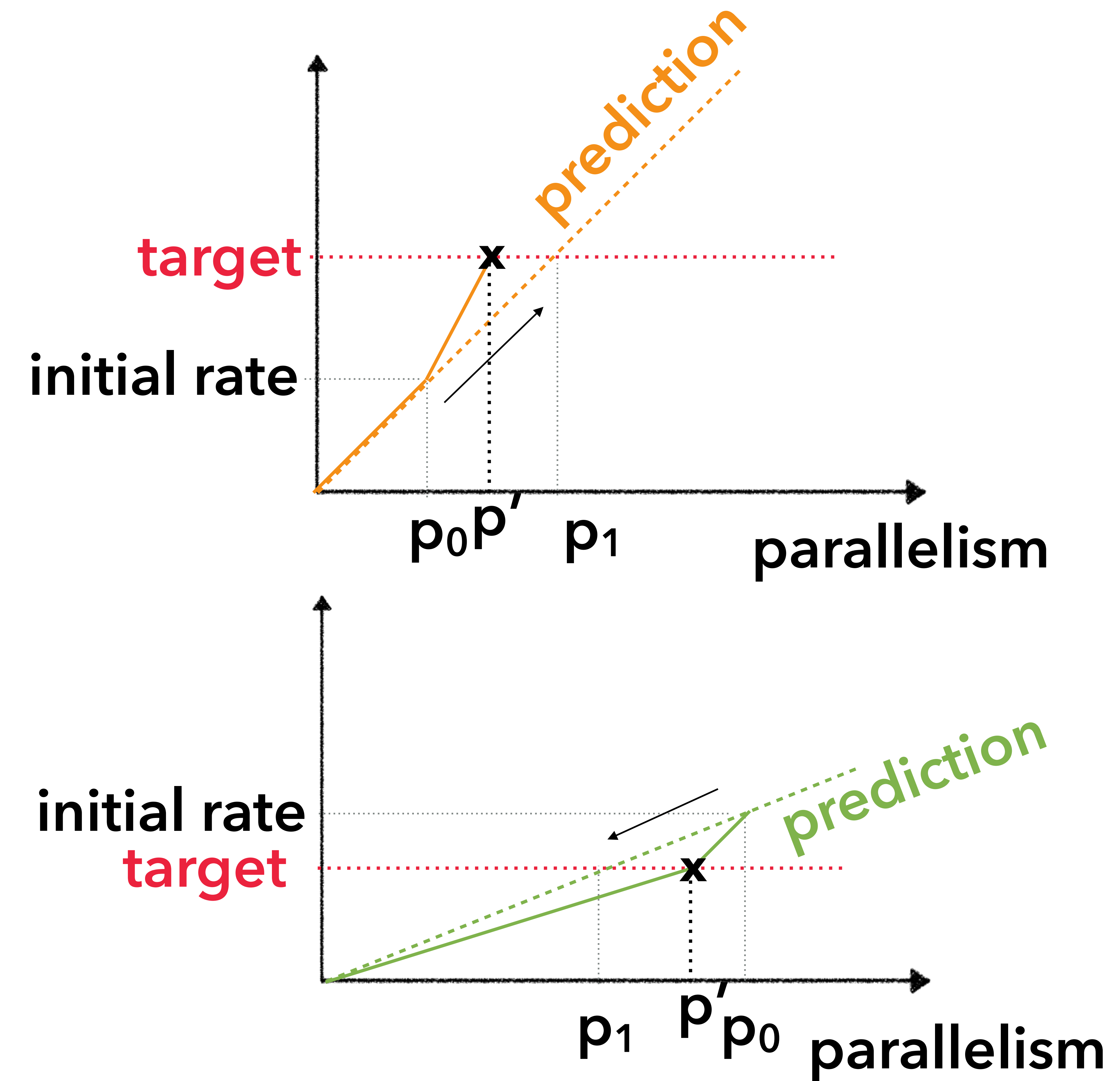




DS2 model properties

If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down



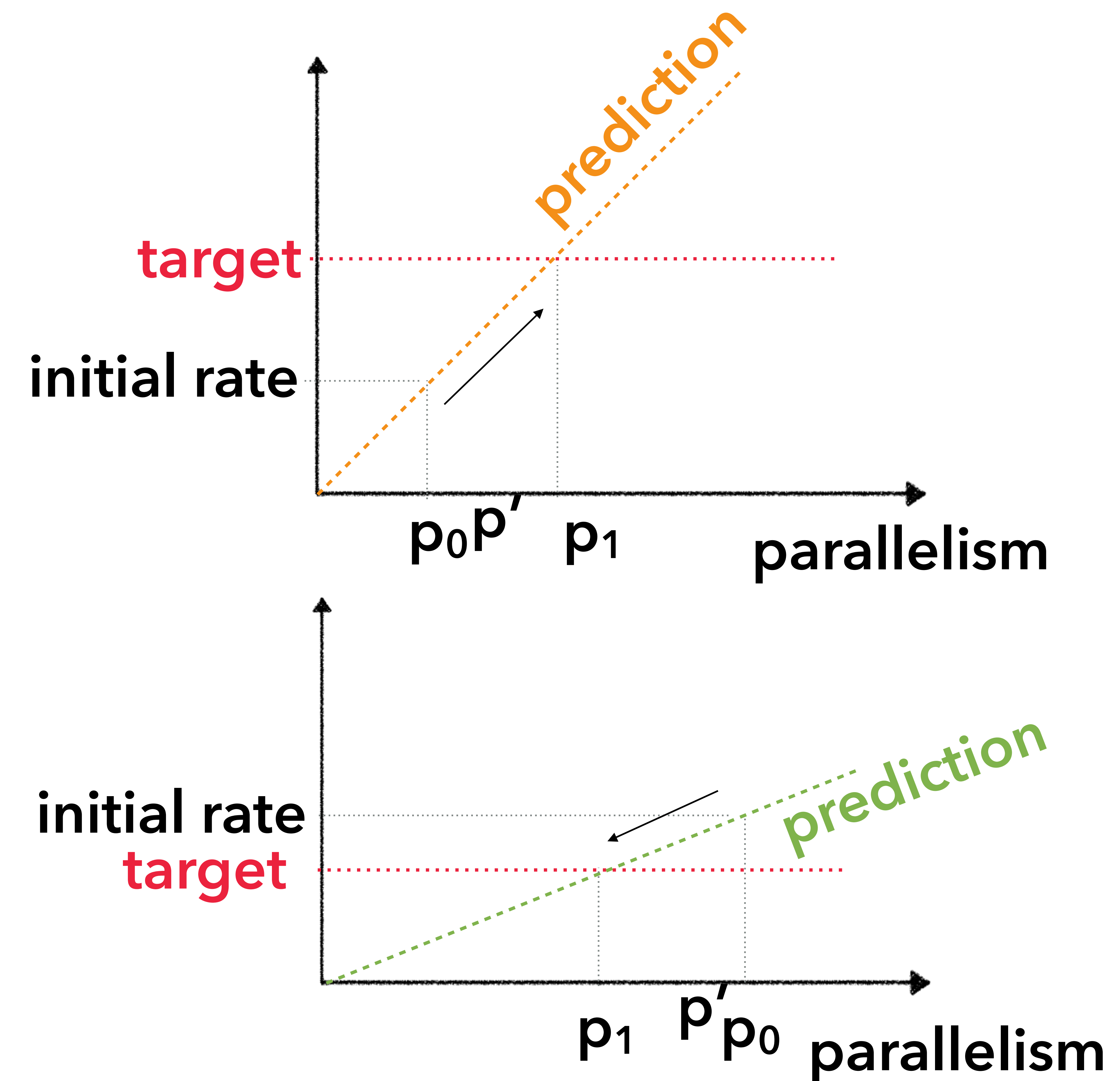
DS2 model properties

If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down

Ideal rates act as un upper bound when scaling up and as a lower bound when scaling down:

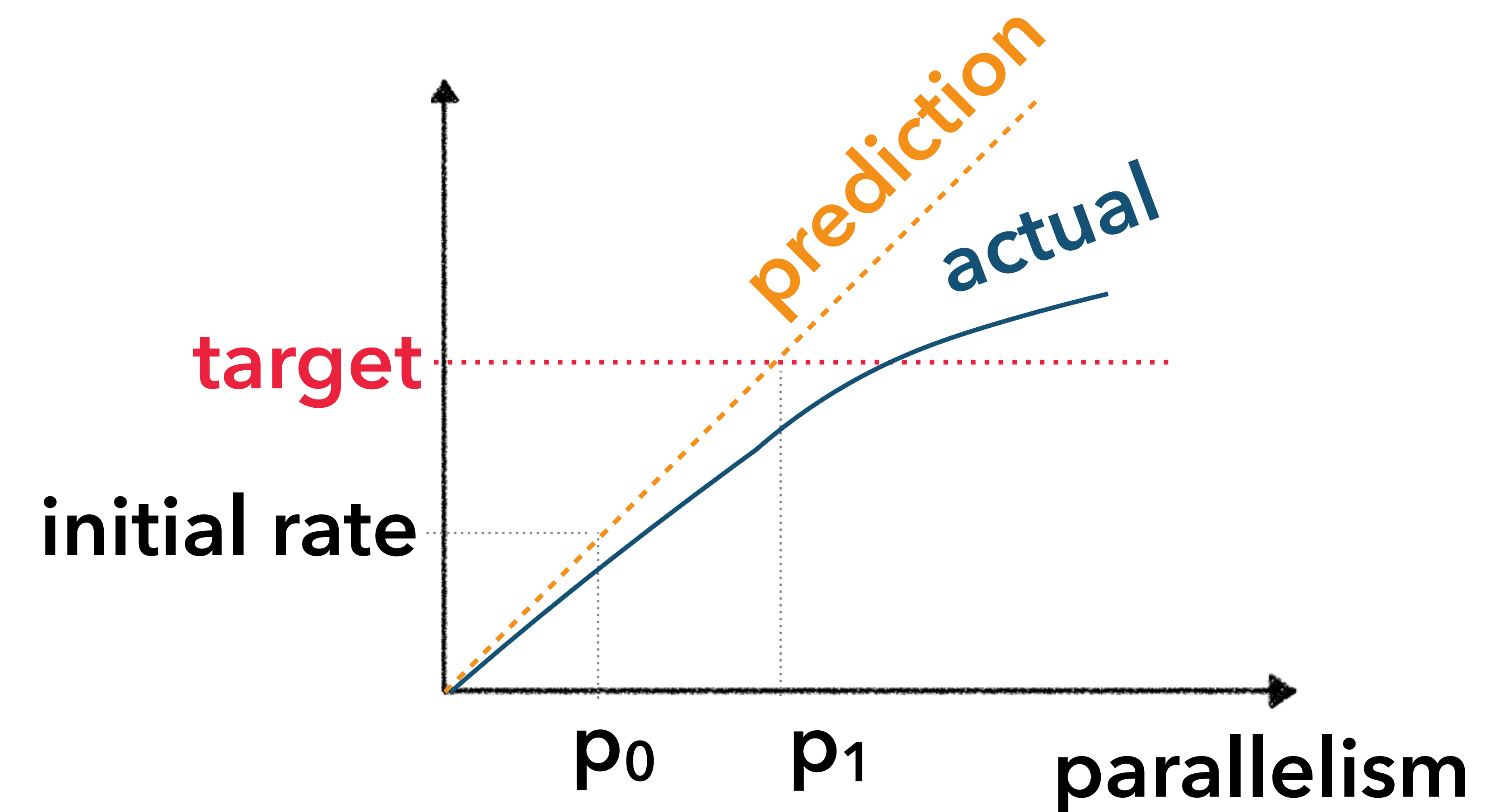
DS2 will **converge monotonically** to the target rate



DS2 model properties

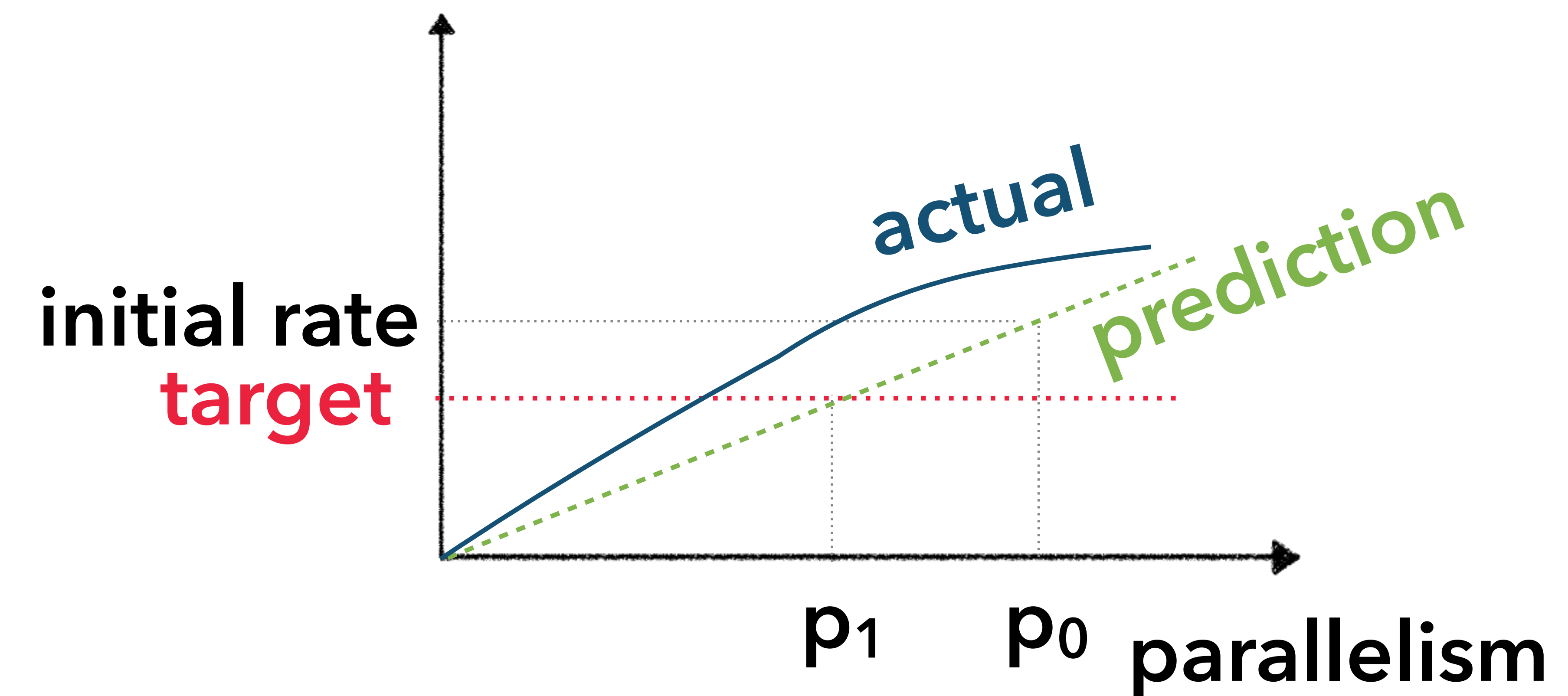
If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down

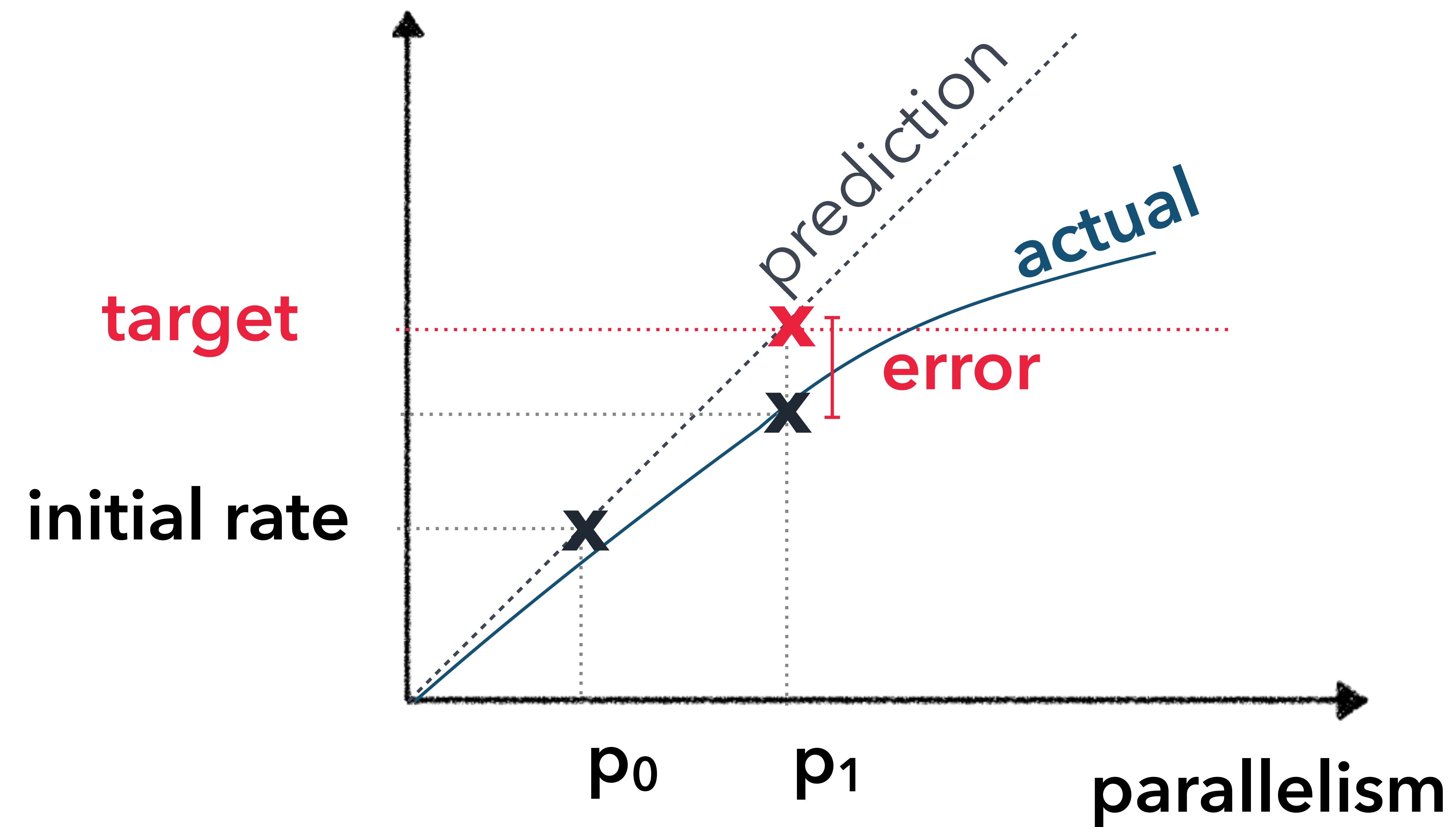


Ideal rates act as un upper bound when scaling up and as a lower bound when scaling down:

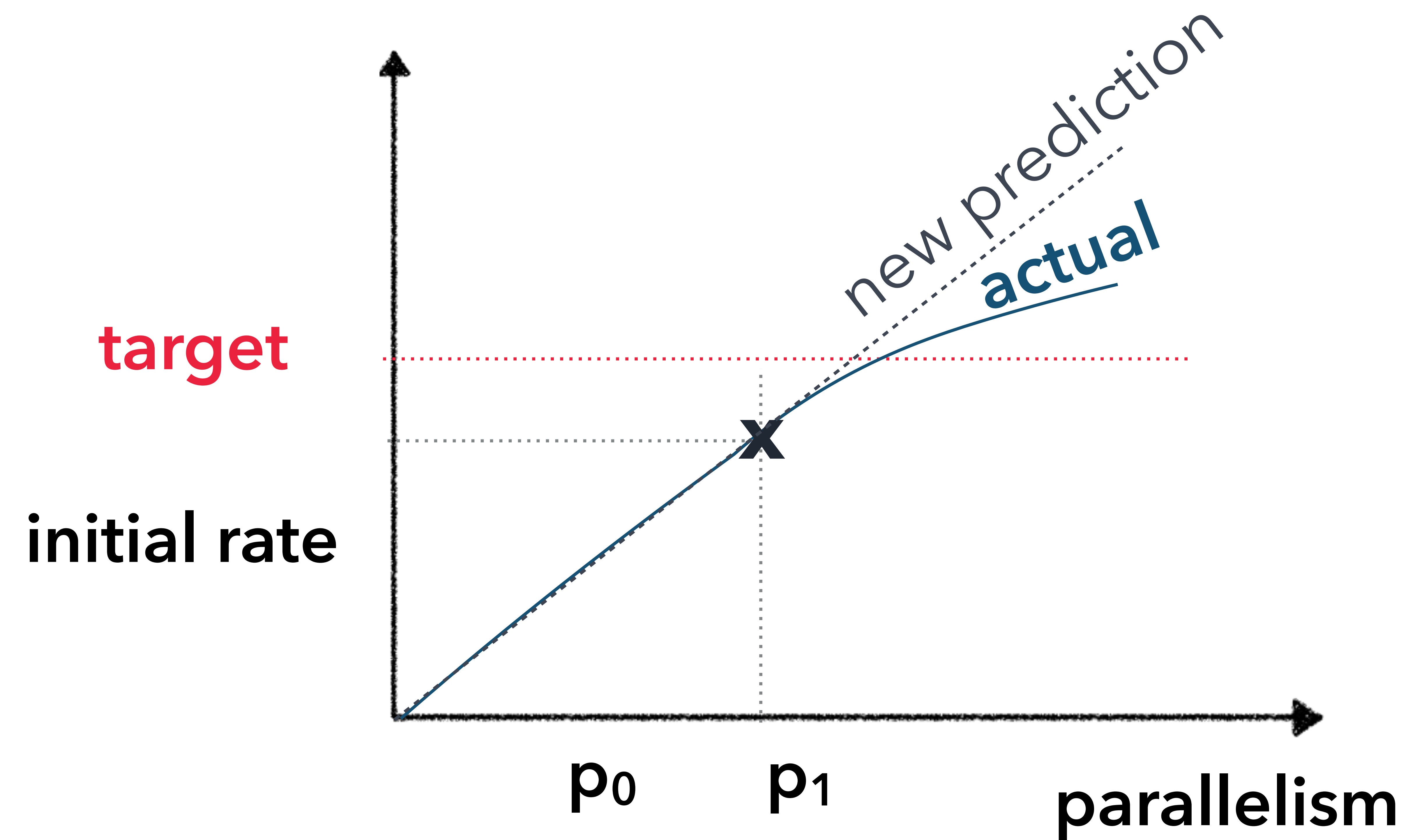
DS2 will **converge monotonically** to the target rate



