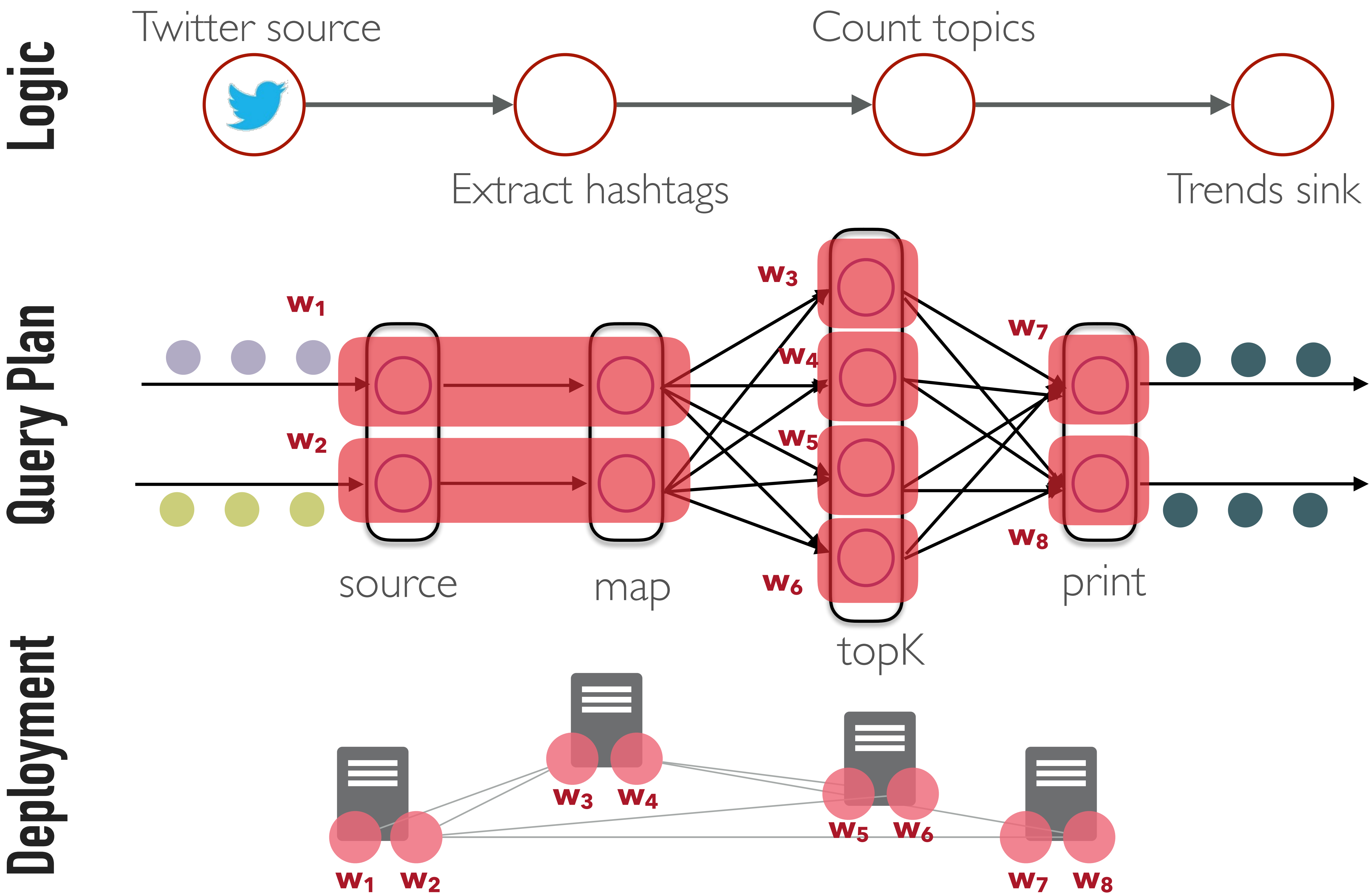
Towards self-managed, re-configurable streaming dataflow systems