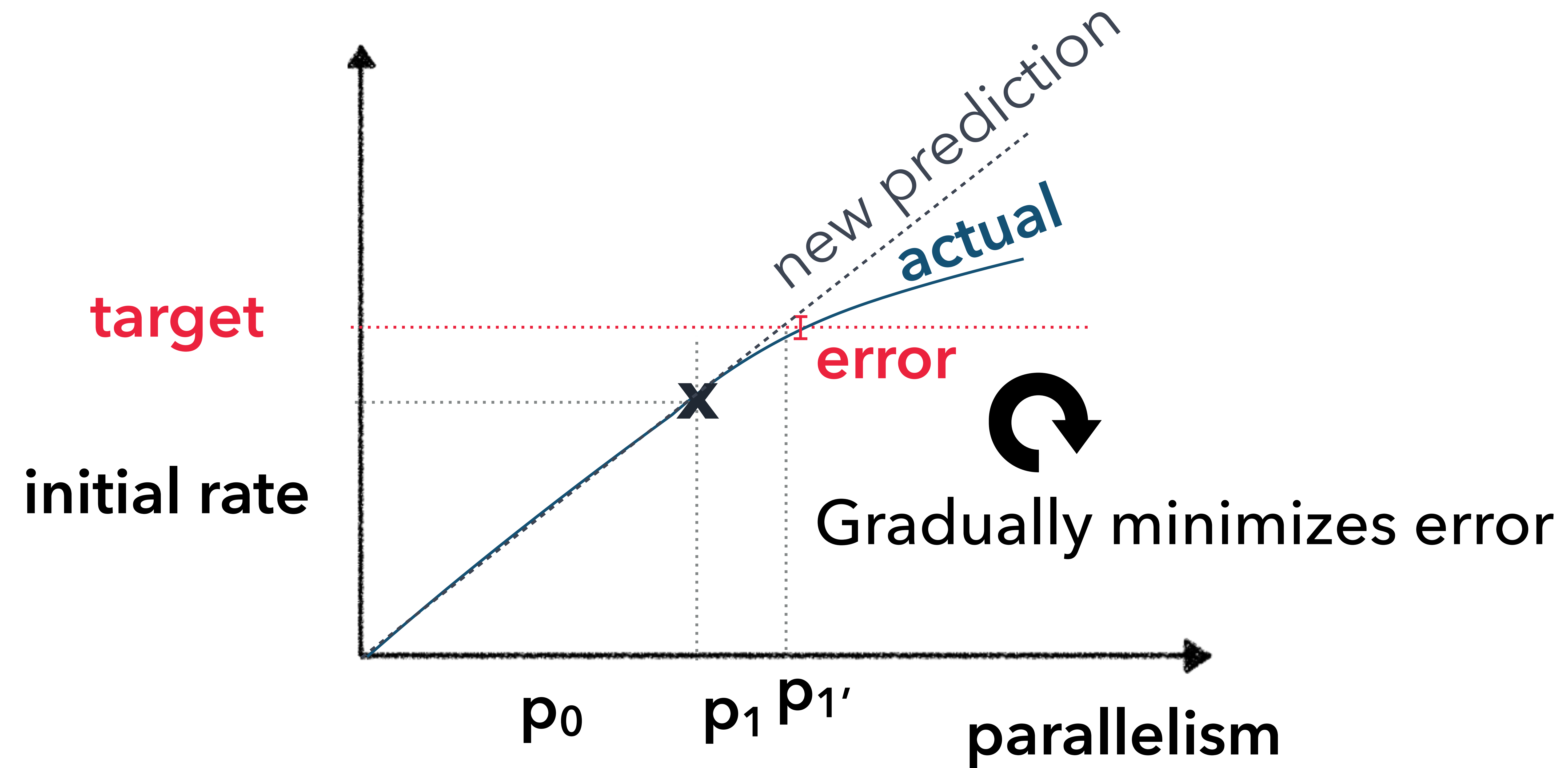
DS2 model properties

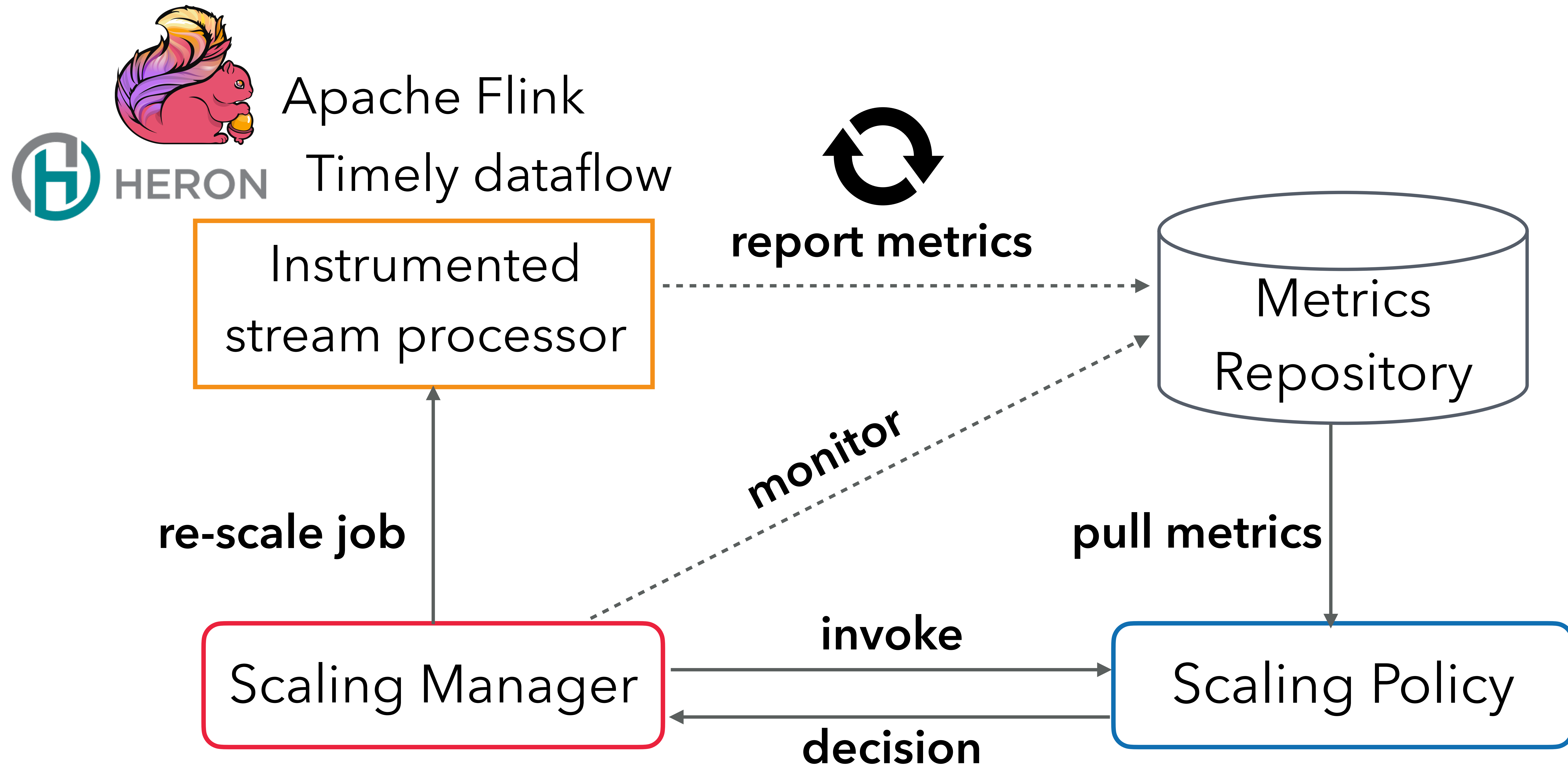


DS2 model properties



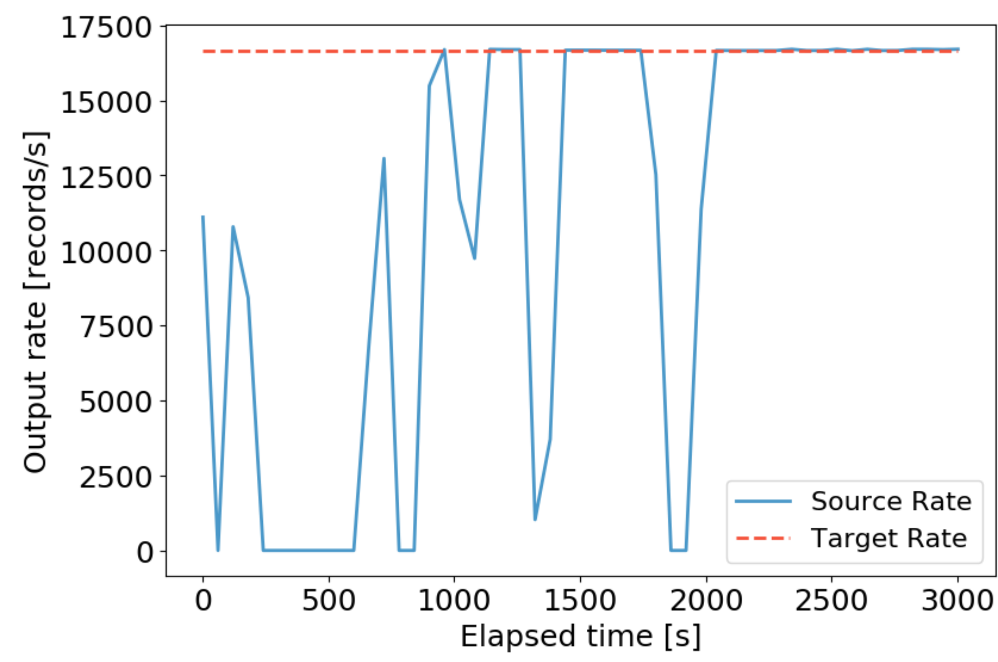
DS2 model properties

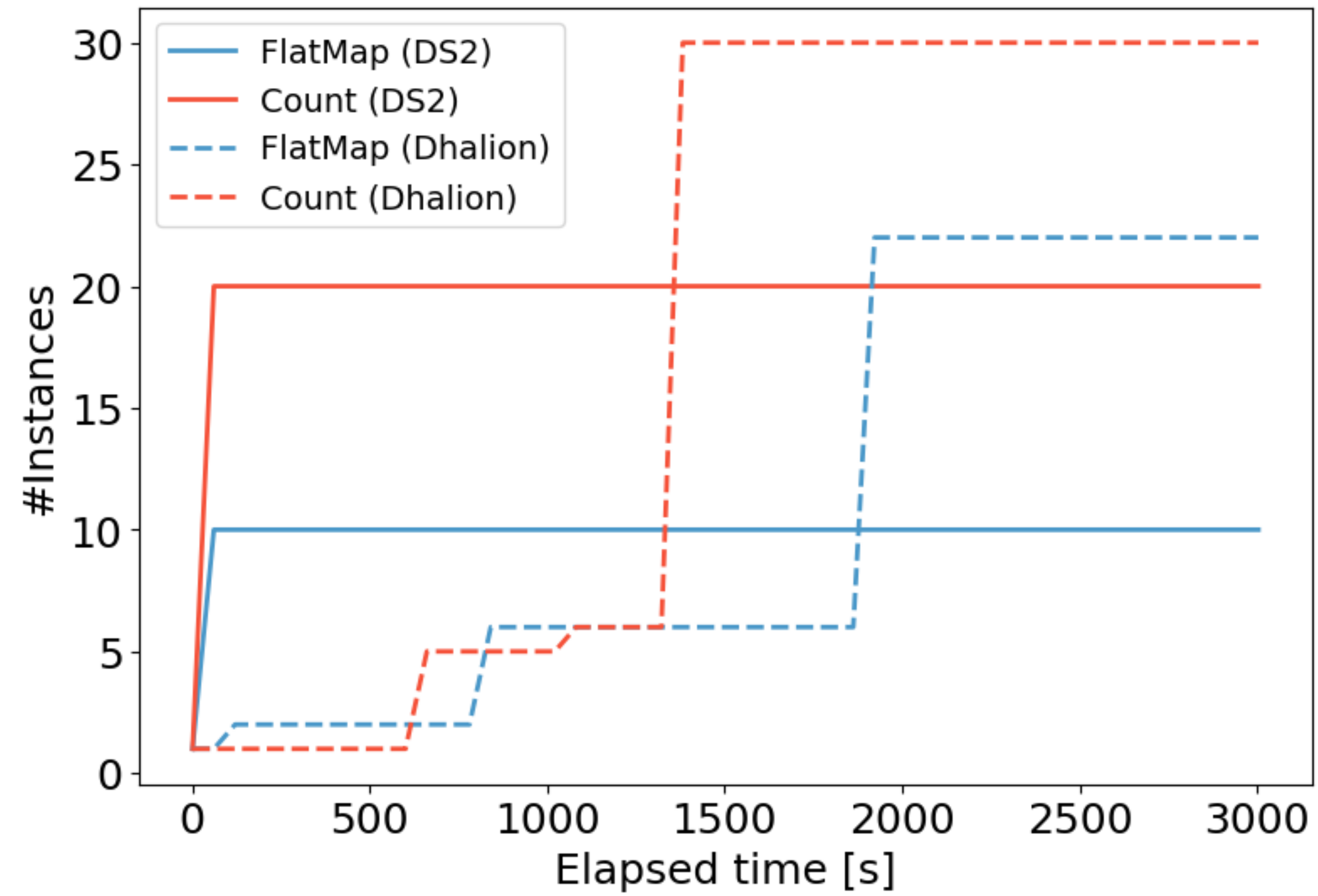



Apache Flink
HERON
Timely dataflow
Instrumented stream processor
report metrics
Metrics Repository
monitor
re-scale job
pull metrics
invoke
Scaling Manager
Scaling Policy
decision

Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

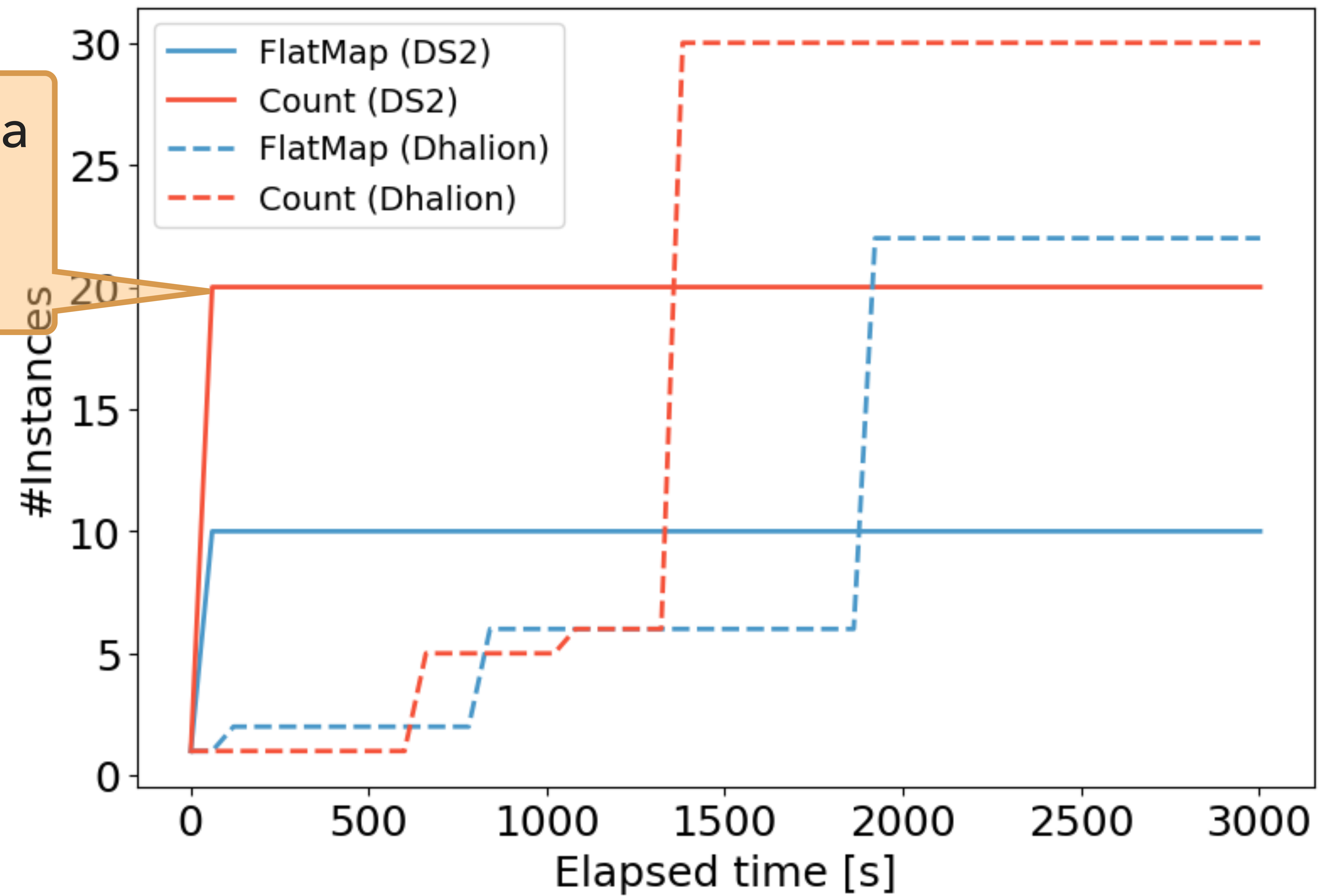




Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

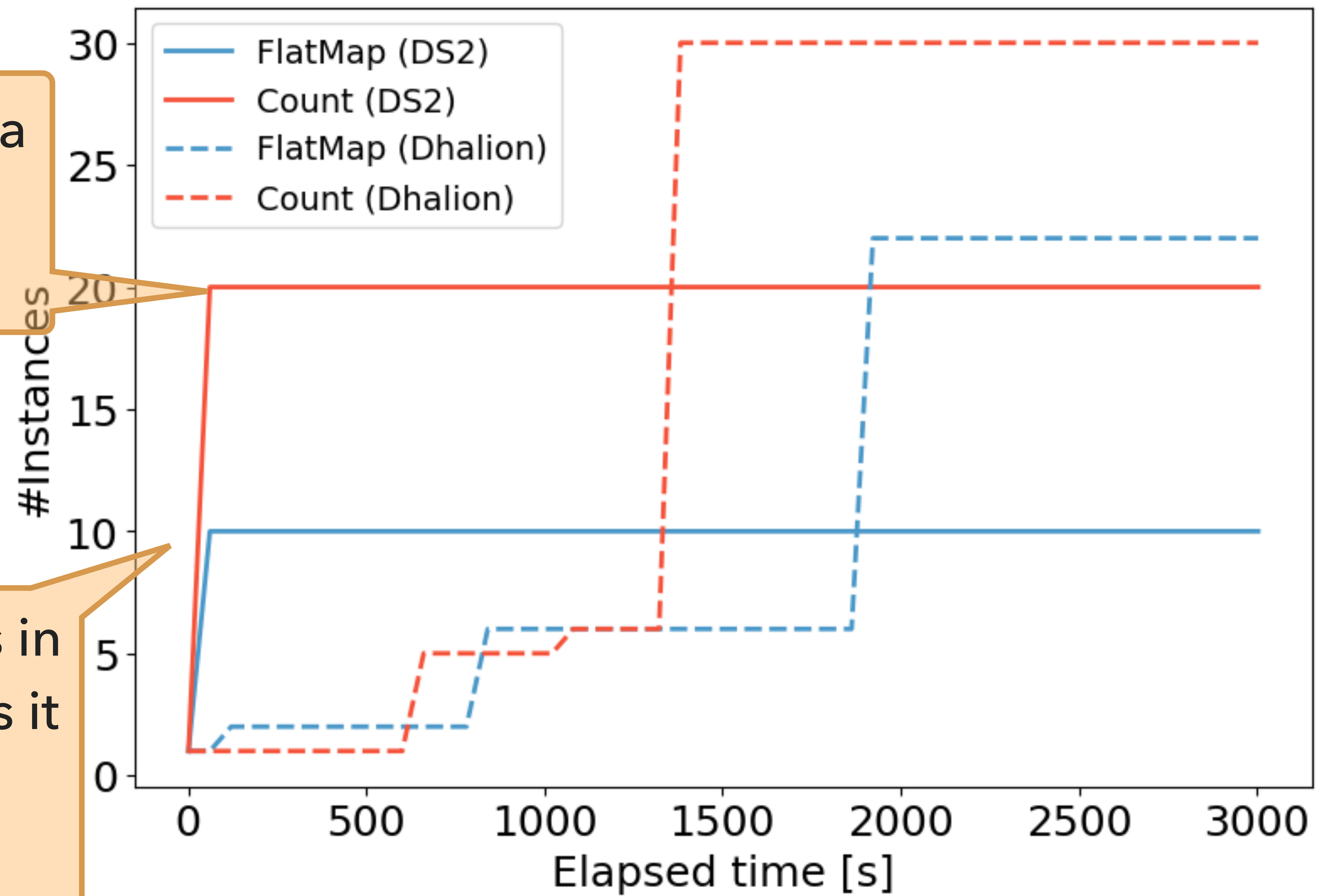
DS2 converges in a **single step** for both operators



Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

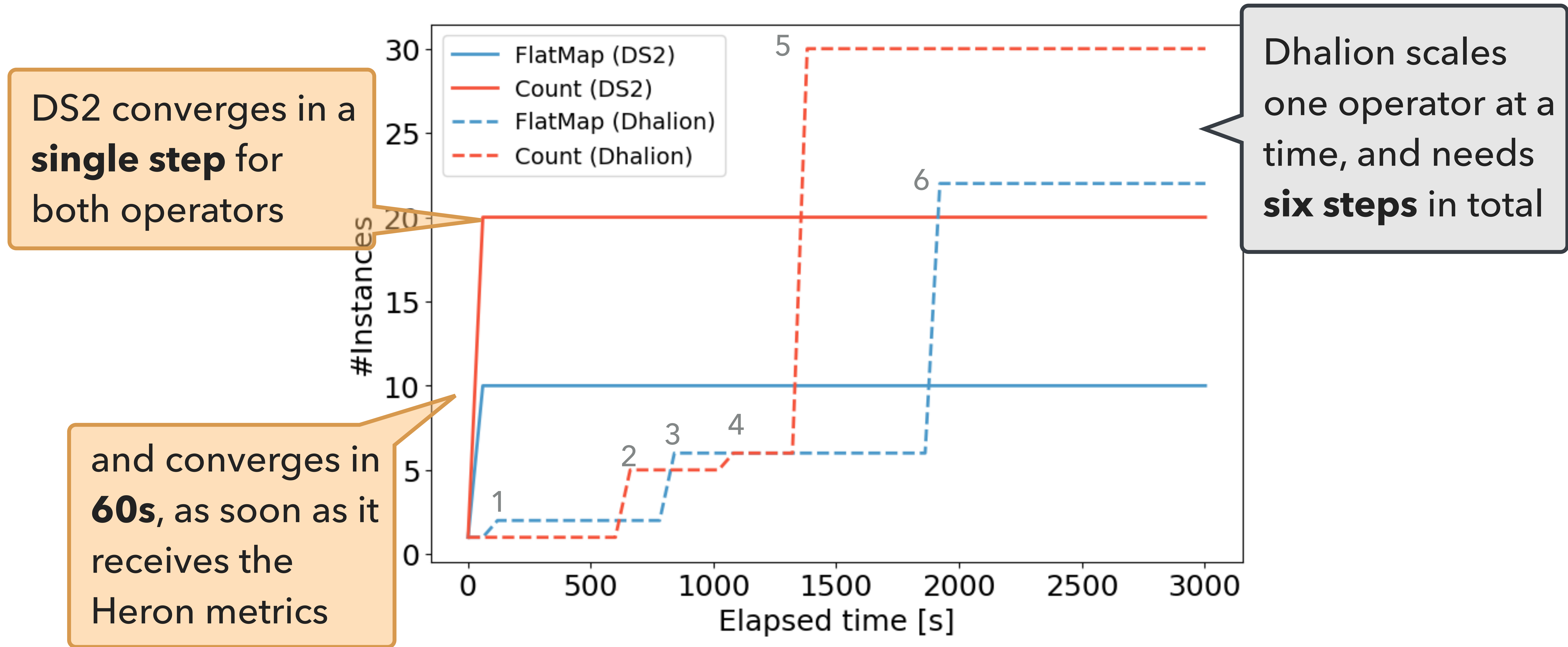
DS2 converges in a **single step** for both operators



and converges in **60s**, as soon as it receives the Heron metrics

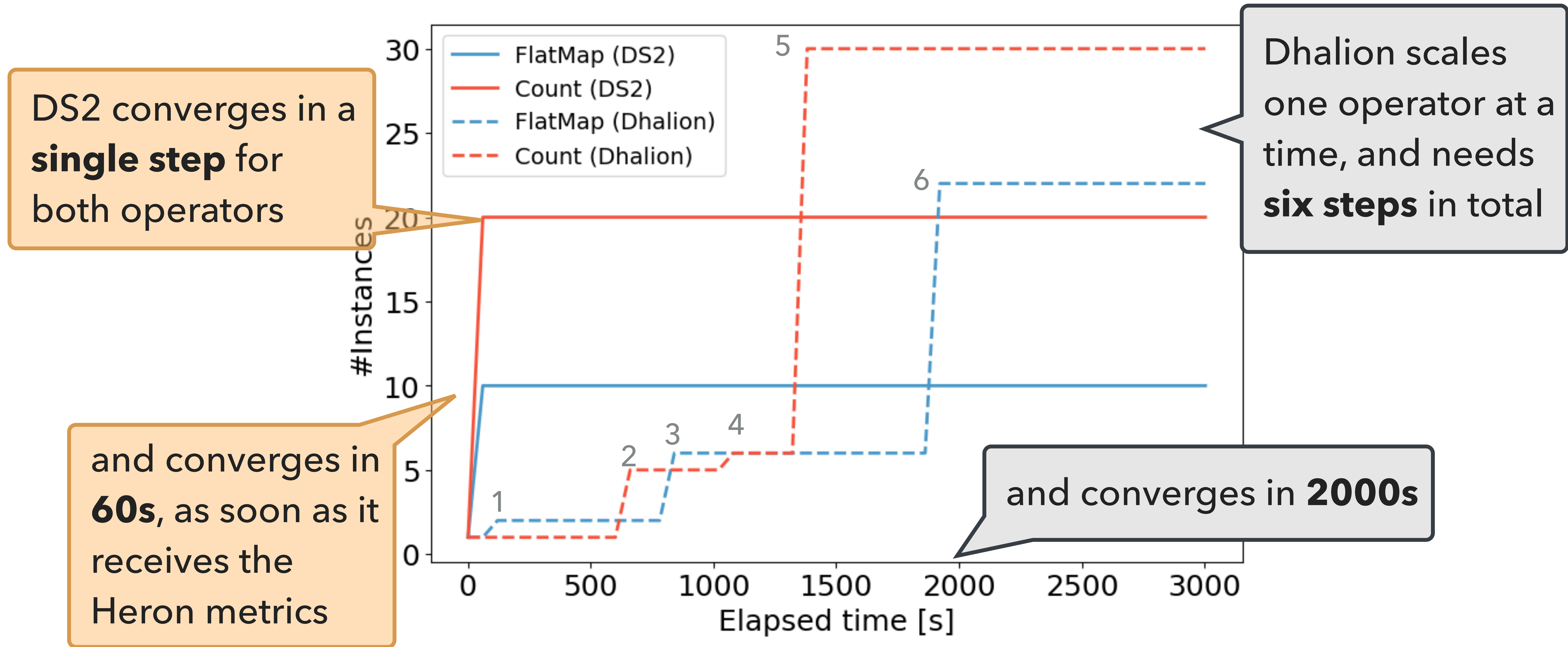
Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s



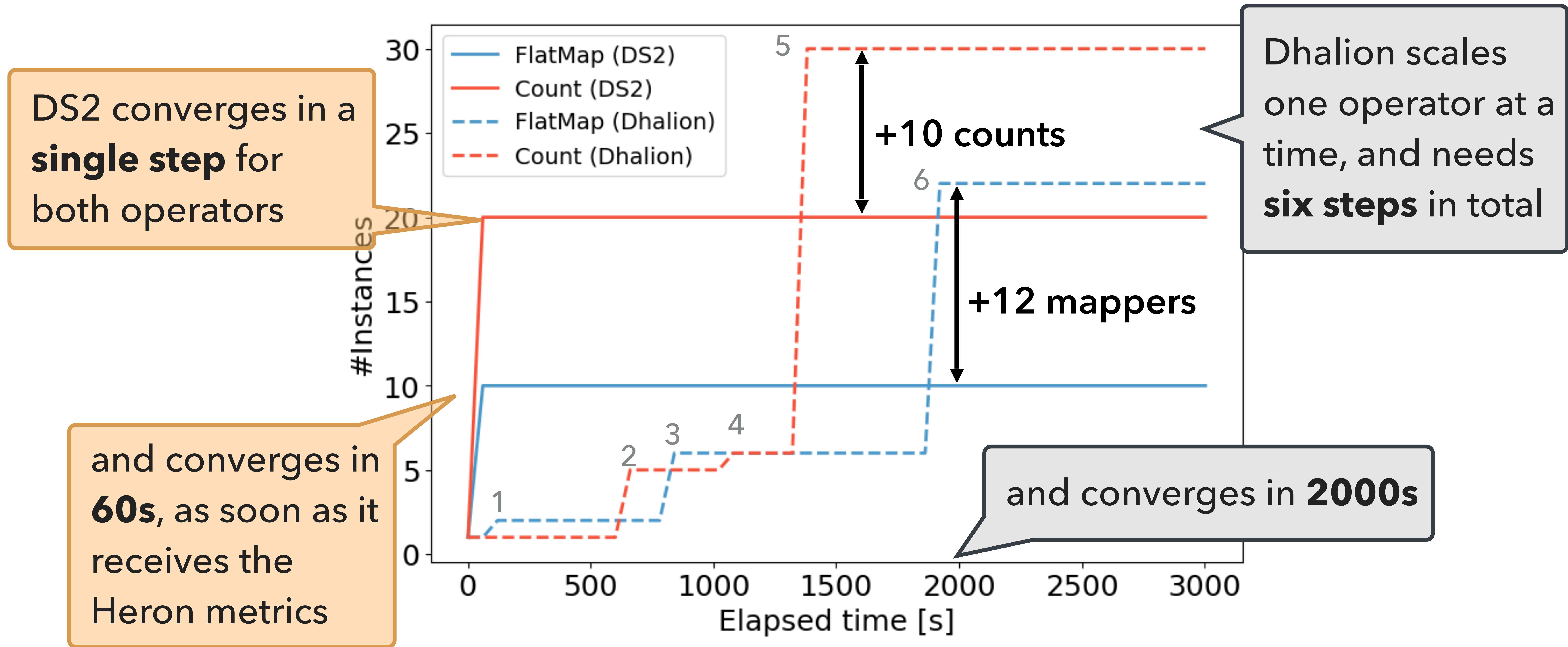
Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s



Initially under-provisioned wordcount dataflow

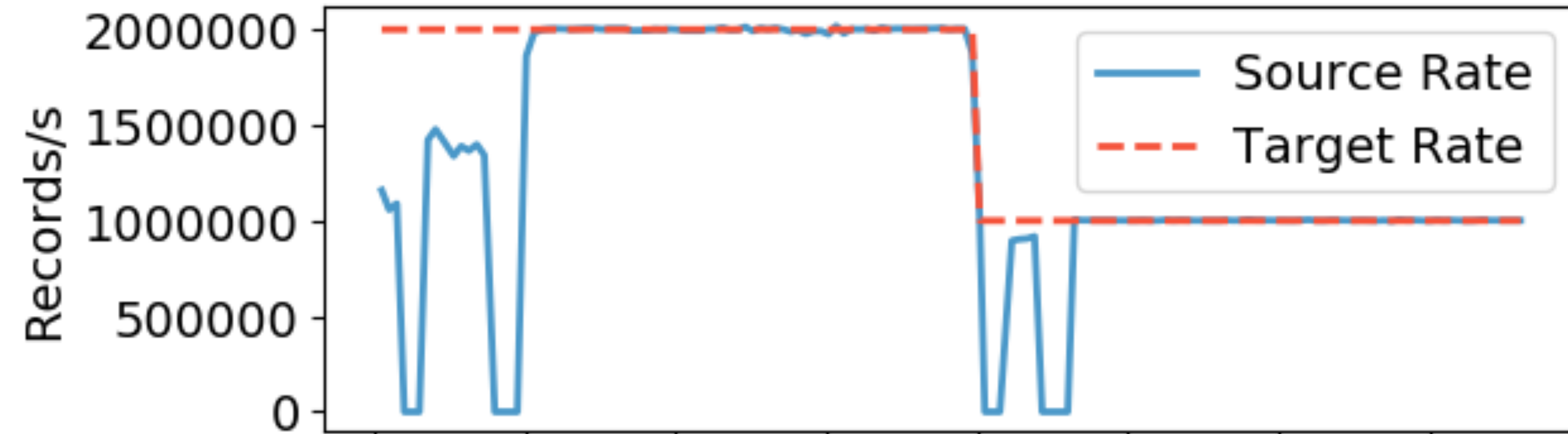
Target rate: 16.700 rec/s

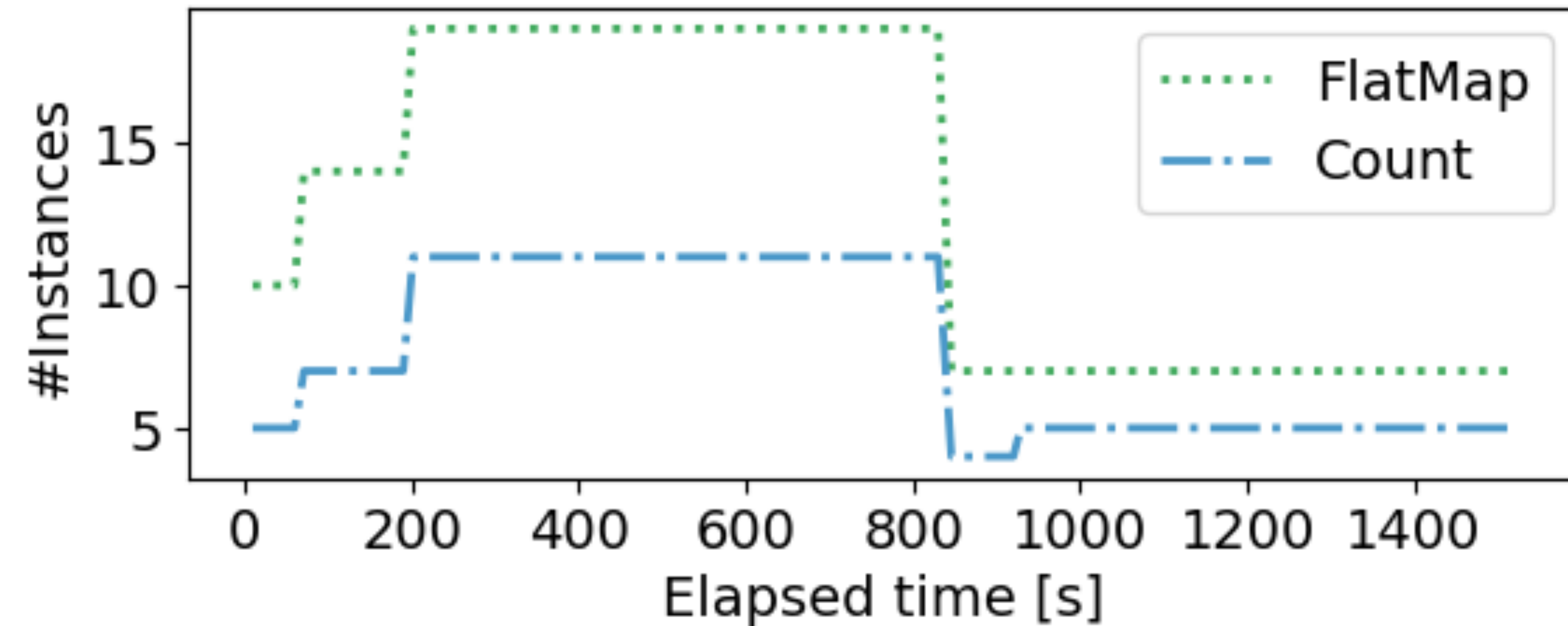


DS2 on Flink

Initially under-provisioned wordcount

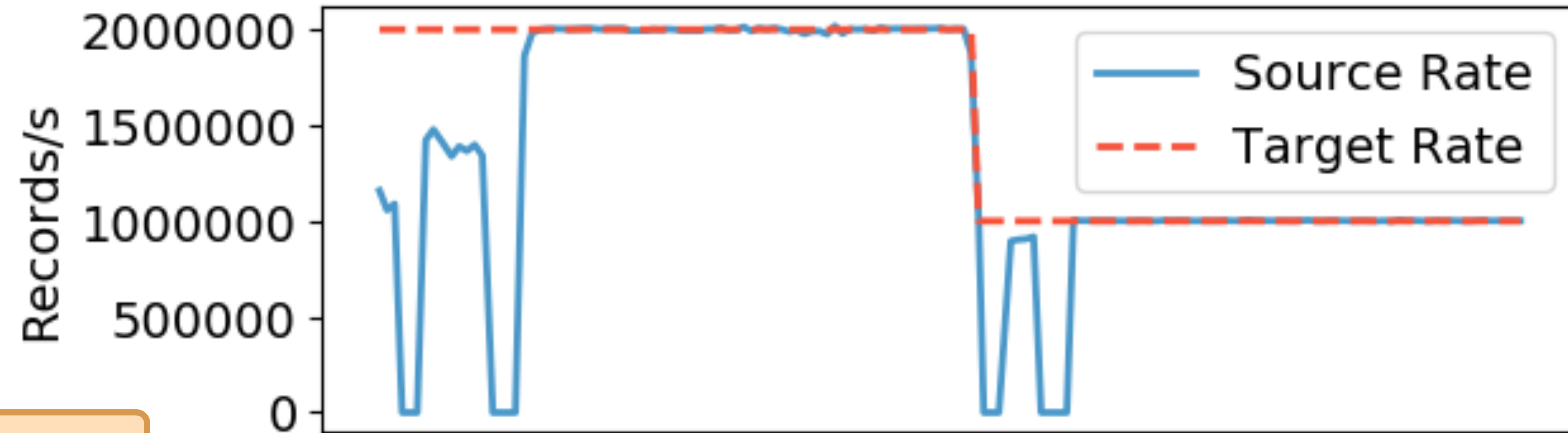
Target rate: 2.000.000 rec/s, drops to half at 800s

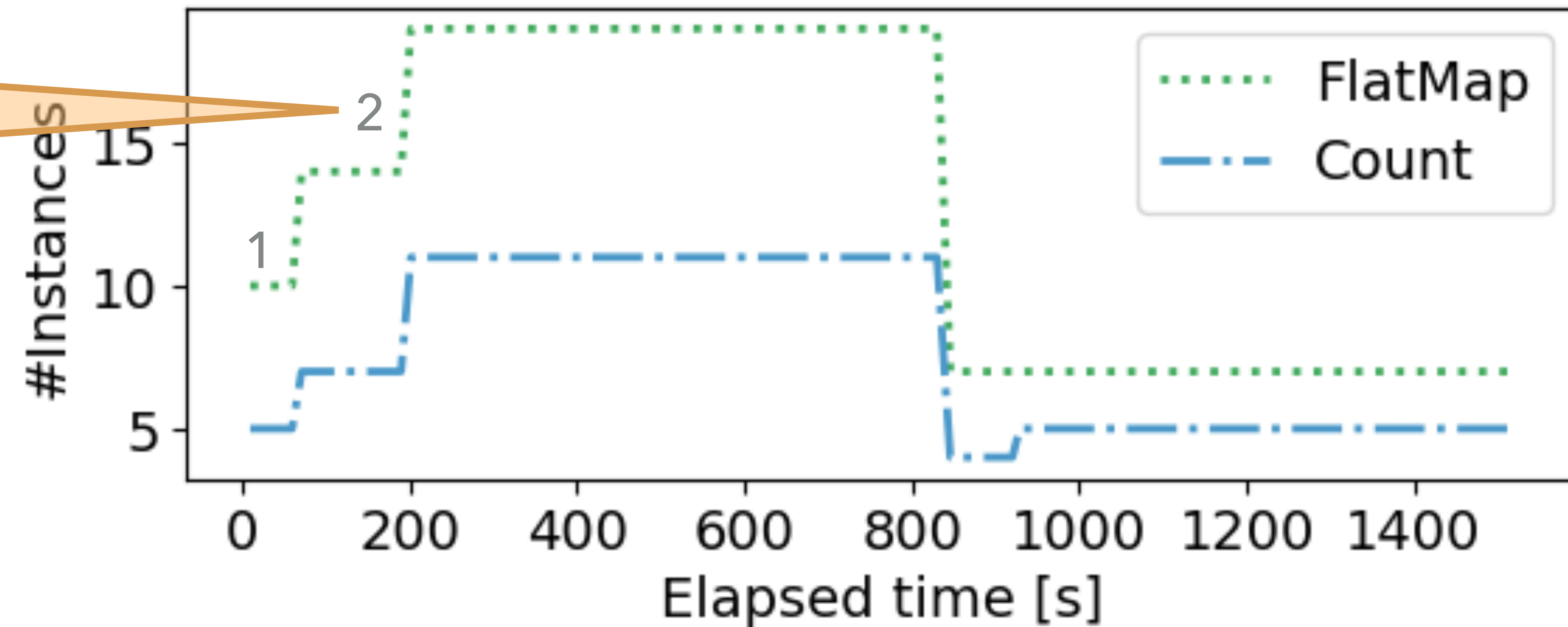




DS2 on Flink

Initially under-provisioned wordcount
Target rate: 2.000.000 rec/s, drops to half at 800s