Vasia Kalavri
kalavriv@inf.ethz.ch

THE DATAFLOW MODEL

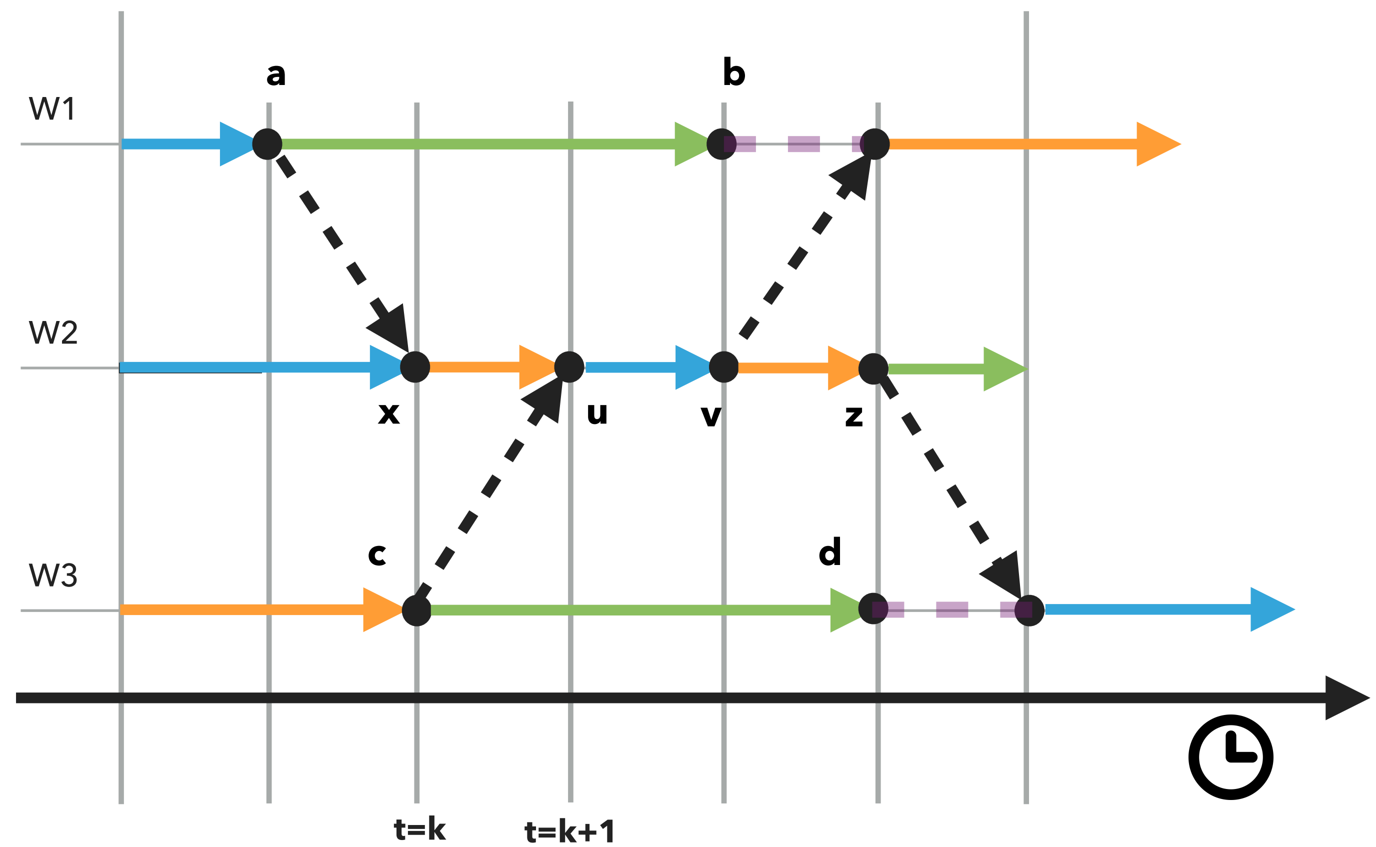
- Computations as **Directed Acyclic Graphs** (DAGs)
  - nodes are operators and edges are data channels
  - operators can accumulate state, have multiple inputs, express event-time custom window-based logic
- Transformations are **data-parallel**
  - distributed workers (threads) execute one parallel instance of one of more operators on *disjoint data partitions*