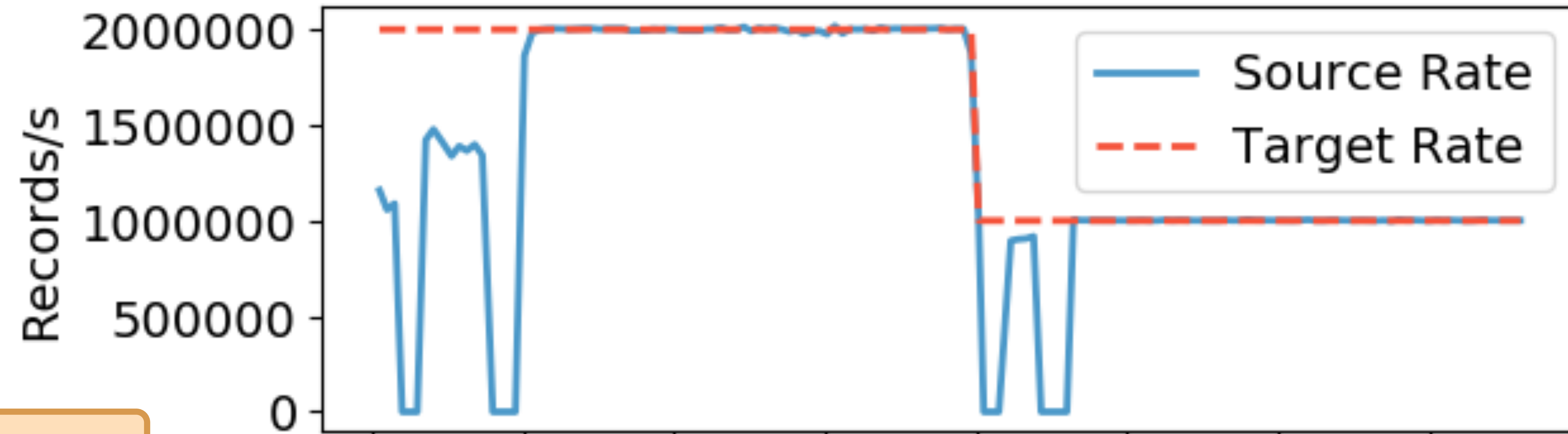


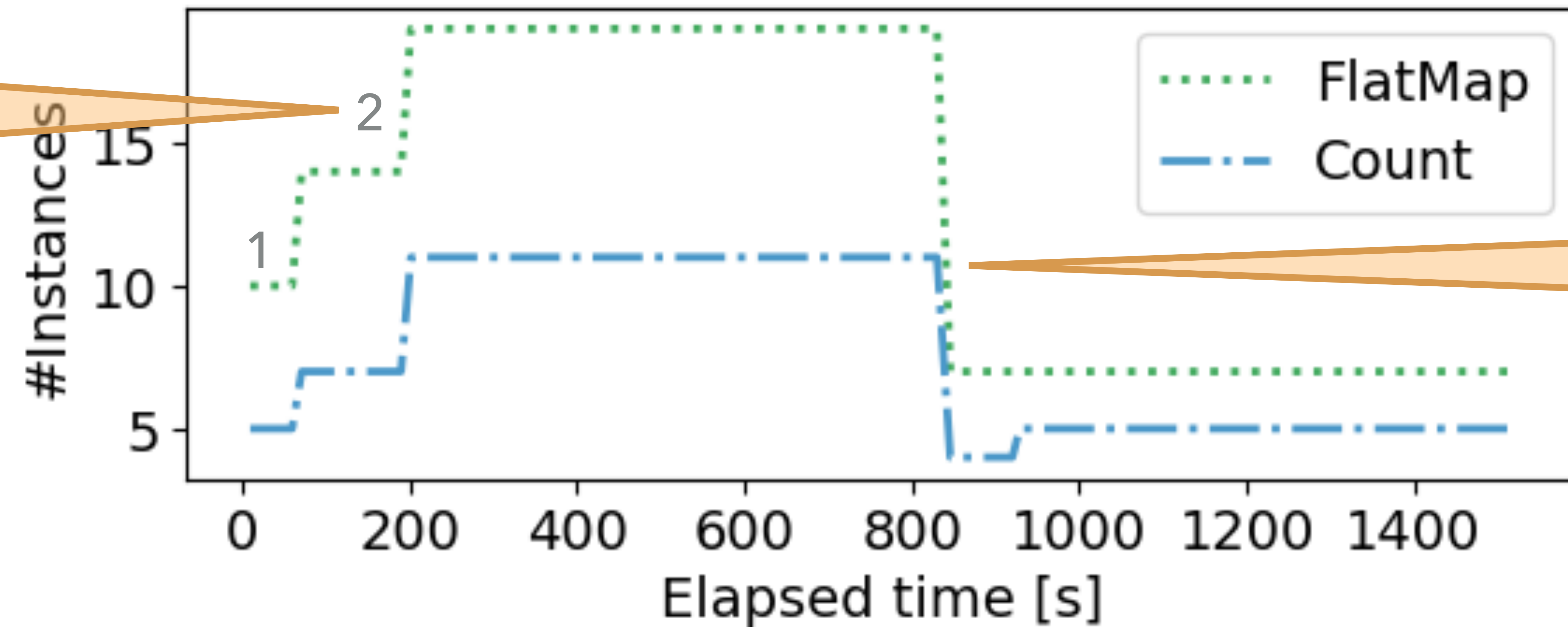
DS2 converges in **2 steps** for both operators

DS2 on Flink

Initially under-provisioned wordcount

Target rate: 2.000.000 rec/s, drops to half at 800s



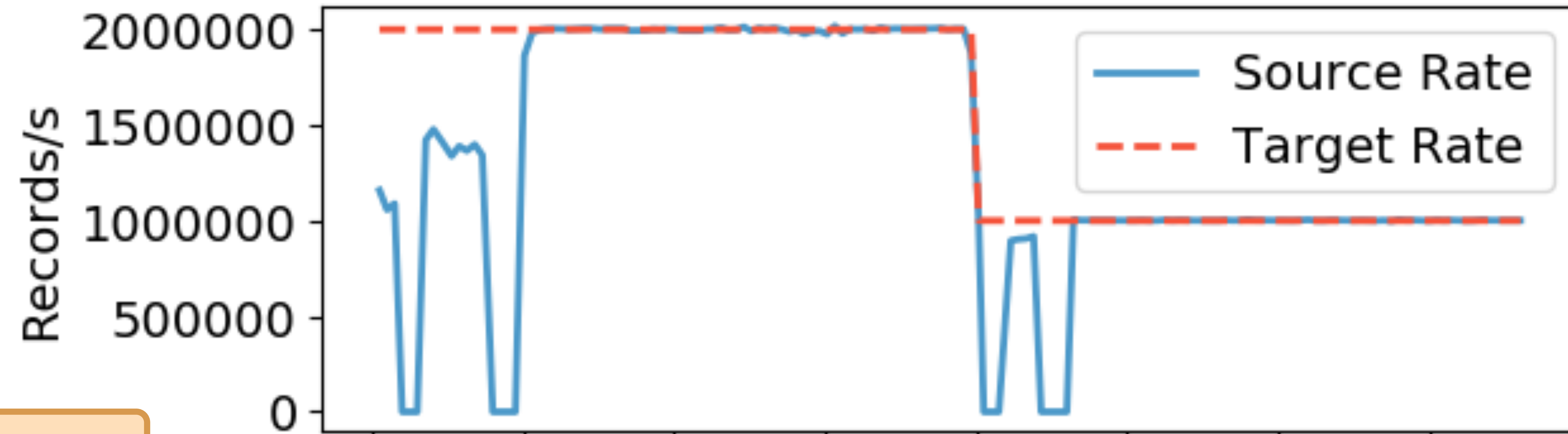


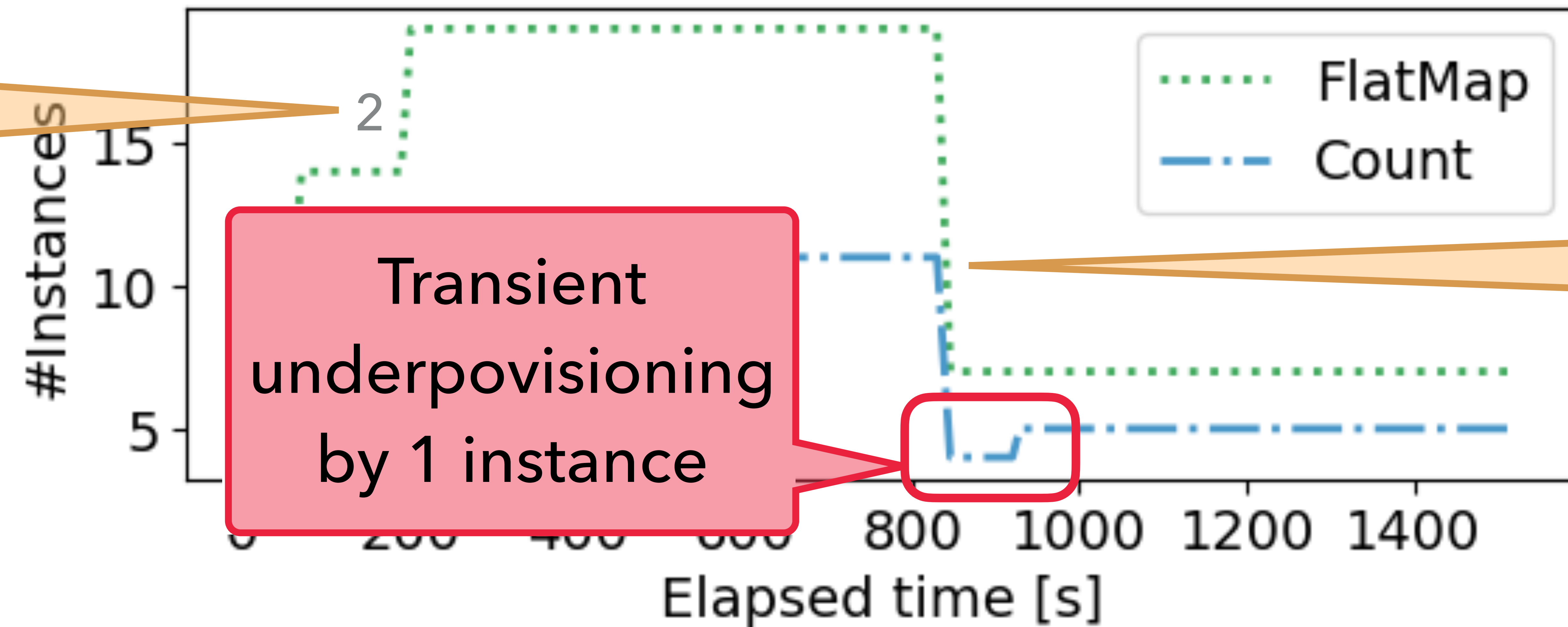
DS2 converges in **2 steps** for both operators

DS2 reacts **within 3s** when the target rate drops

DS2 on Flink

Initially under-provisioned wordcount
Target rate: 2.000.000 rec/s, drops to half at 800s





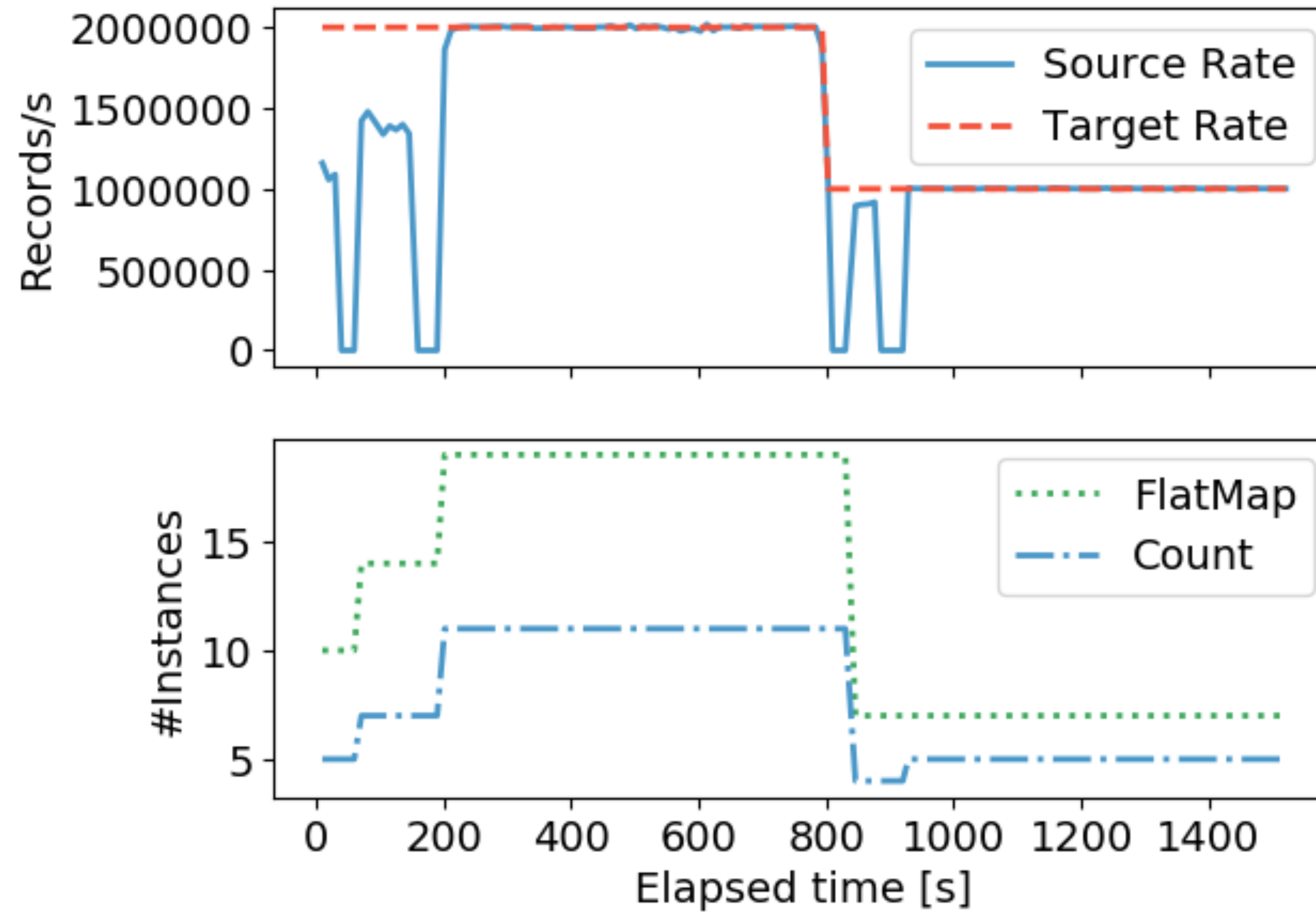
DS2 converges in **2 steps** for both operators

2

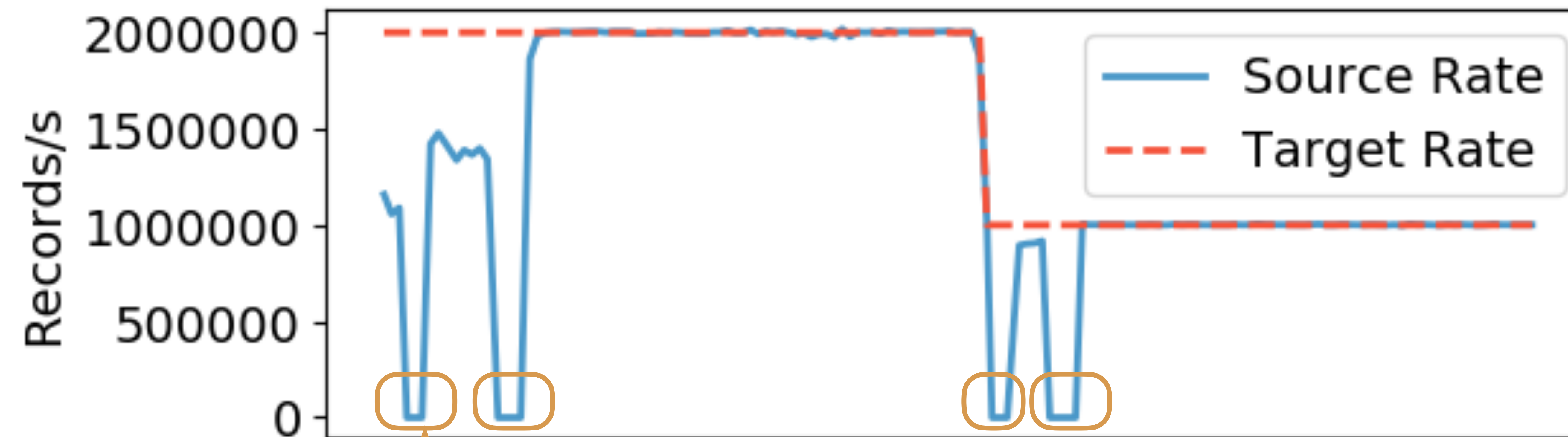
Transient underpovisioning by 1 instance

DS2 reacts **within 3s** when the target rate drops

DS2 scaling actions on Apache Flink wordcount

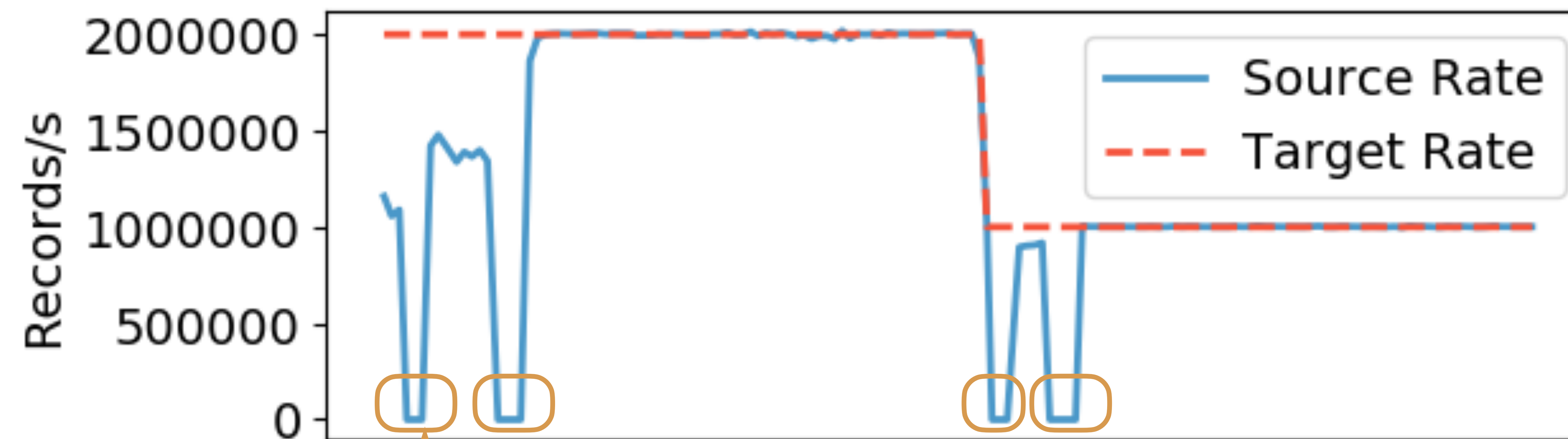


DS2 scaling actions on Apache Flink wordcount



Every reconfiguration takes ~**30s** during which the system is **unavailable**

DS2 scaling actions on Apache Flink wordcount



Every reconfiguration takes ~**30s** during which the system is **unavailable**

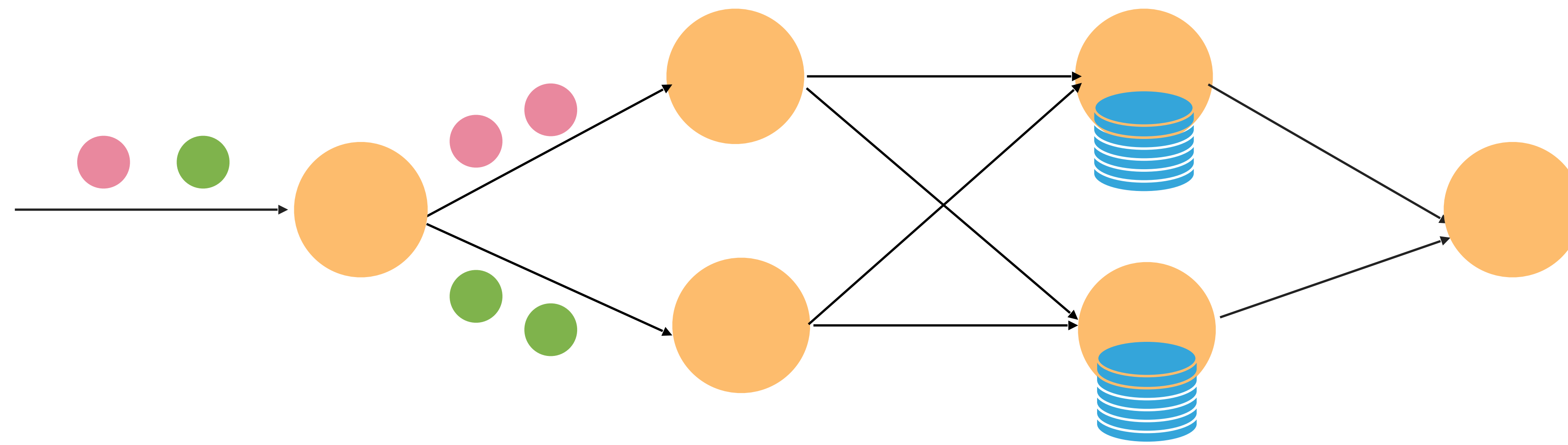
Re-configuration requires **state migration** with correctness guarantees.