- Queries are **long-running**
  - input streams are potentially *unbounded*
  - results are *continuously produced*

DATAFLOW COMPUTATIONS

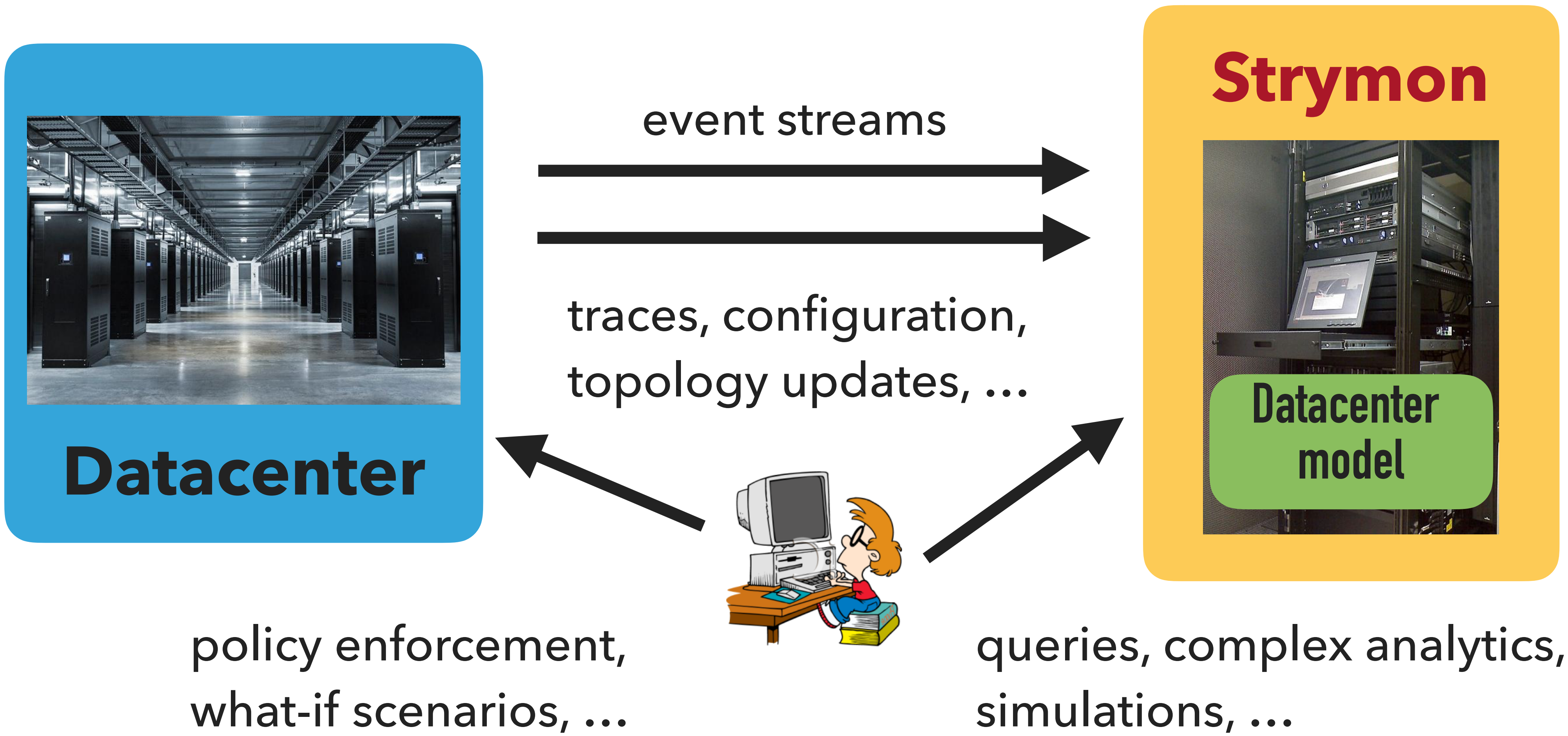


DATAFLOW WORKER ACTIVITIES

- Parallel workers perform activities
  - **receive message**
  - **deserialize**
  - **process**
  - **serialize**
  - **send message**
- Or are **waiting** for
  - input (nothing in the buffer)
  - output (no write buffer available)

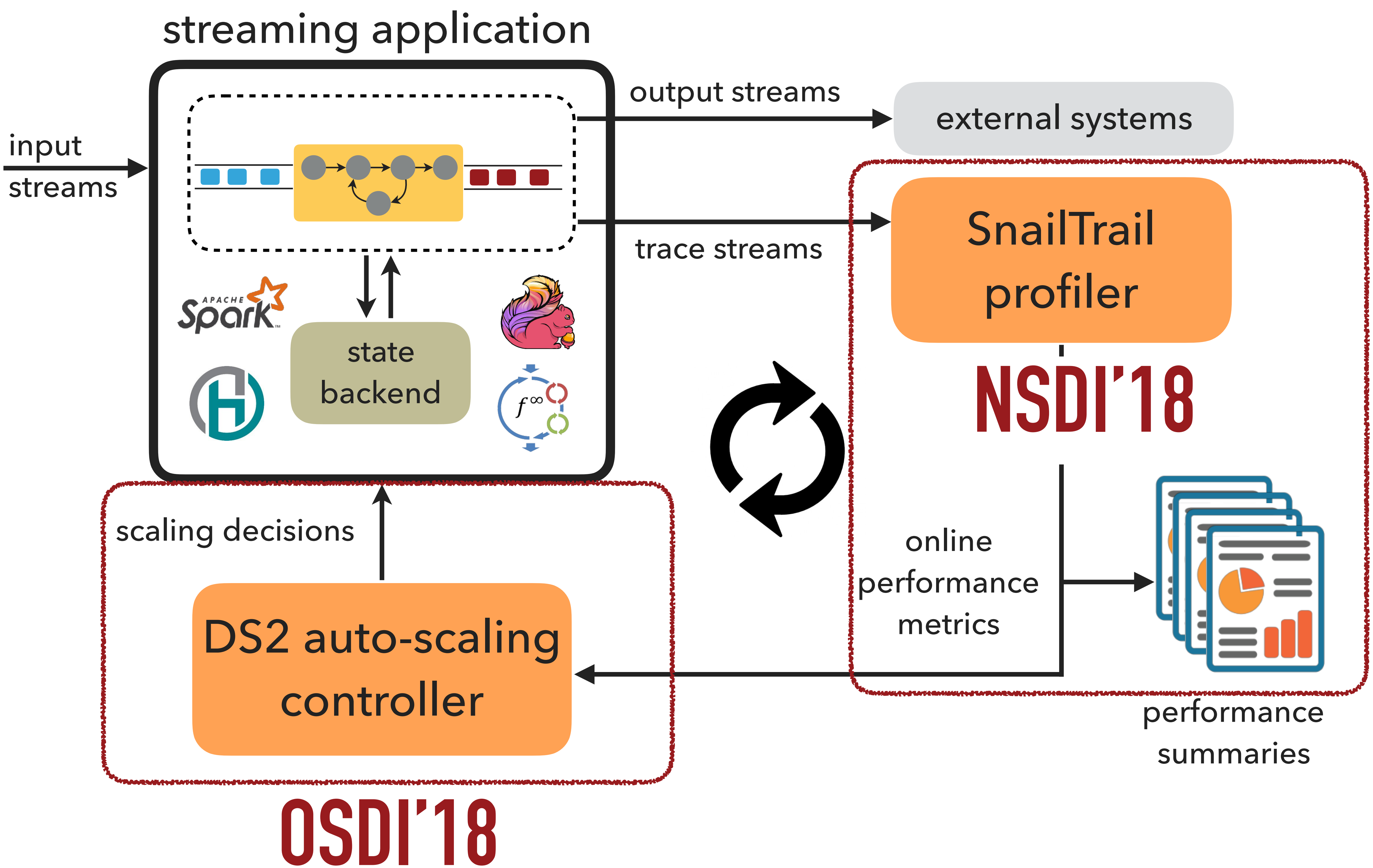


STRYMON: ONLINE DATACENTER ANALYTICS AND MANAGEMENT



strymon.systems.ethz.ch

RECONFIGURABLE STREAM PROCESSING



Snailtrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows

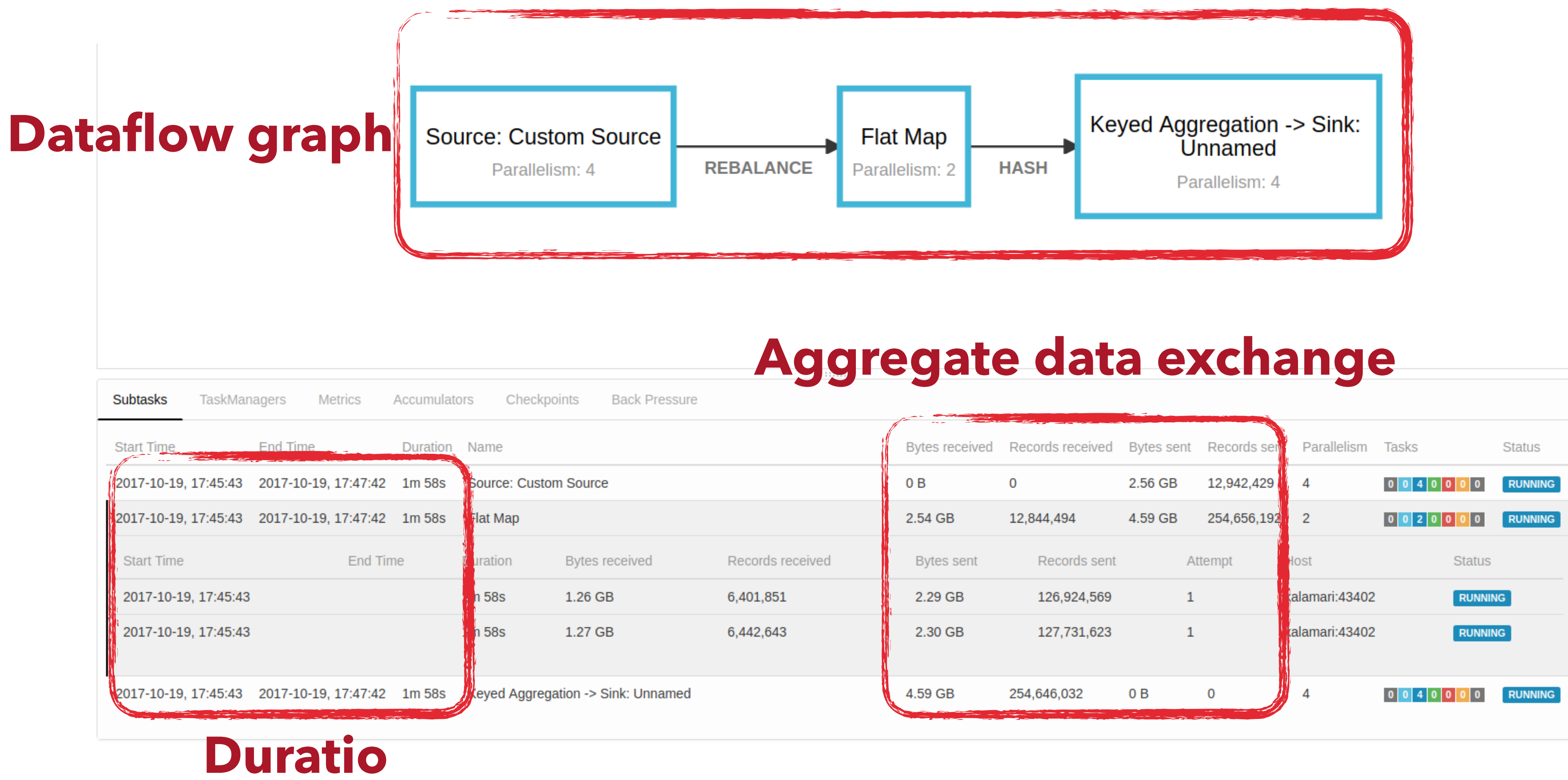
Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, Timothy Roscoe