State migration

State migration strategies

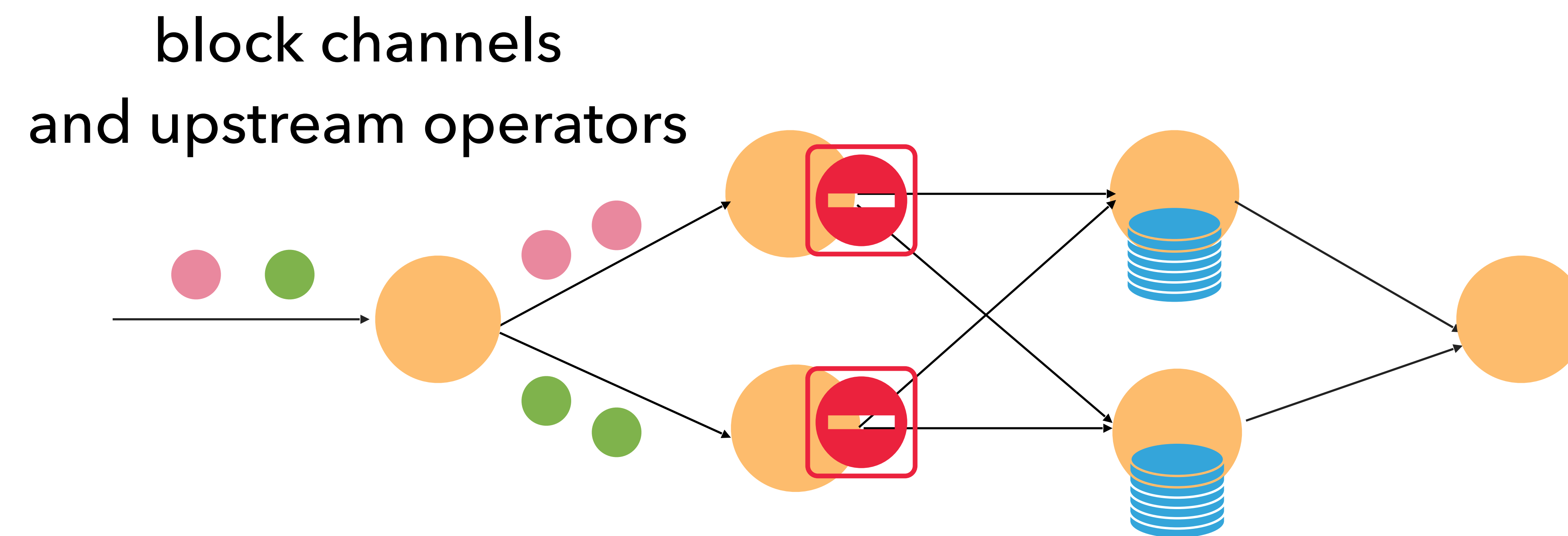
- Stop-and-restart
  - halt the whole computation, take a state snapshot of all operators, restart
  - unnecessary stalls if only one or few operators need to be rescaled
- Partial pause and restart
  - only temporarily block the affected dataflow subgraph
  - usually the operator to be scaled and upstream channels
- All-at-once
  - move state to be migrated in one operation
  - high latency during migration if the state is large
- Progressive
  - move state to be migrated in smaller pieces, e.g. key-by-key
  - can be used to interleave state transfer with processing
  - migration duration might increase

Pause-and-restart state migration



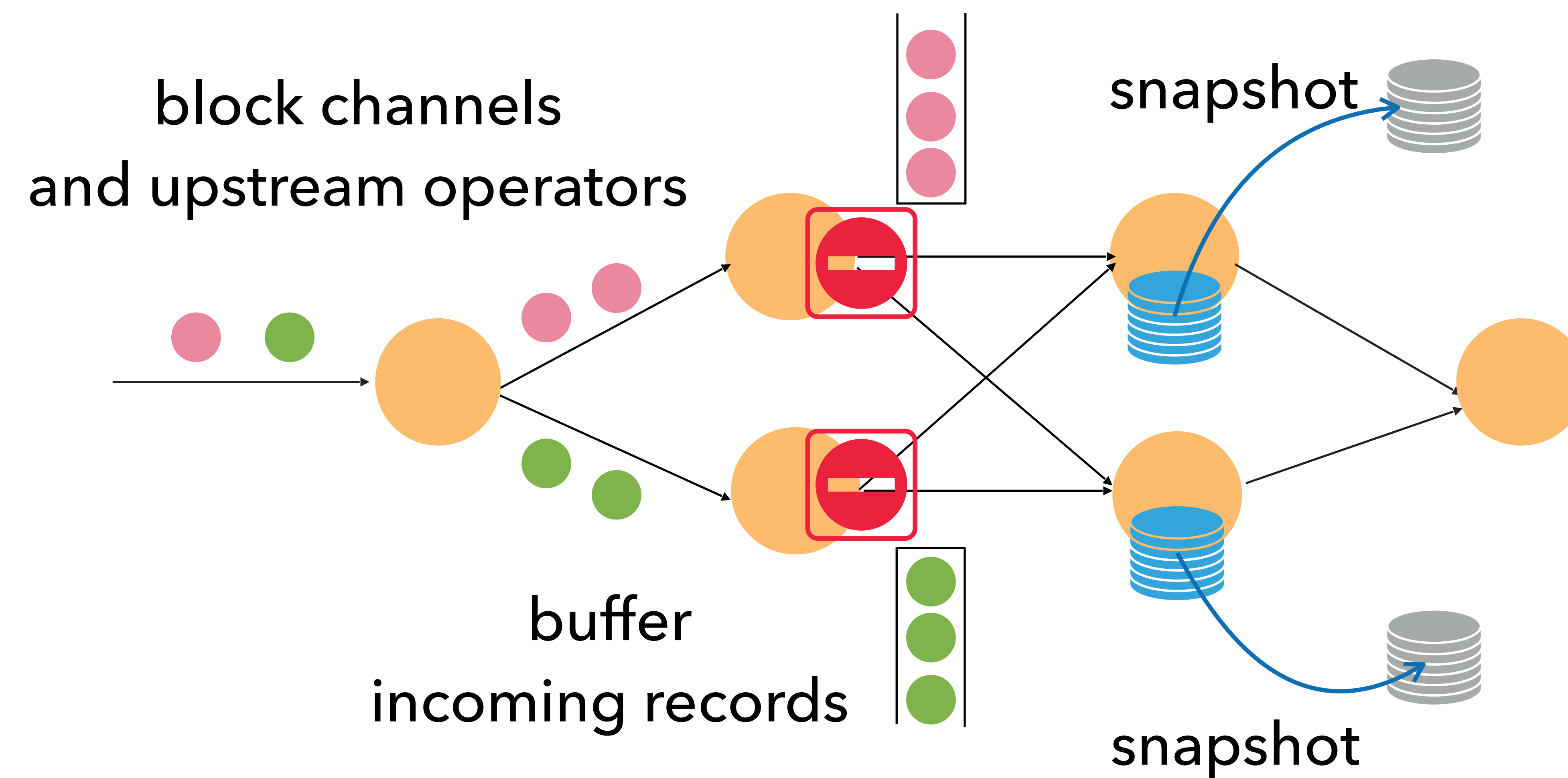
- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

Pause-and-restart state migration



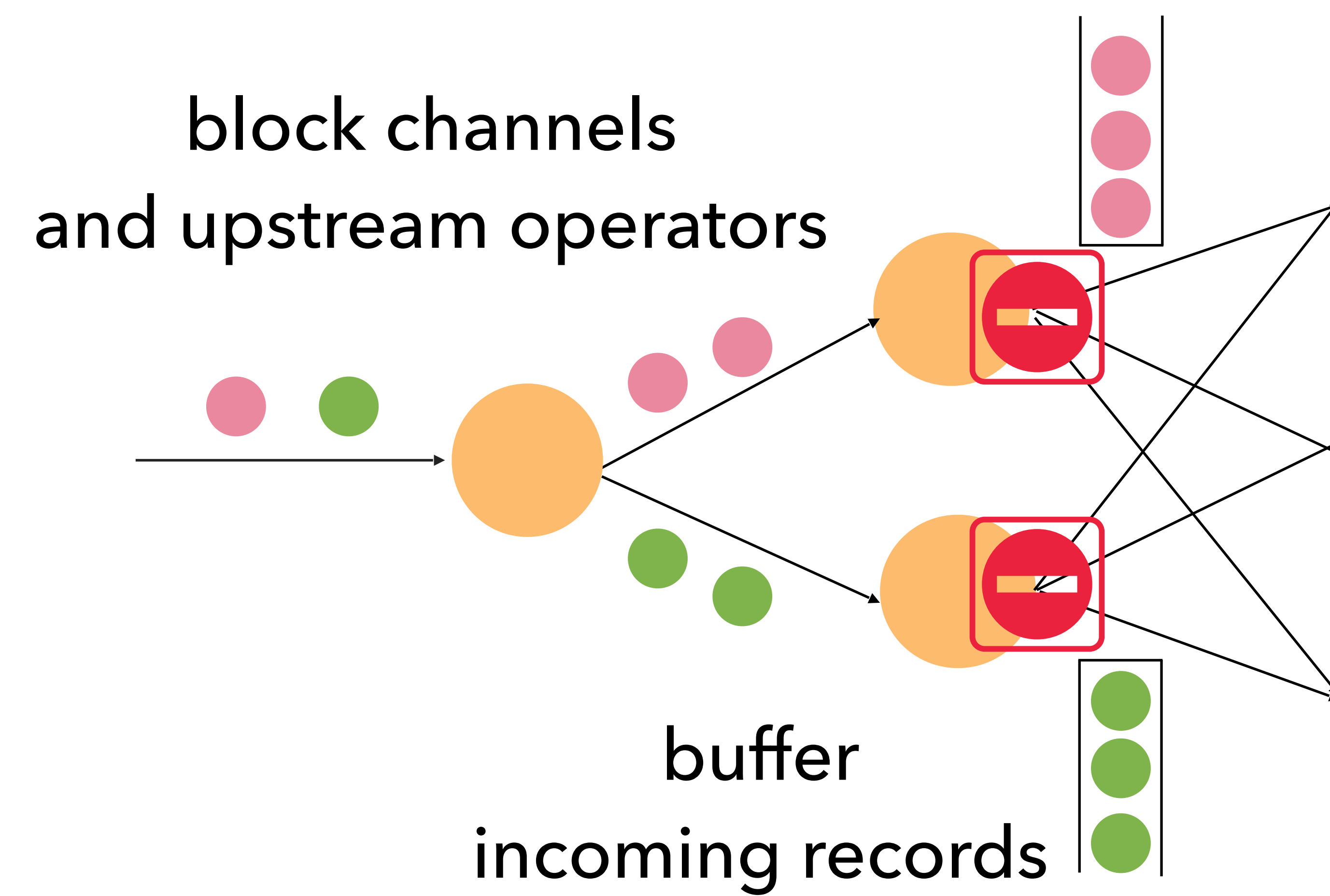
- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

Pause-and-restart state migration



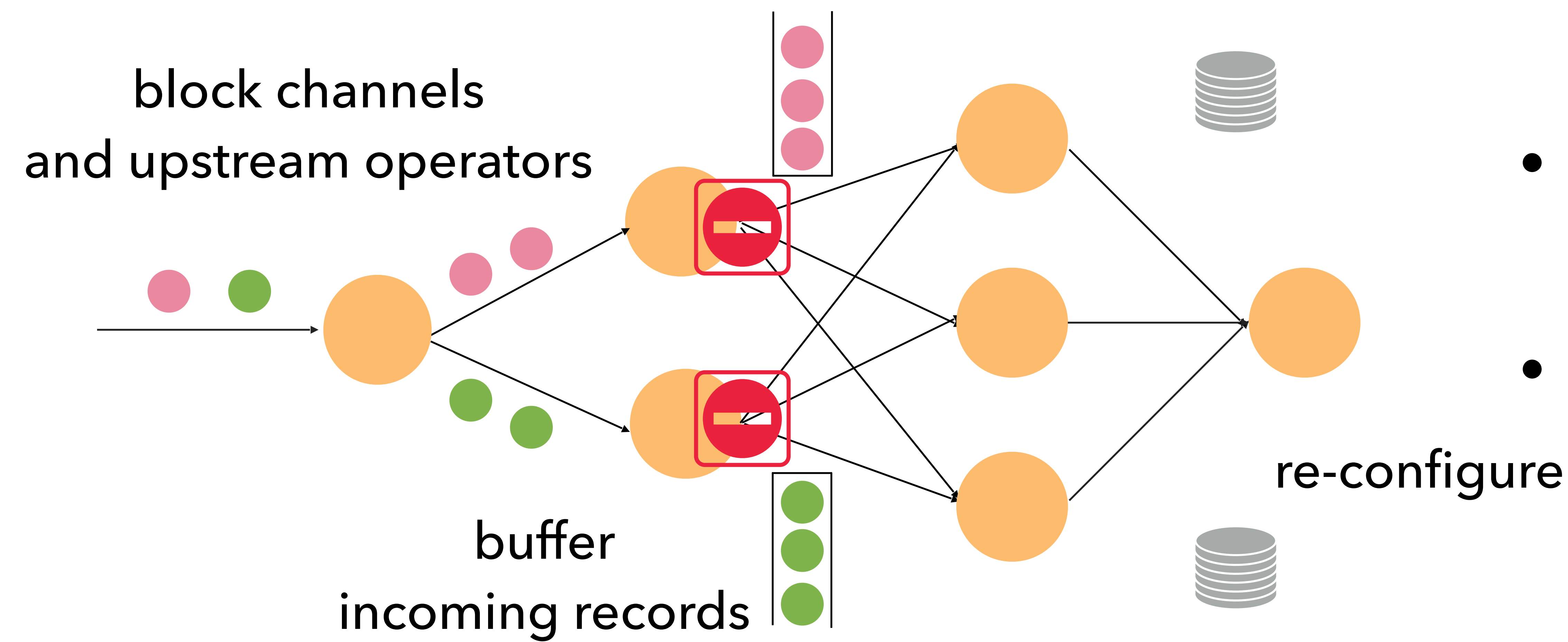
- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

Pause-and-restart state migration



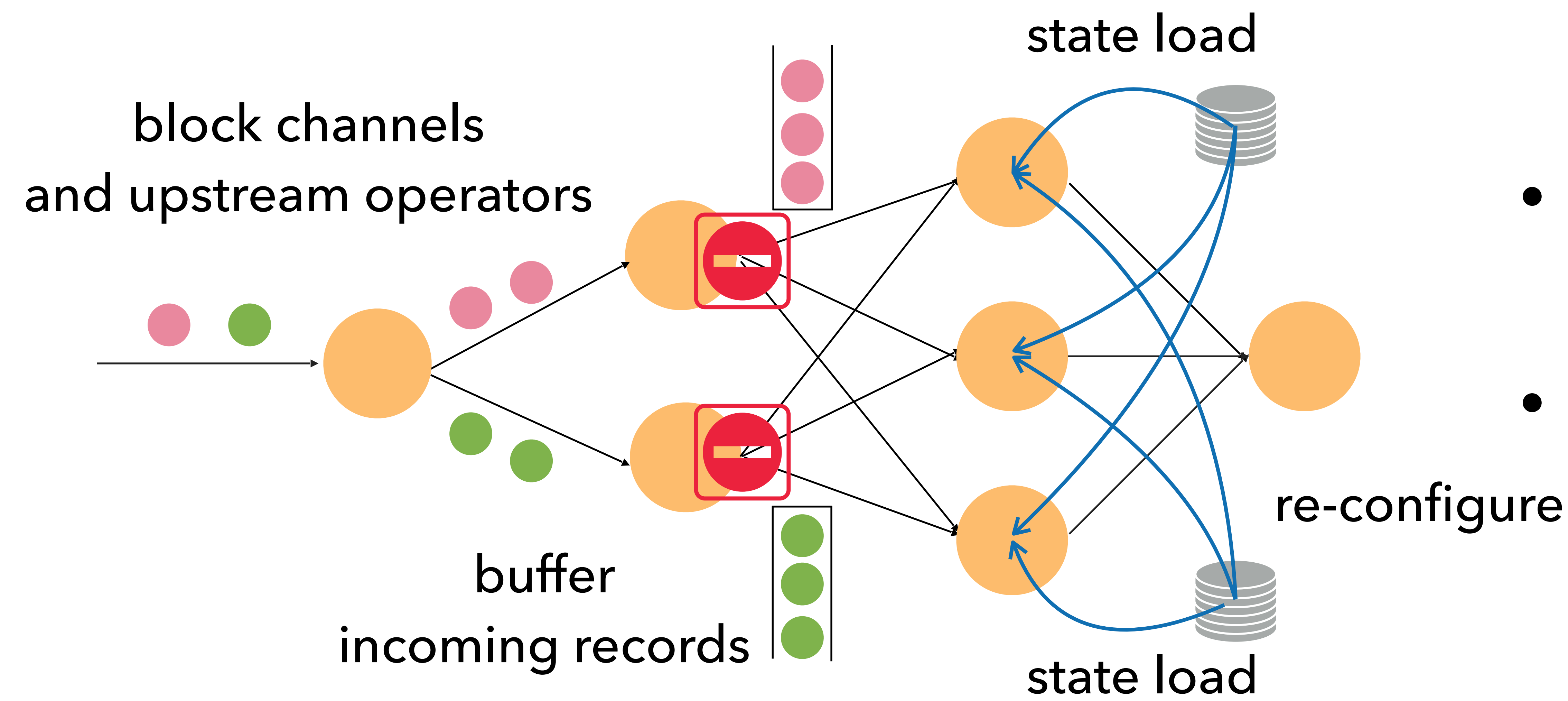
- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

Pause-and-restart state migration



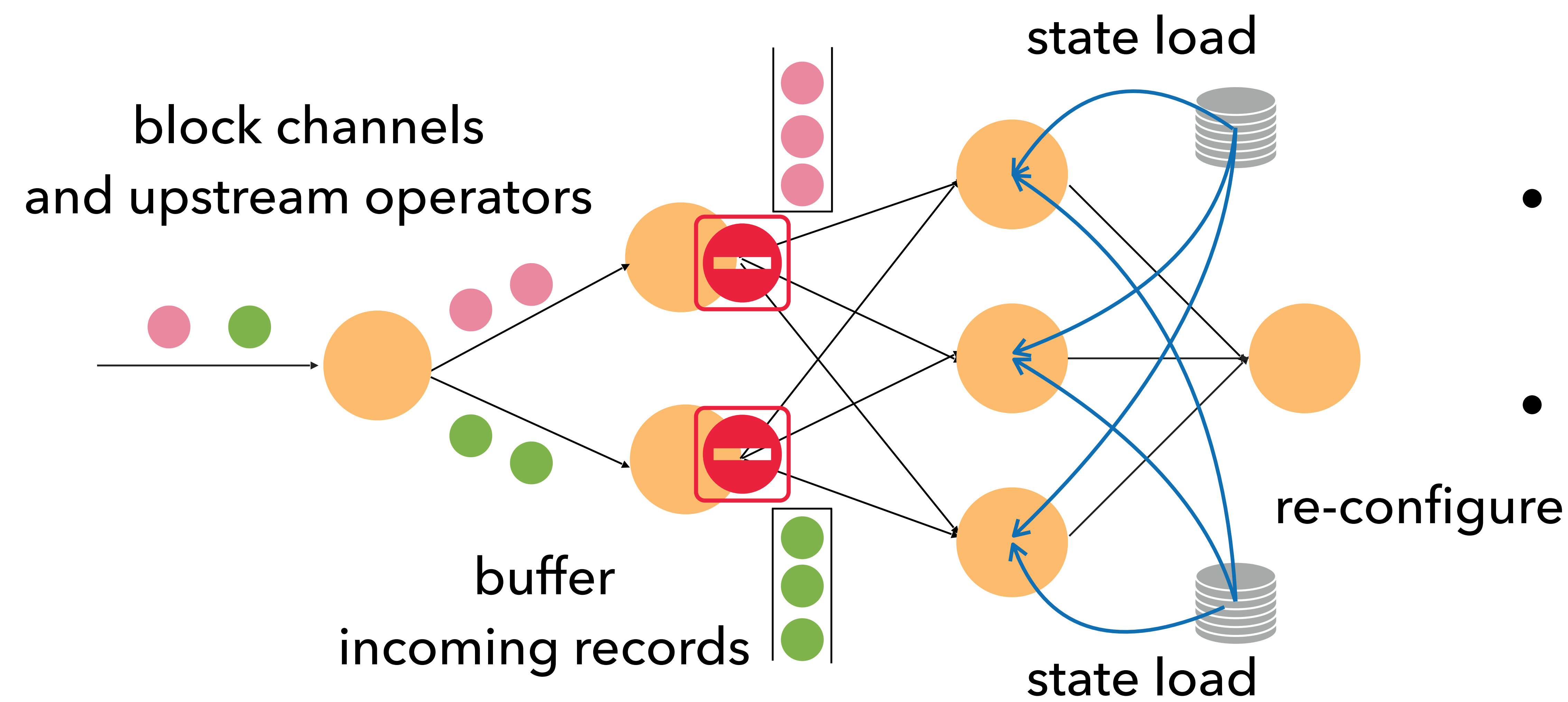
- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