Systems Group, ETH Zurich

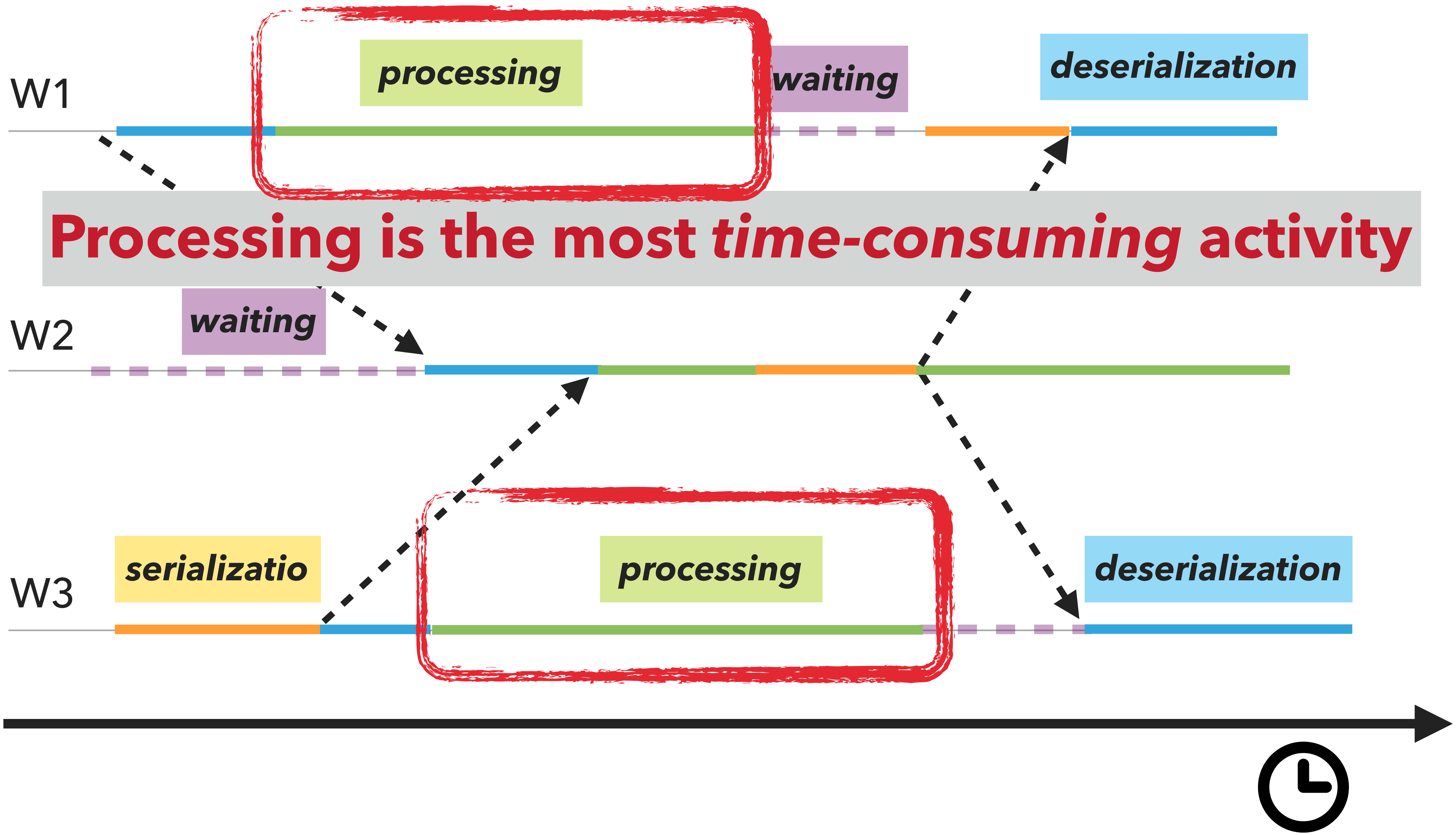
nsdi '18

PERFORMANCE TROUBLESHOOTING

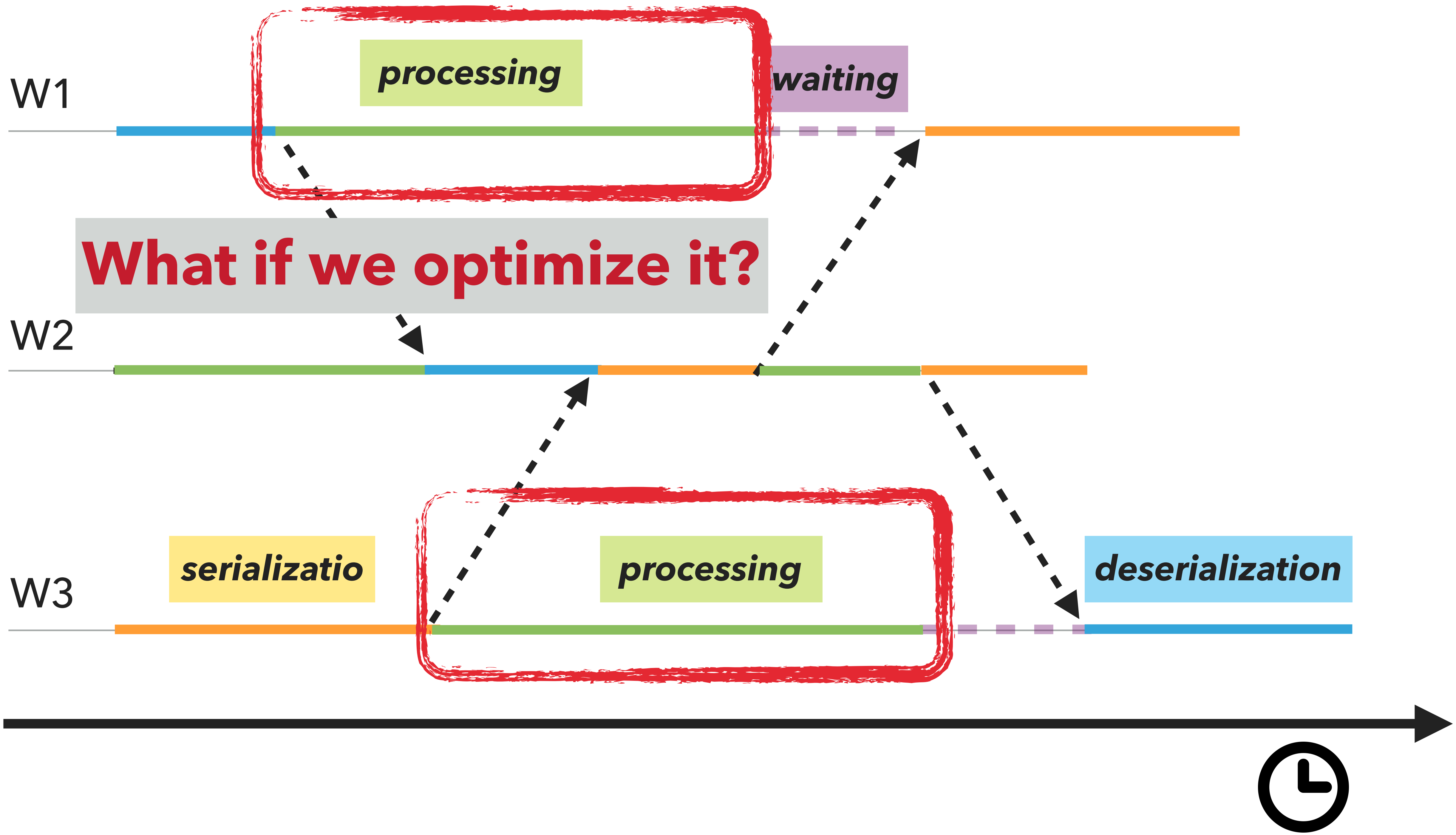
- long-running, dynamic workloads
- many tasks, activities, operators, dependencies
- conventional profiling tools provide *aggregate* information



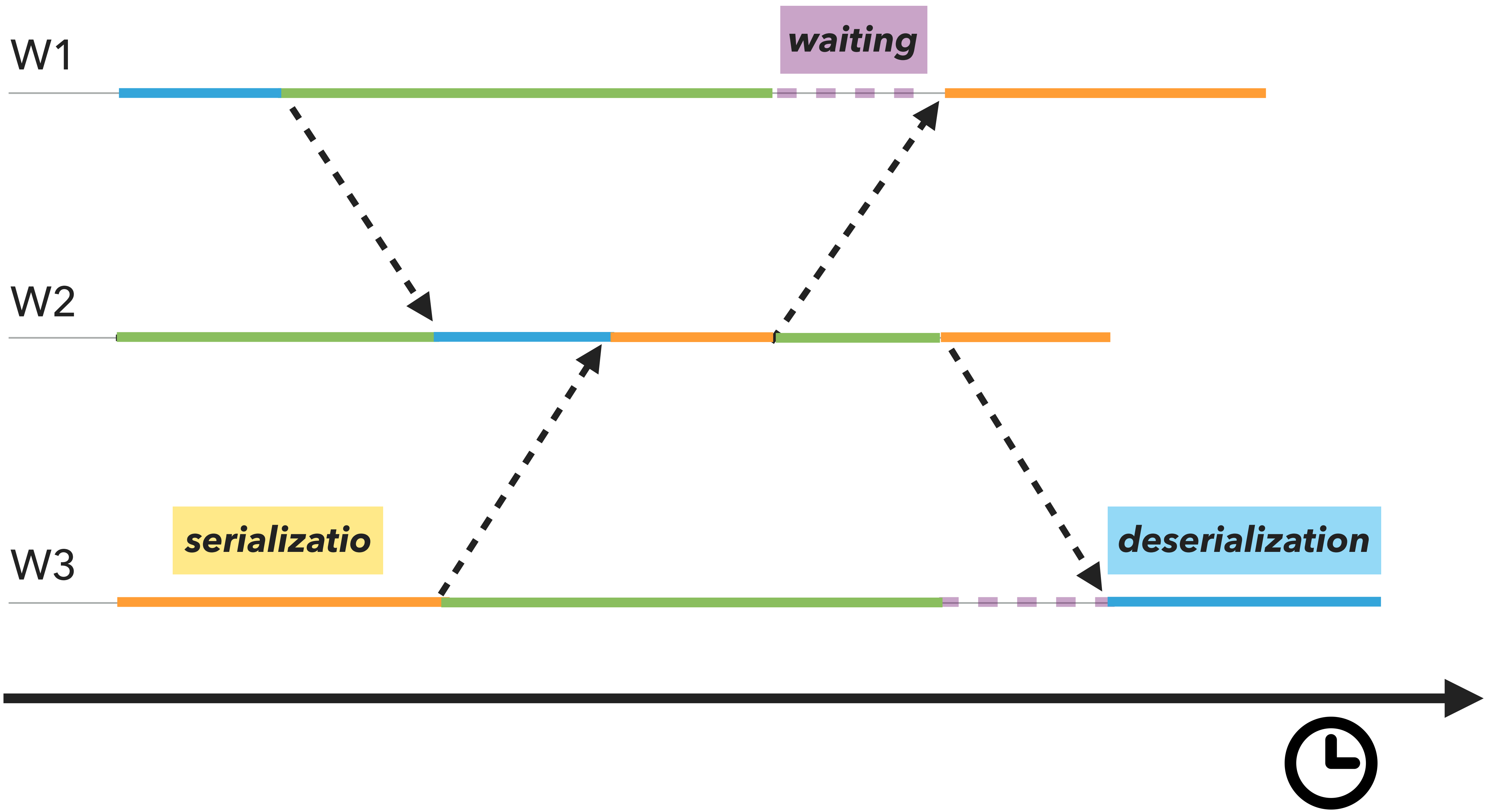
OPTIMIZING ACTIVITY DURATION



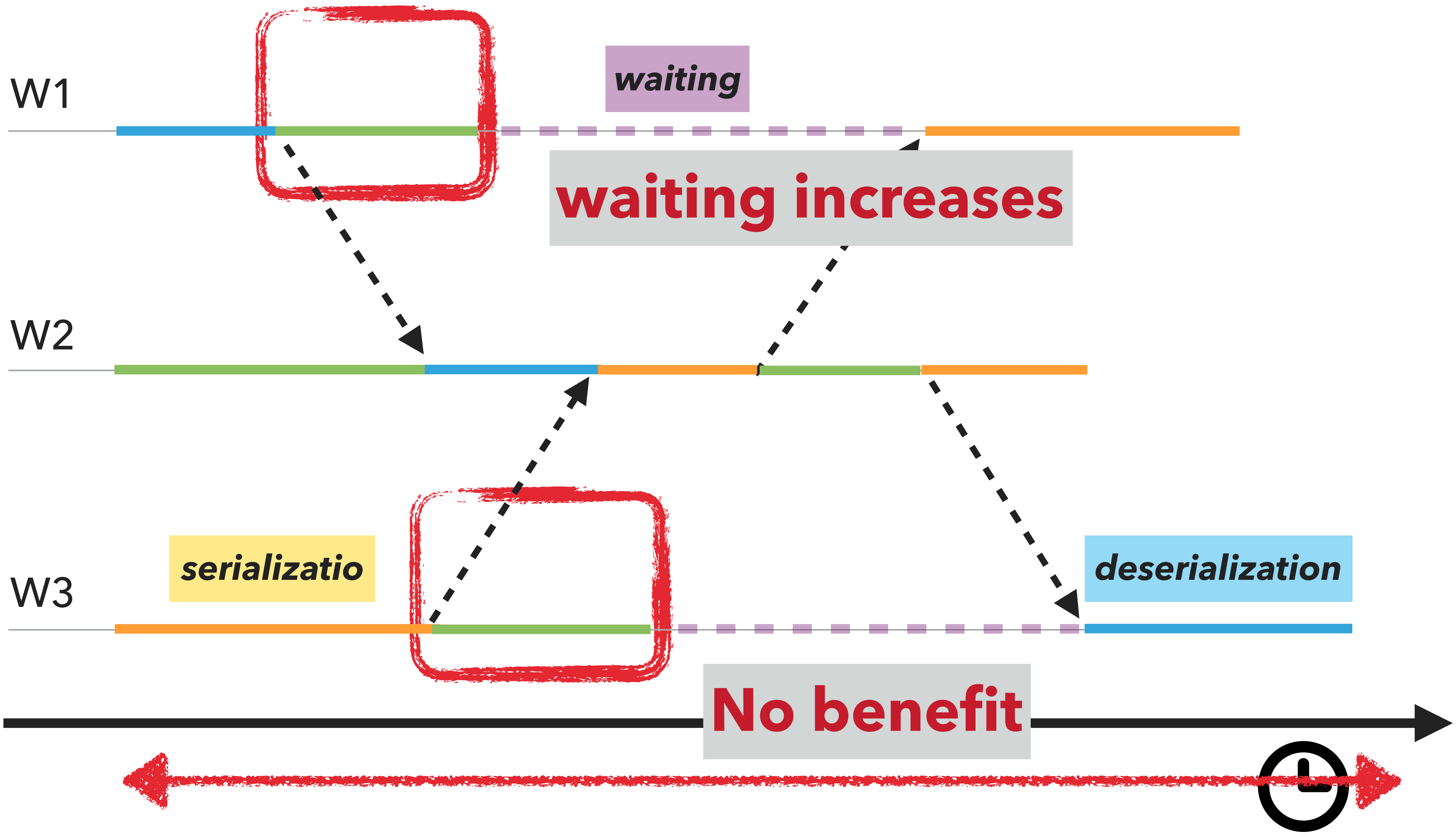
OPTIMIZING ACTIVITY DURATION