Pause-and-restart state migration



- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

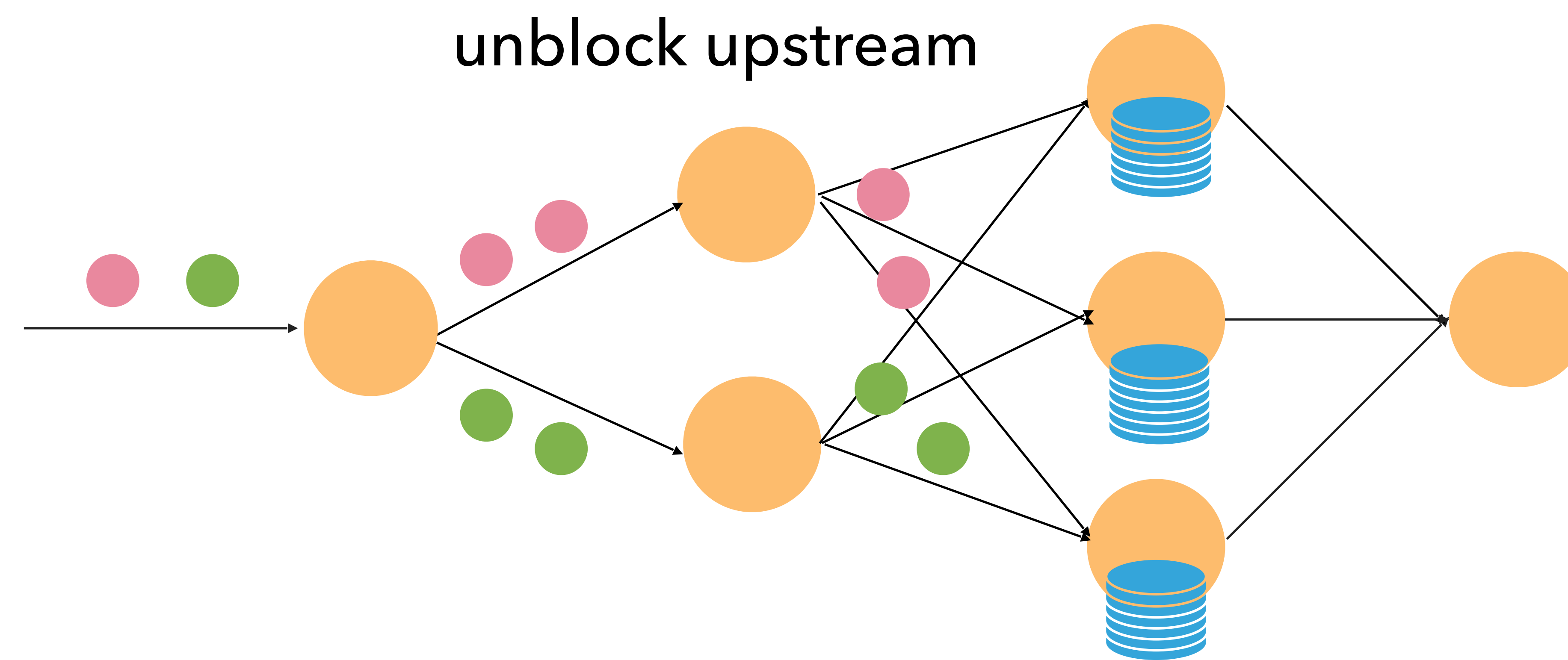
Pause-and-restart state migration



- State is **scoped** to a single task
- Each stateful task is responsible for **processing** *and* **state management**

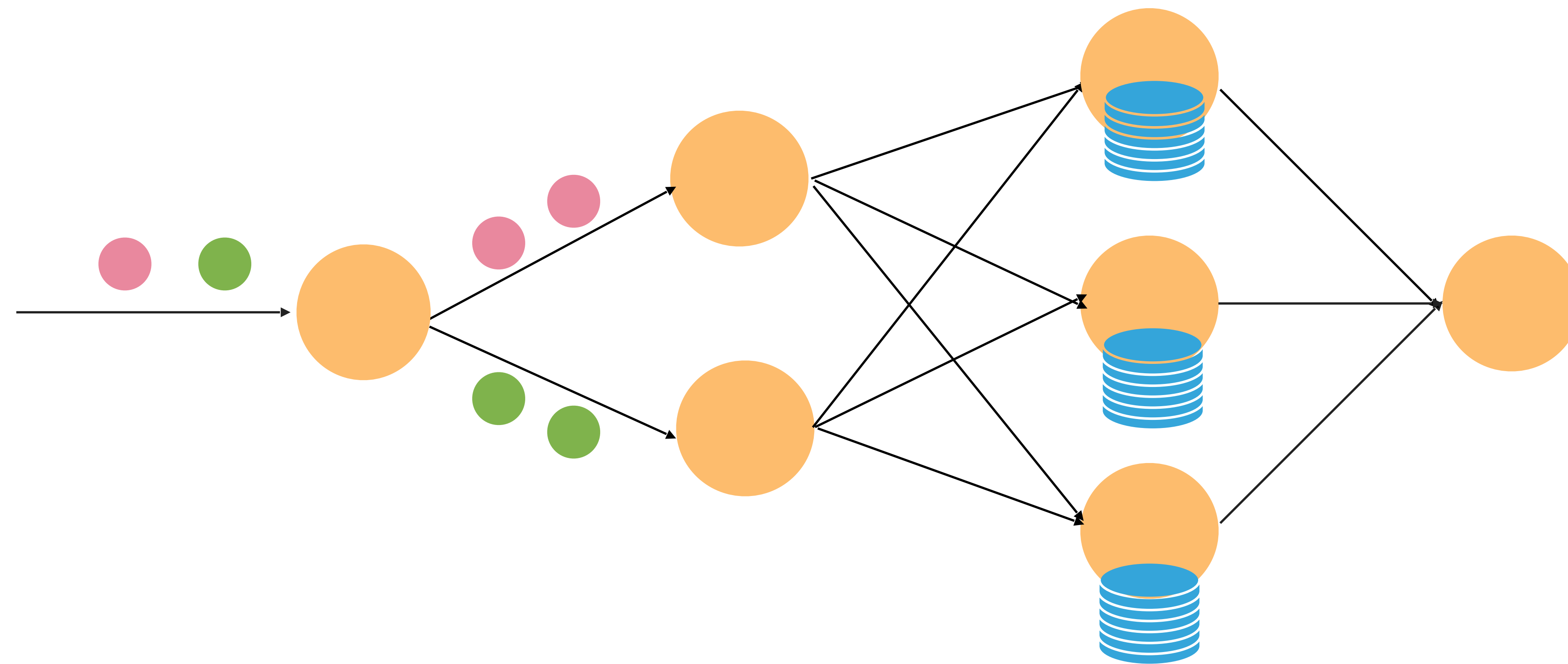
All affected operators **block** until the reconfiguration is complete

Pause-and-restart state migration

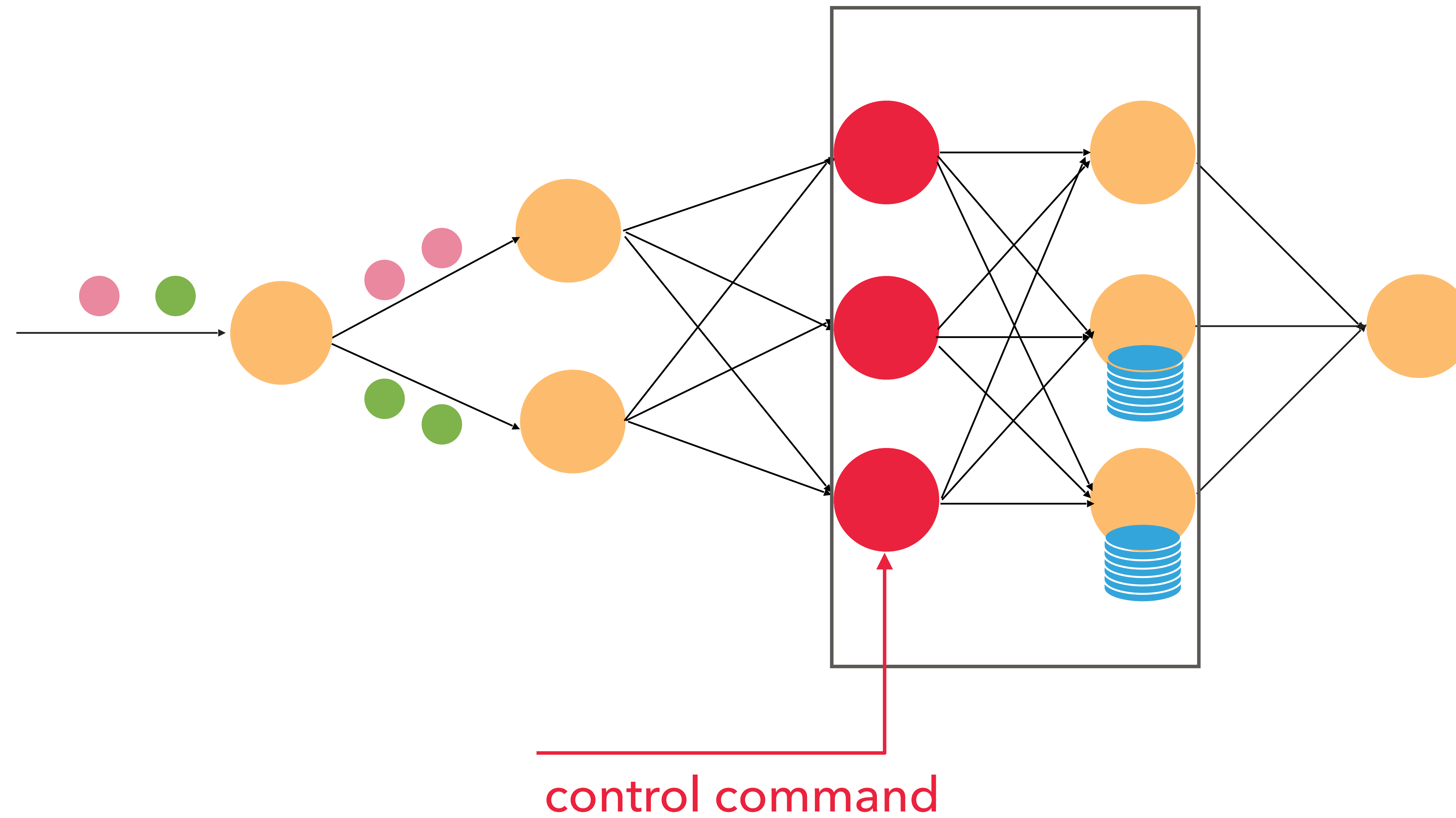


Live state migration

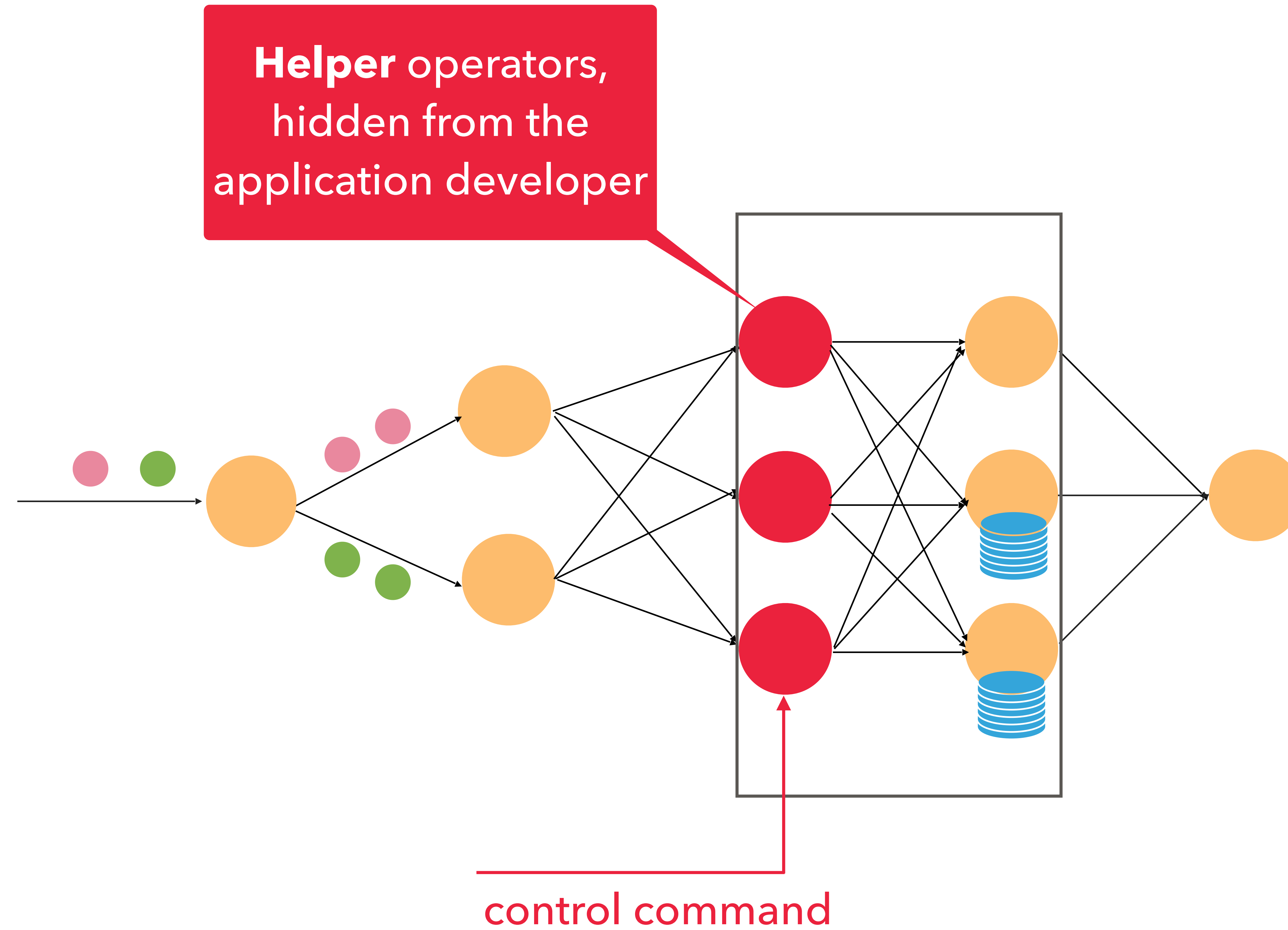
Intuition: treat state migration as **a dataflow operation** and *interleave* **fine-grained** state transfers with processing.