OPTIMIZING ACTIVITY DURATION

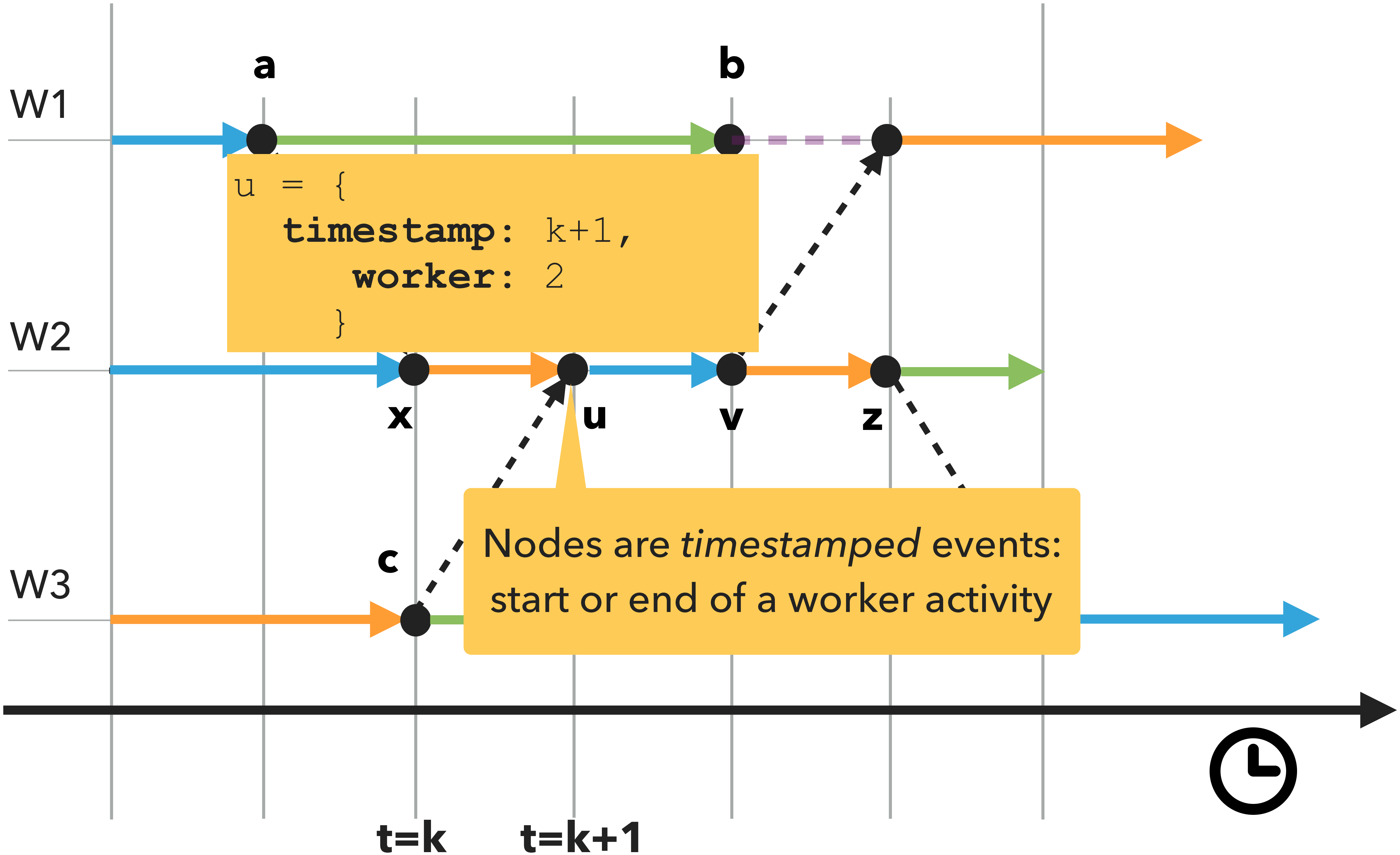


OPTIMIZING ACTIVITY DURATION

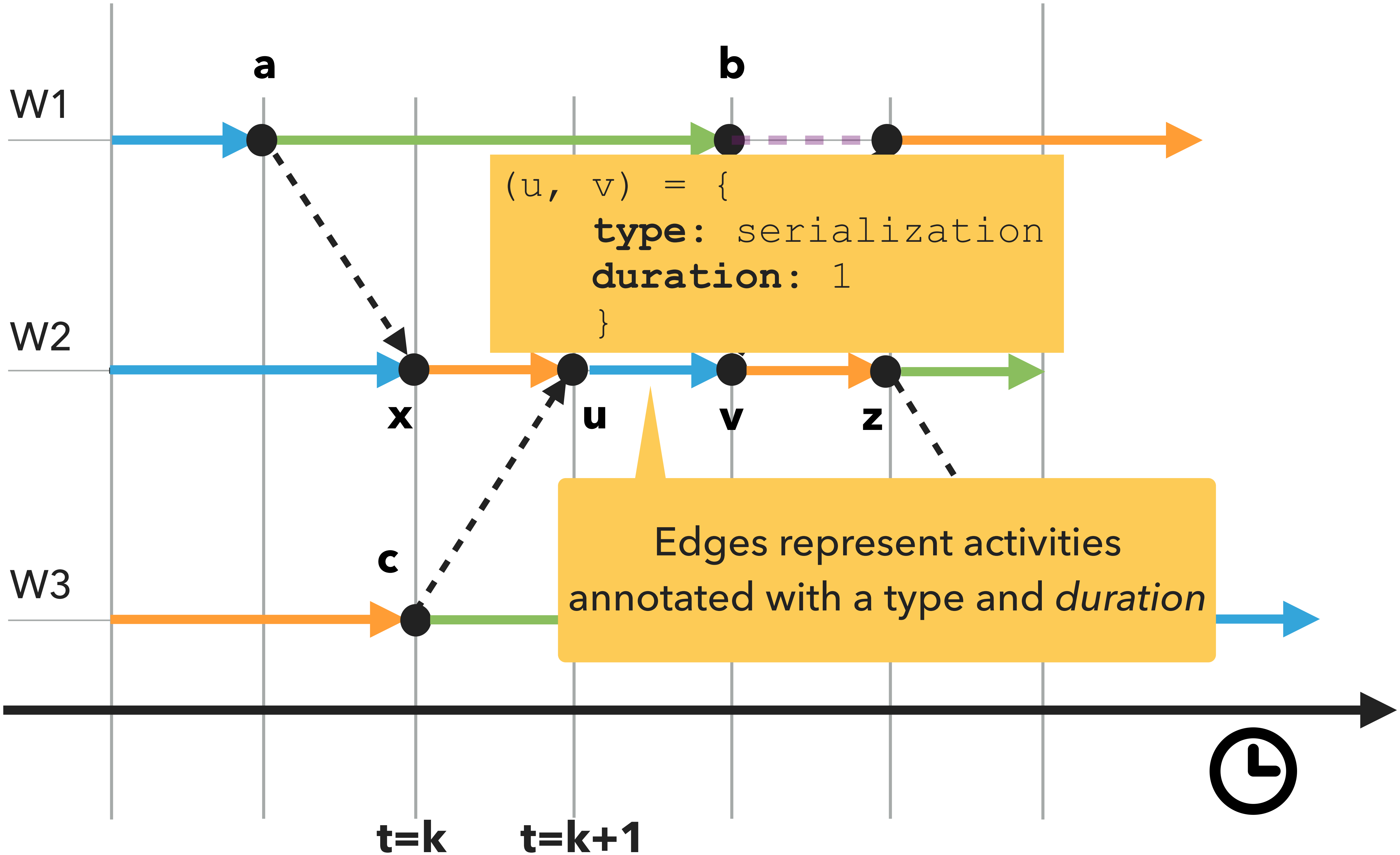