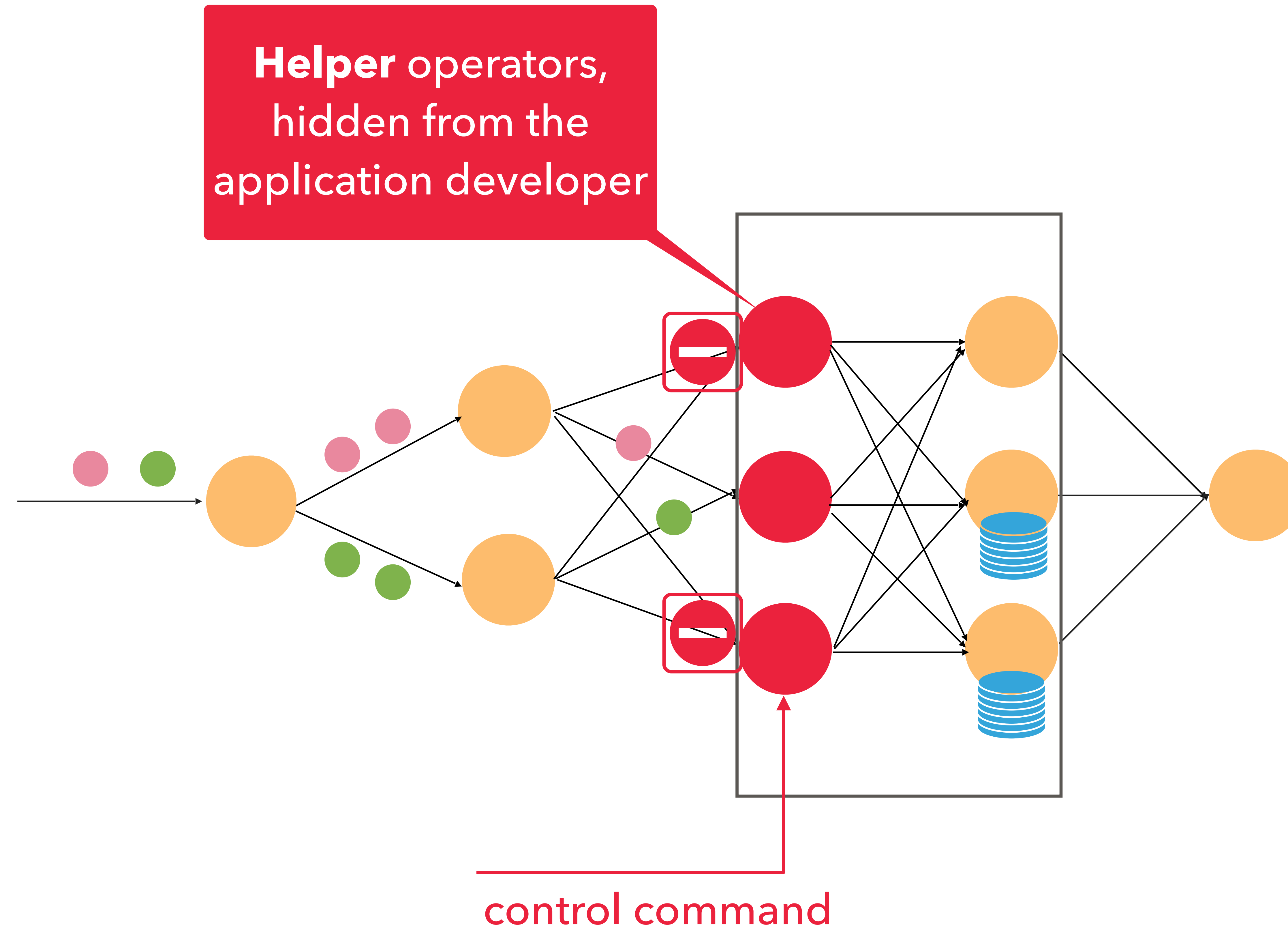
Live state migration



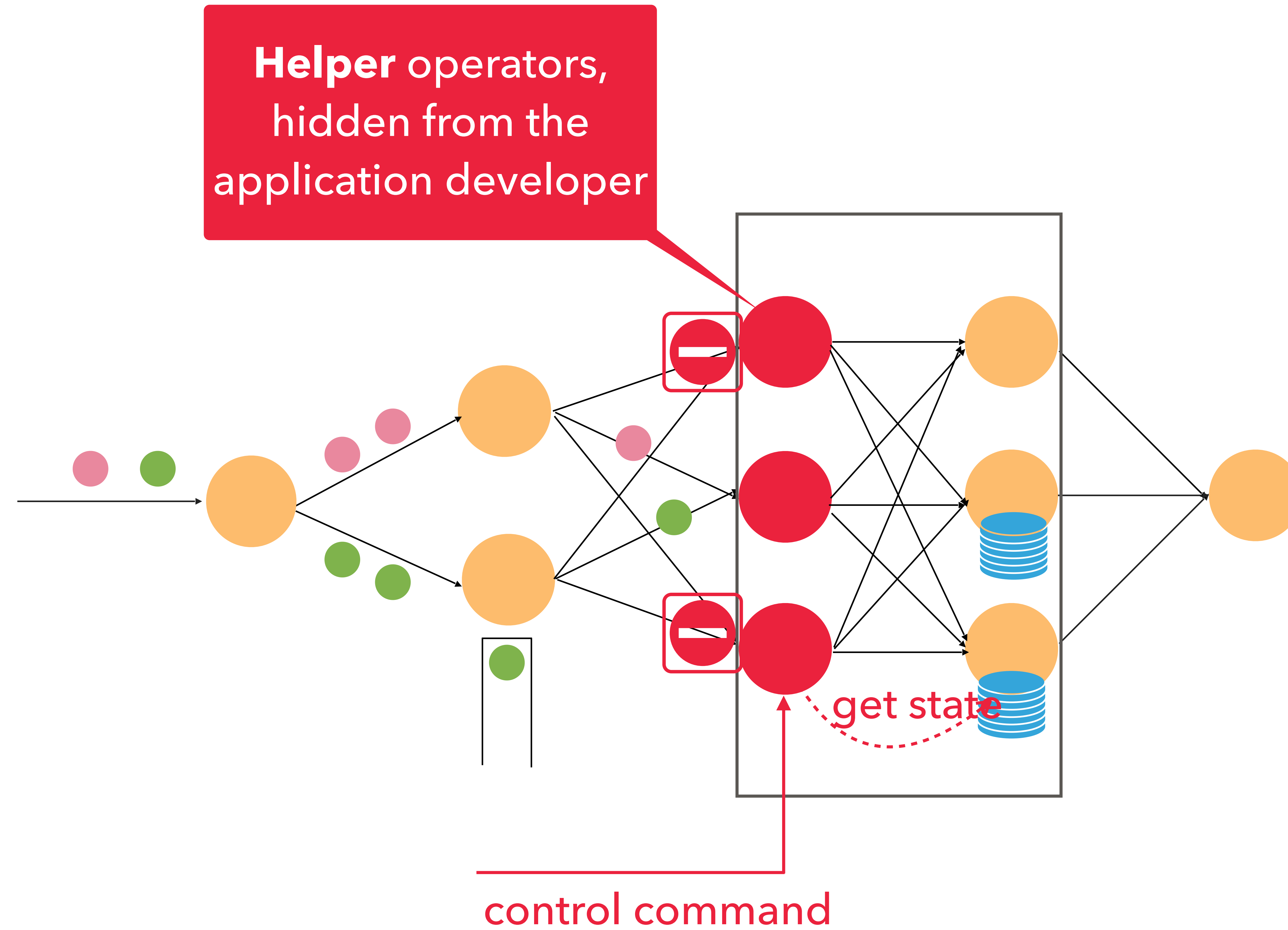
Live state migration



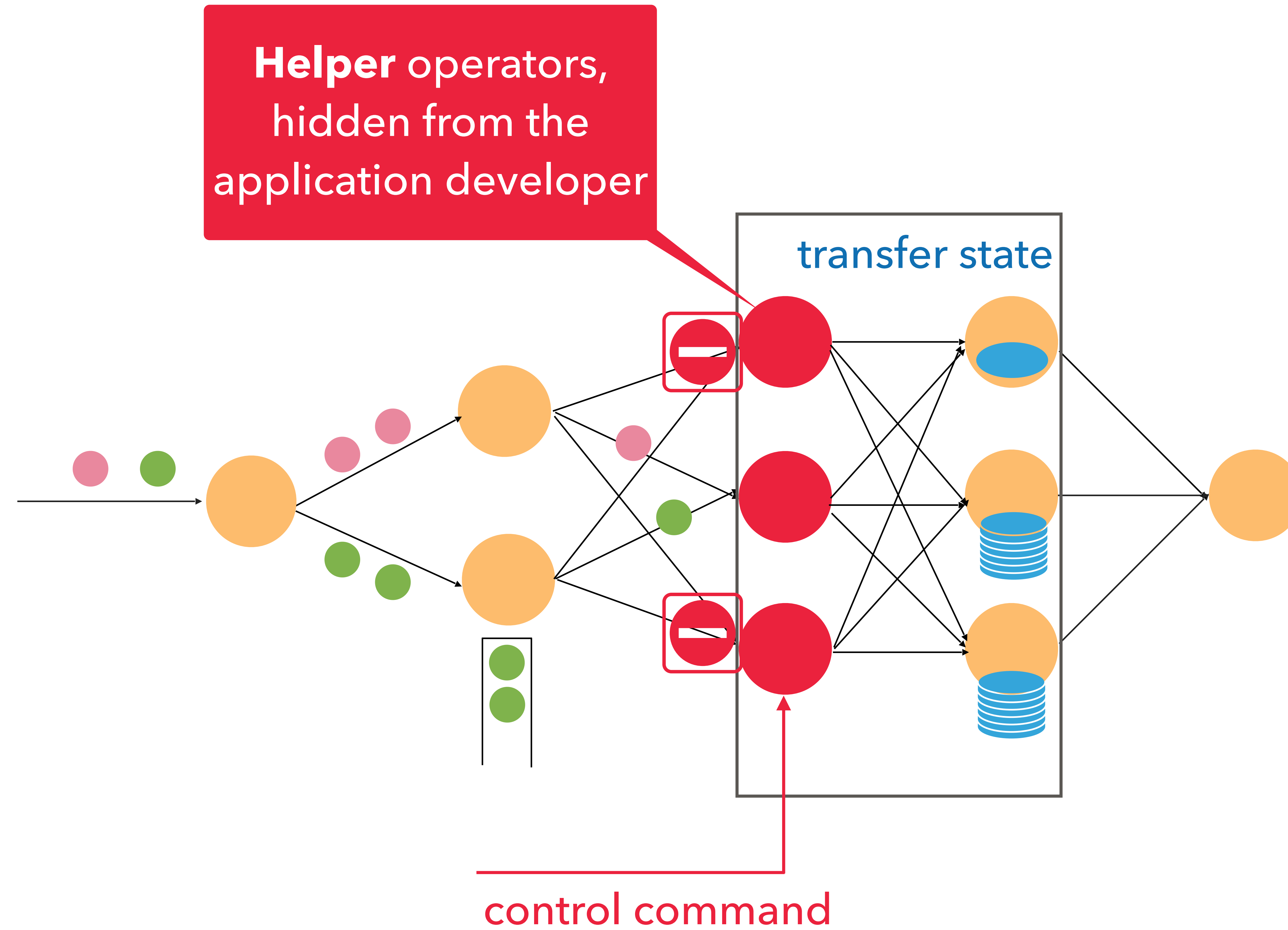
Live state migration



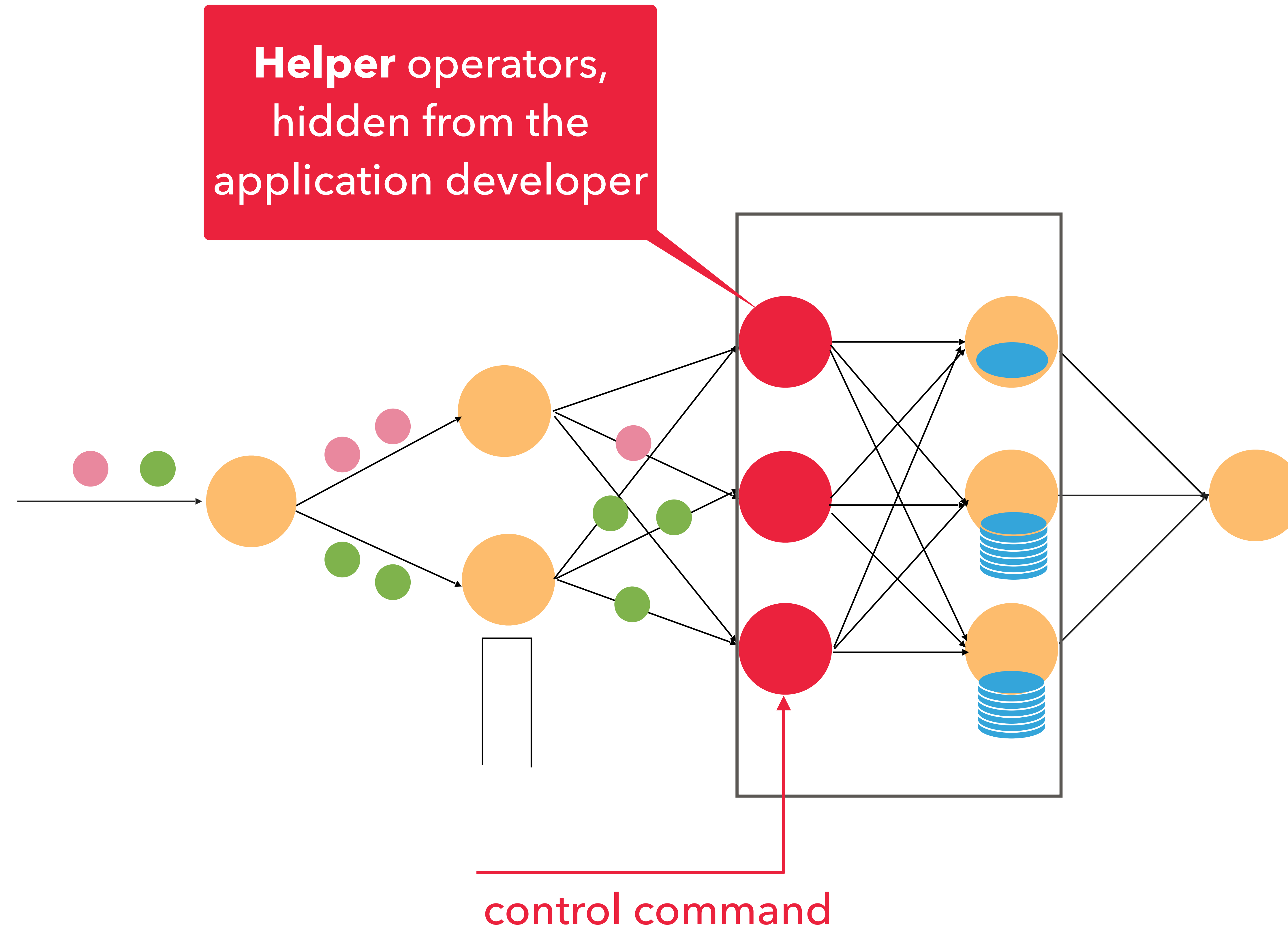
Live state migration



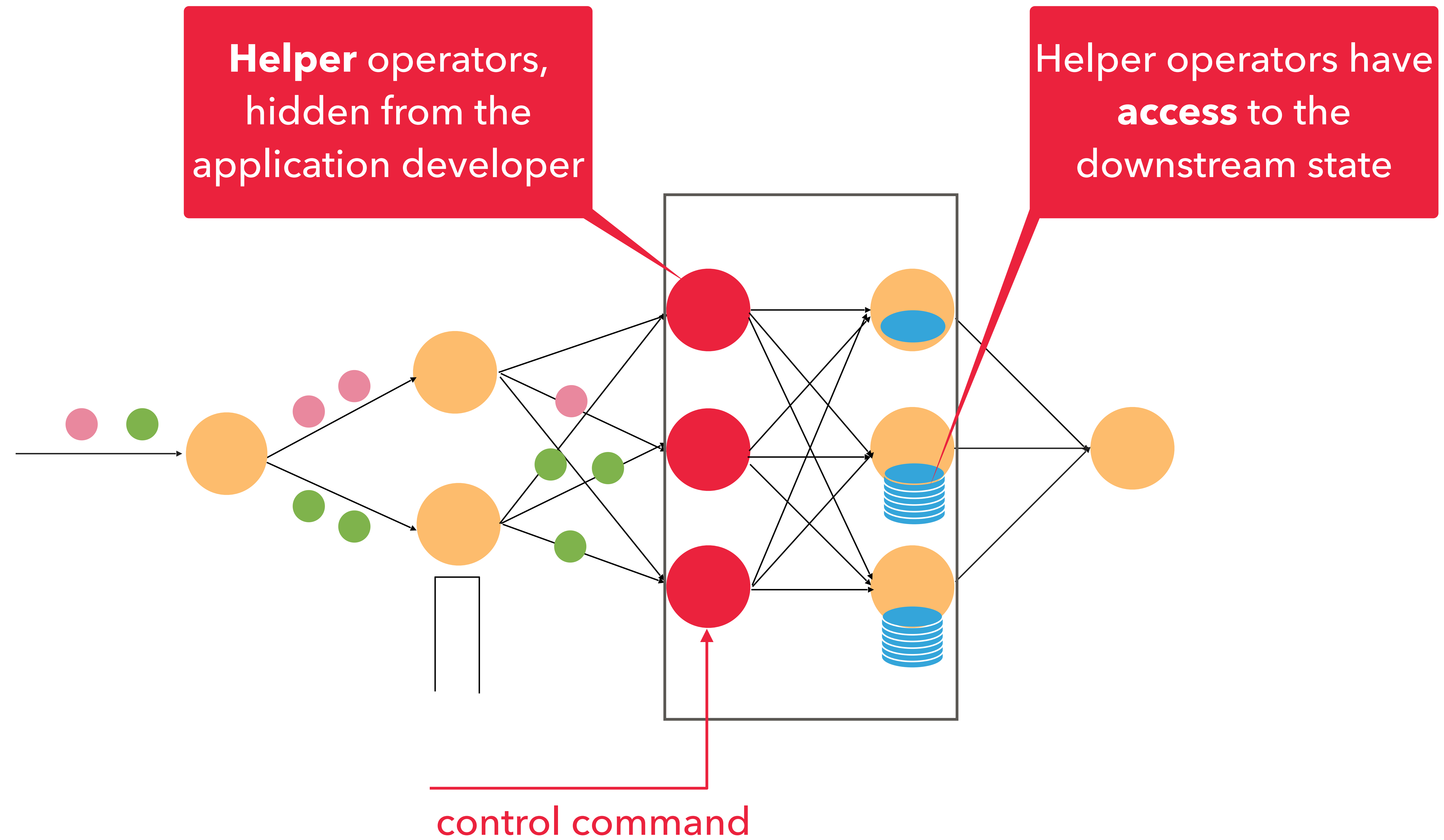
Live state migration



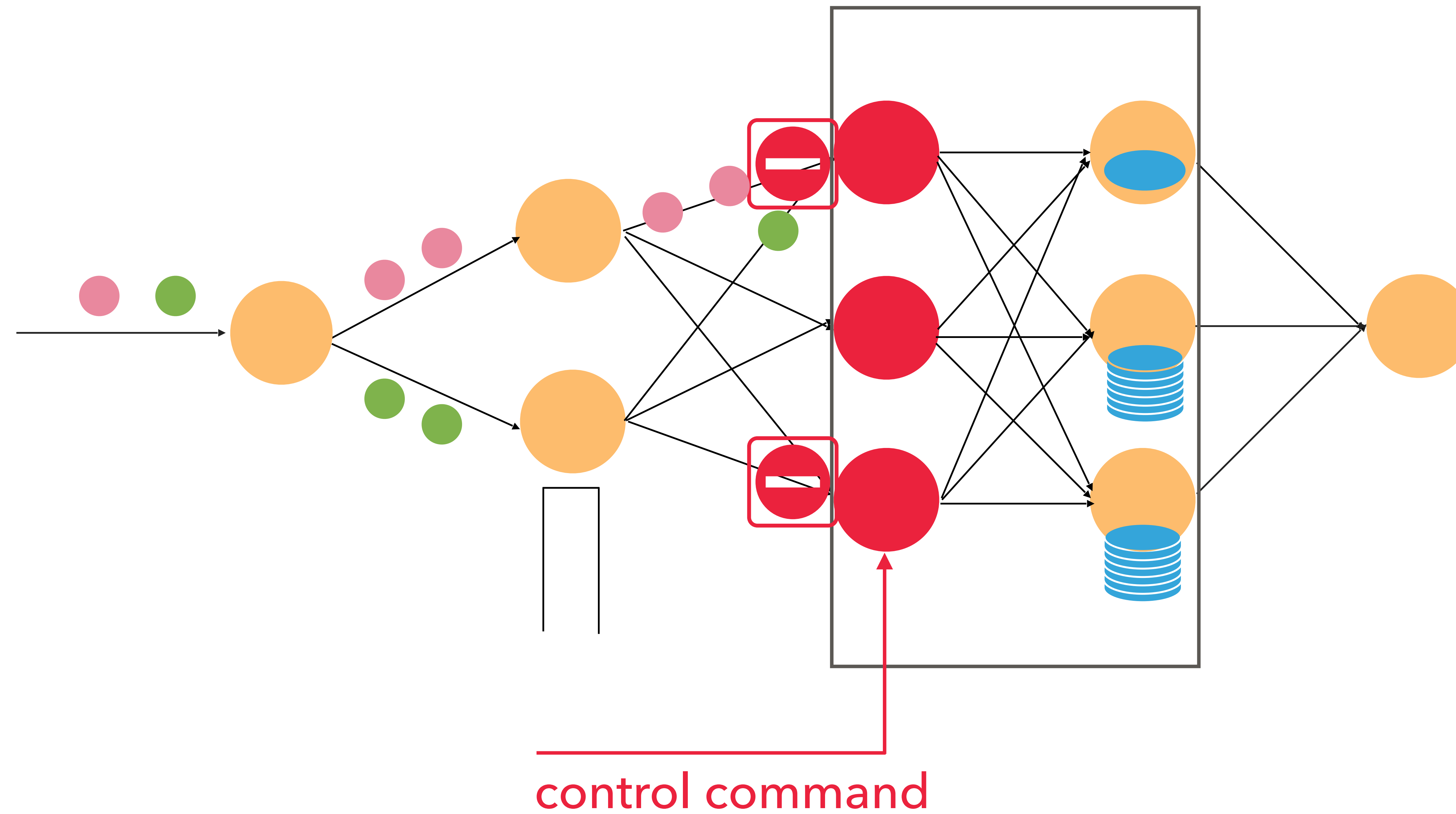
Live state migration



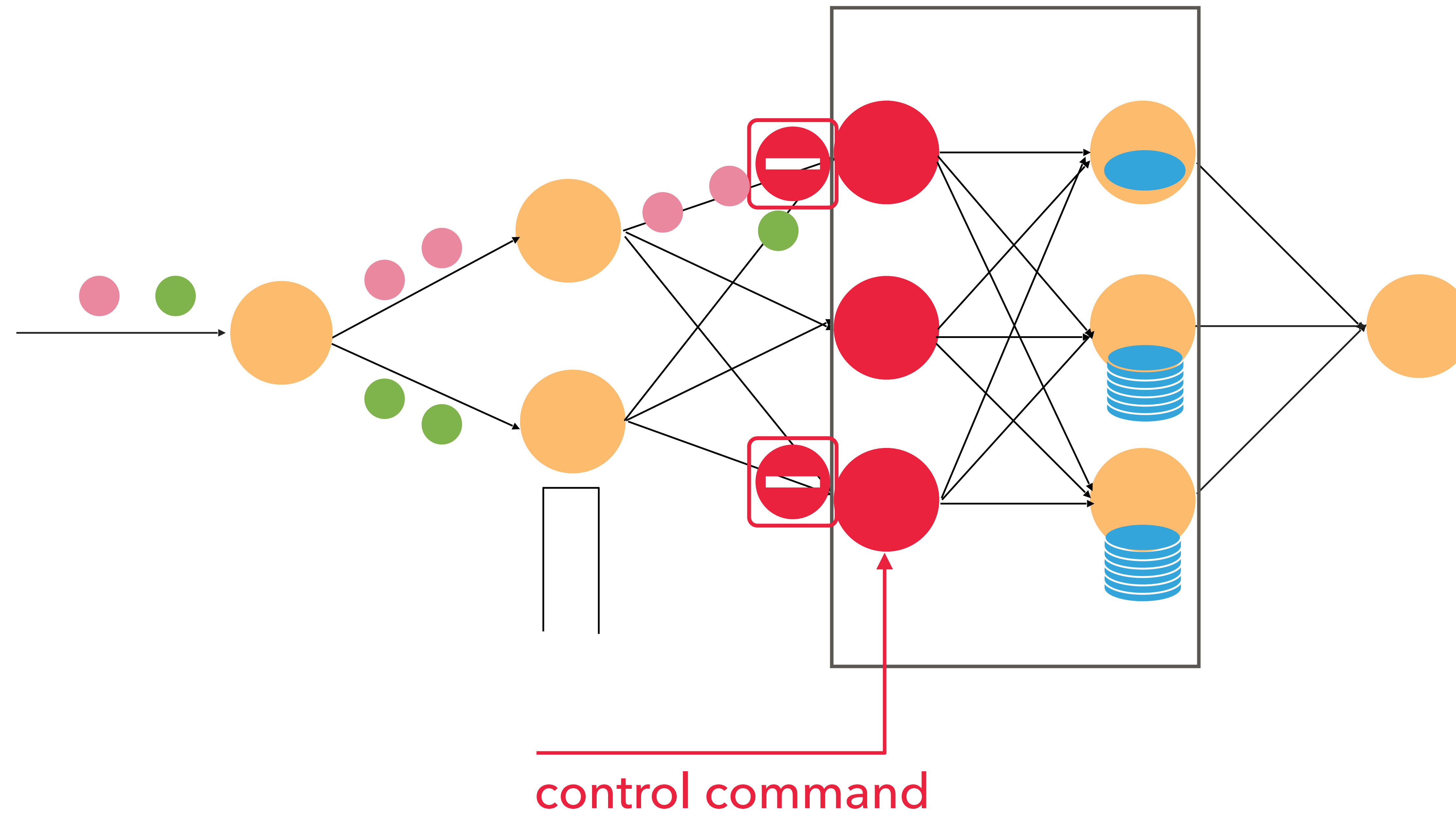
Live state migration



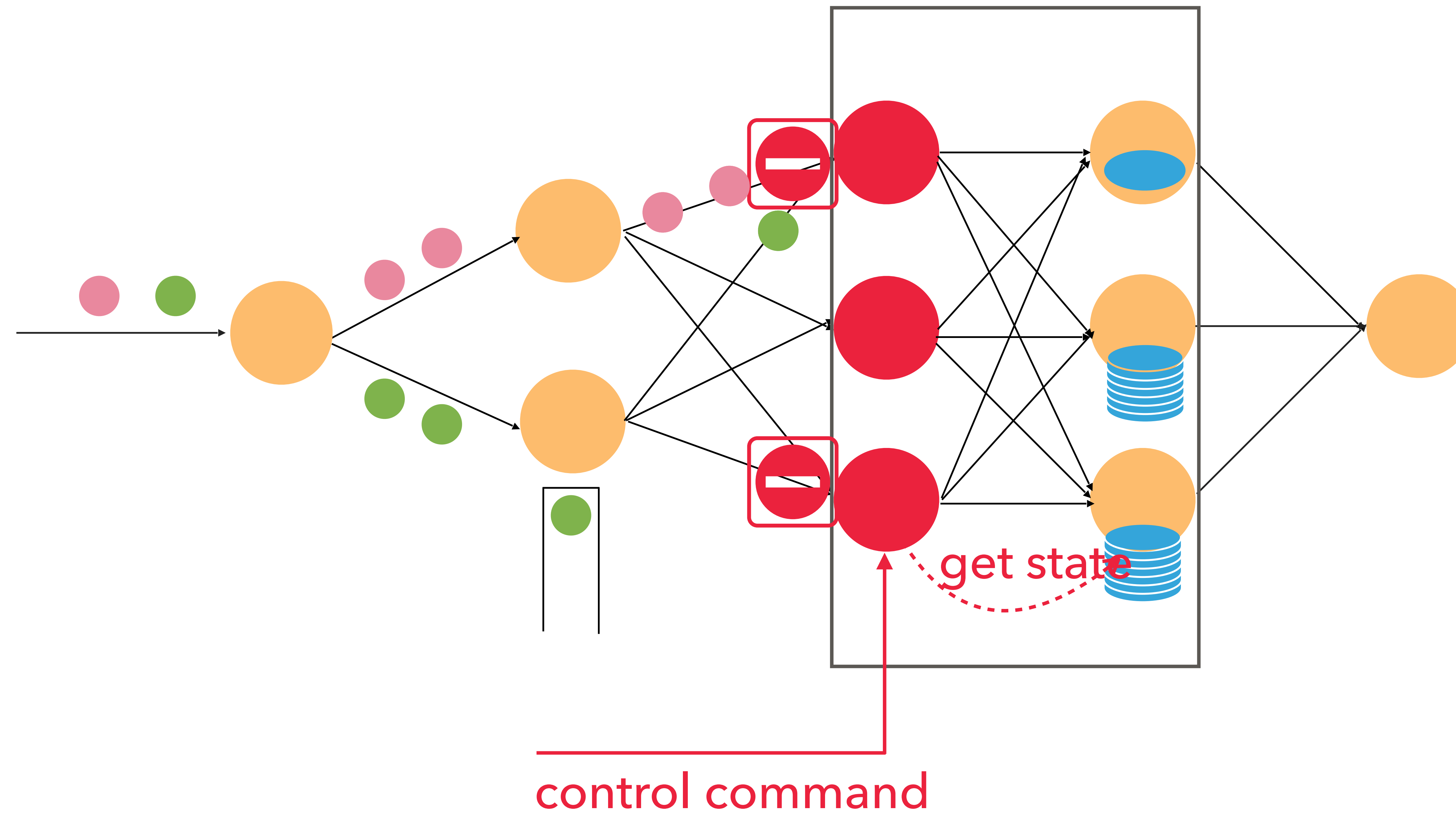
Live state migration



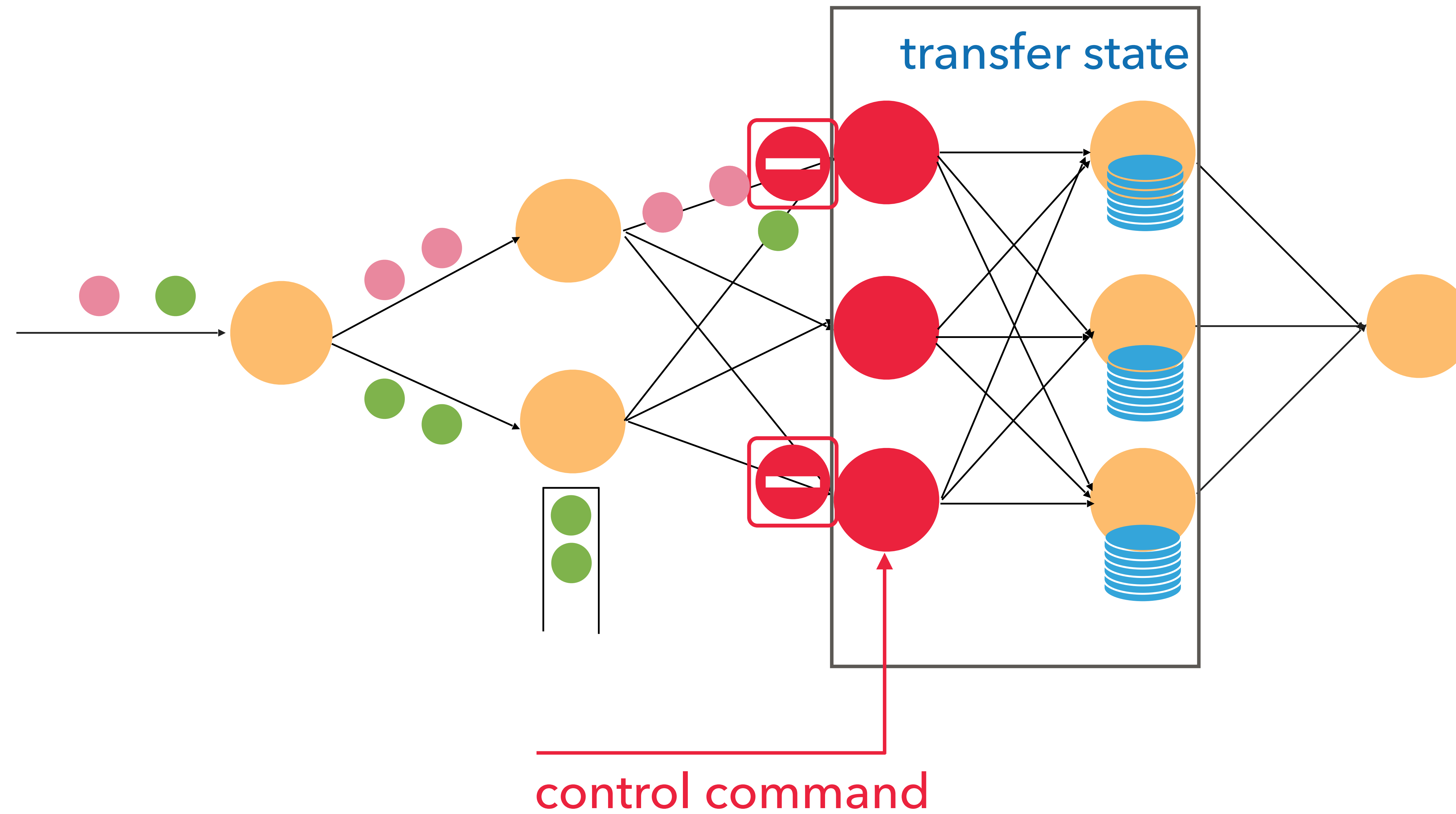
Live state migration



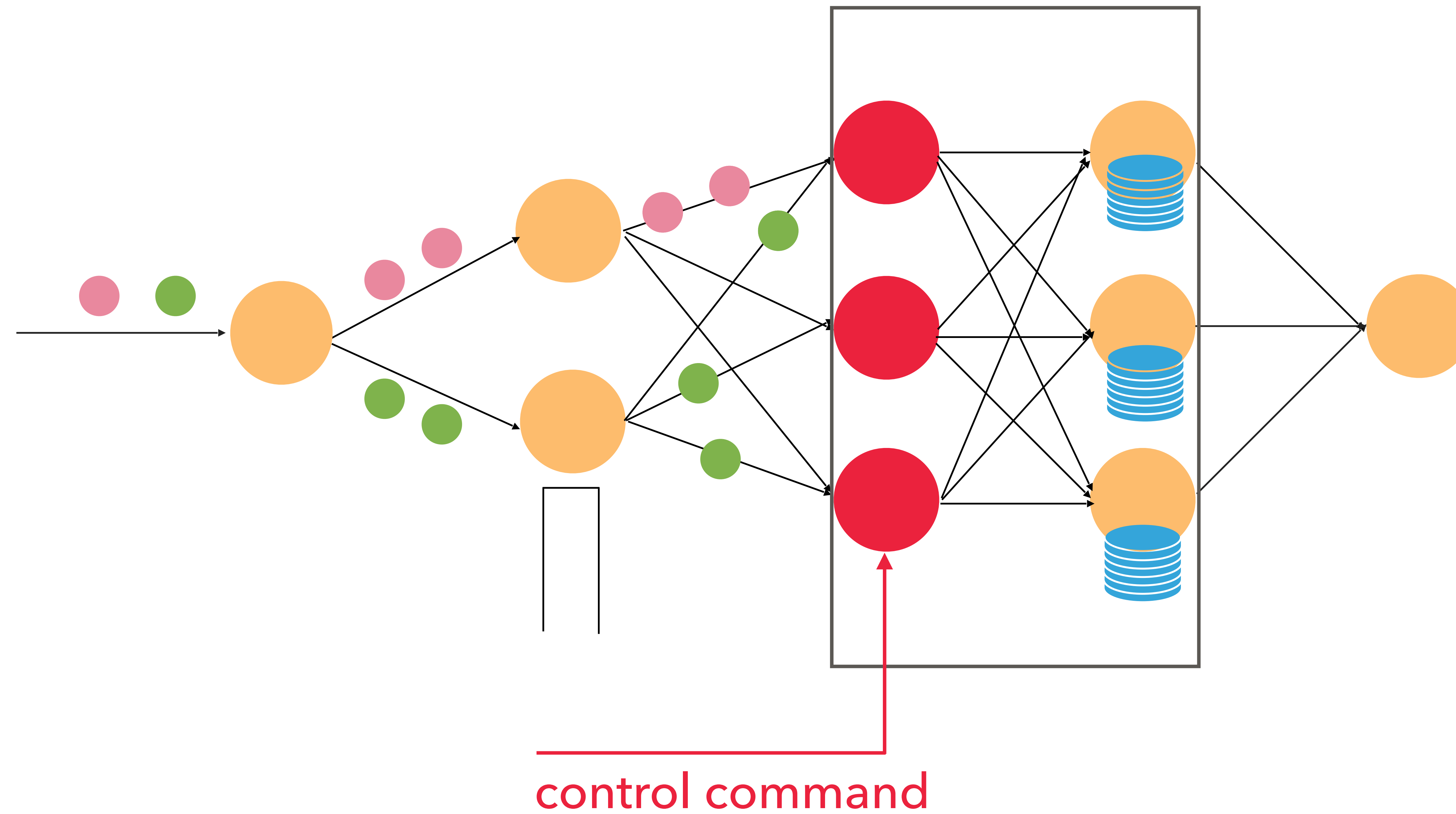
Live state migration



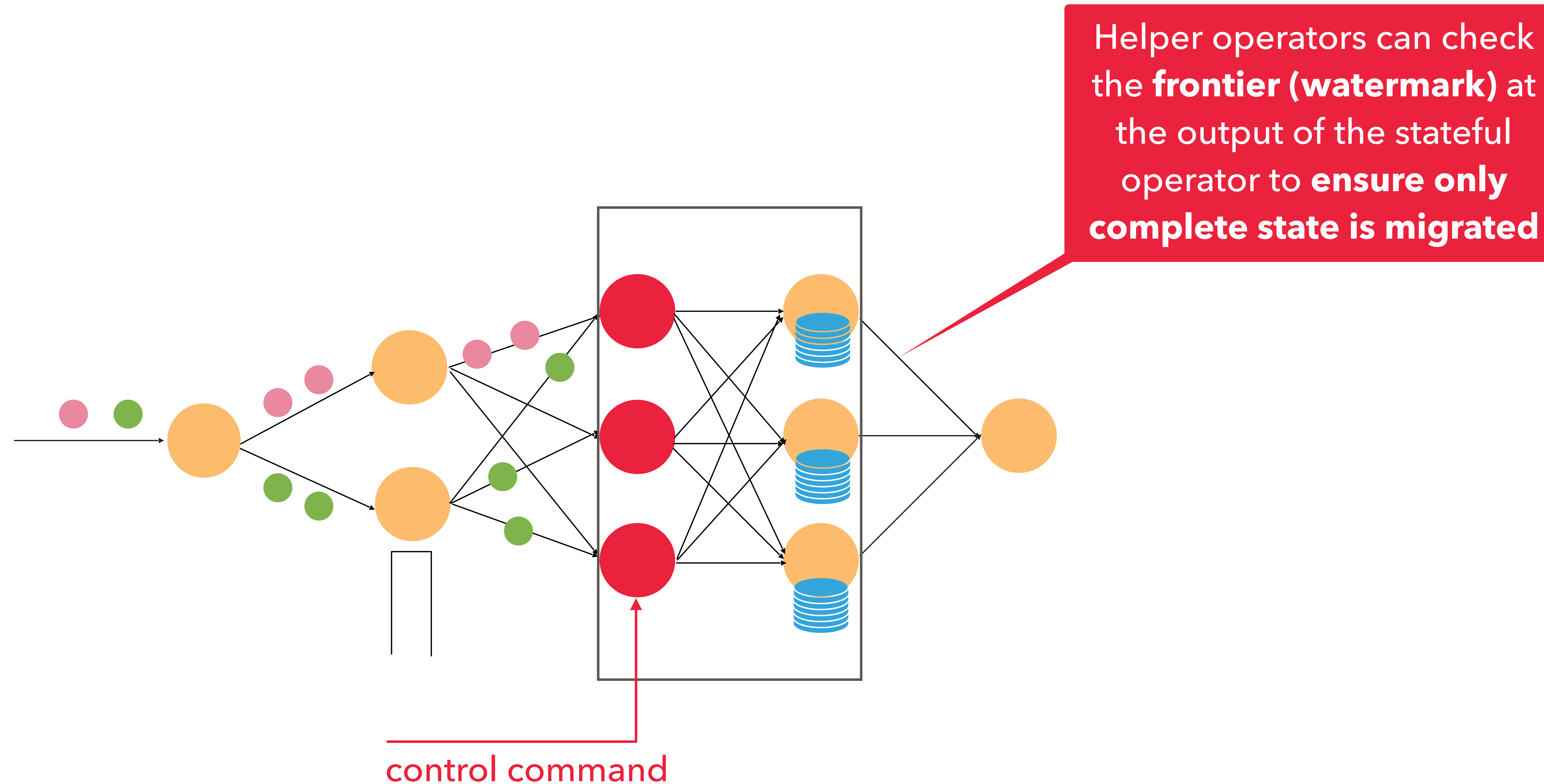
Live state migration



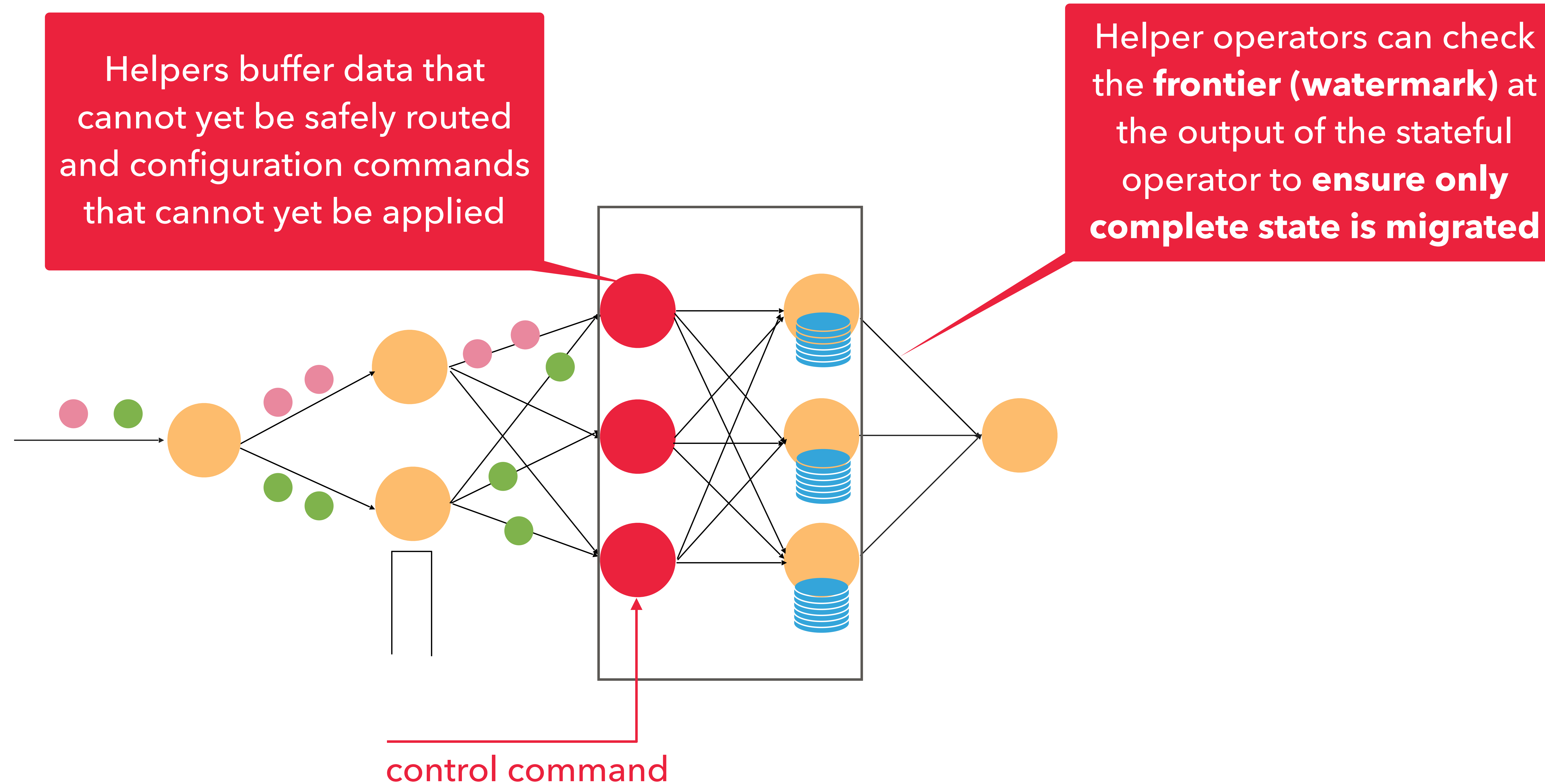
Live state migration



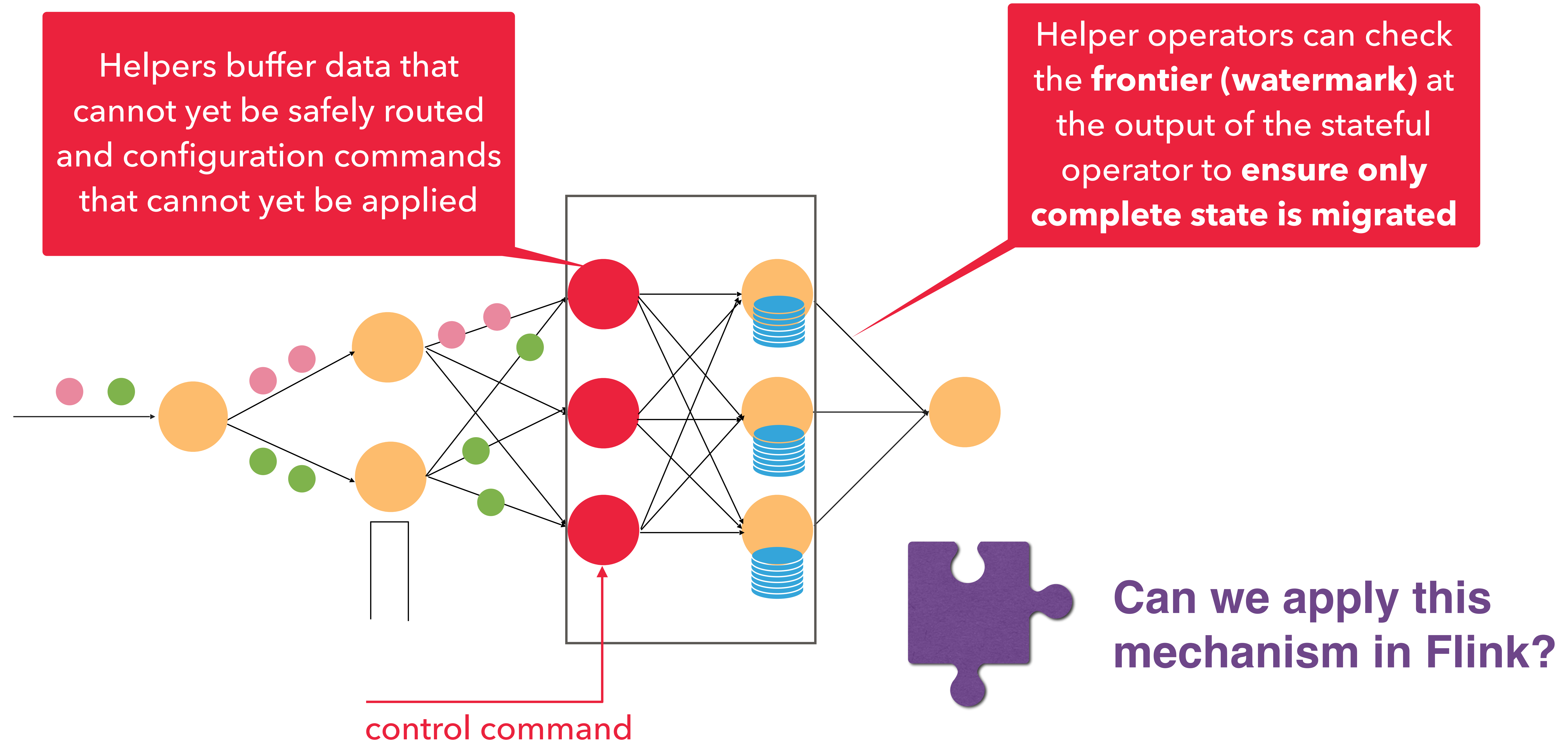
Live state migration



Live state migration



Live state migration



Lecture references

- Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. **Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows**. (OSDI'18).

- Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, Timothy Roscoe. **Megaphone: Latency-conscious state migration for distributed streaming dataflows**. (VLDB 2019).