CRITICAL PATH ANALYSIS

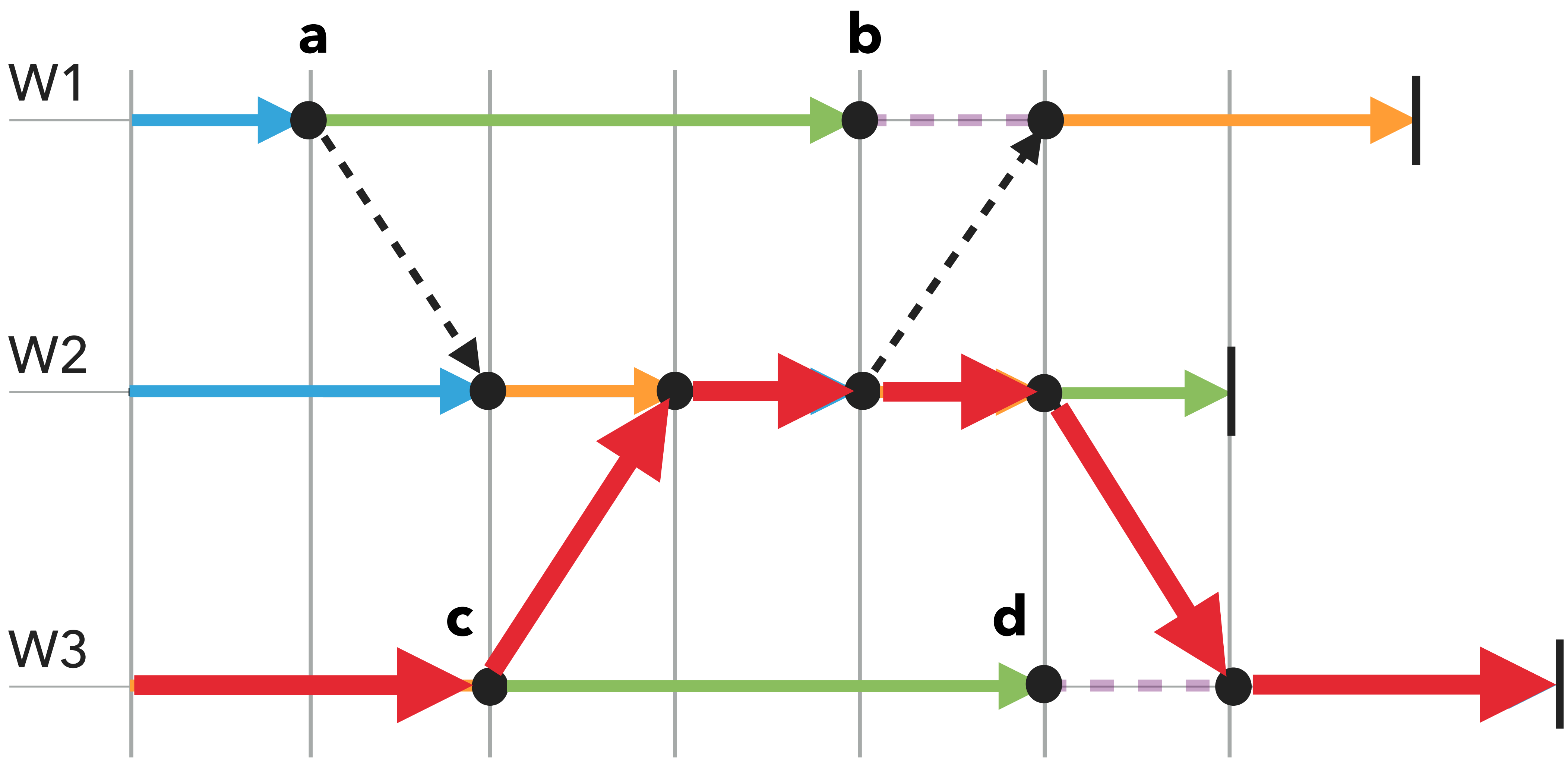
THE PROGRAM ACTIVITY GRAPH (PAG)



THE PROGRAM ACTIVITY GRAPH (PAG)



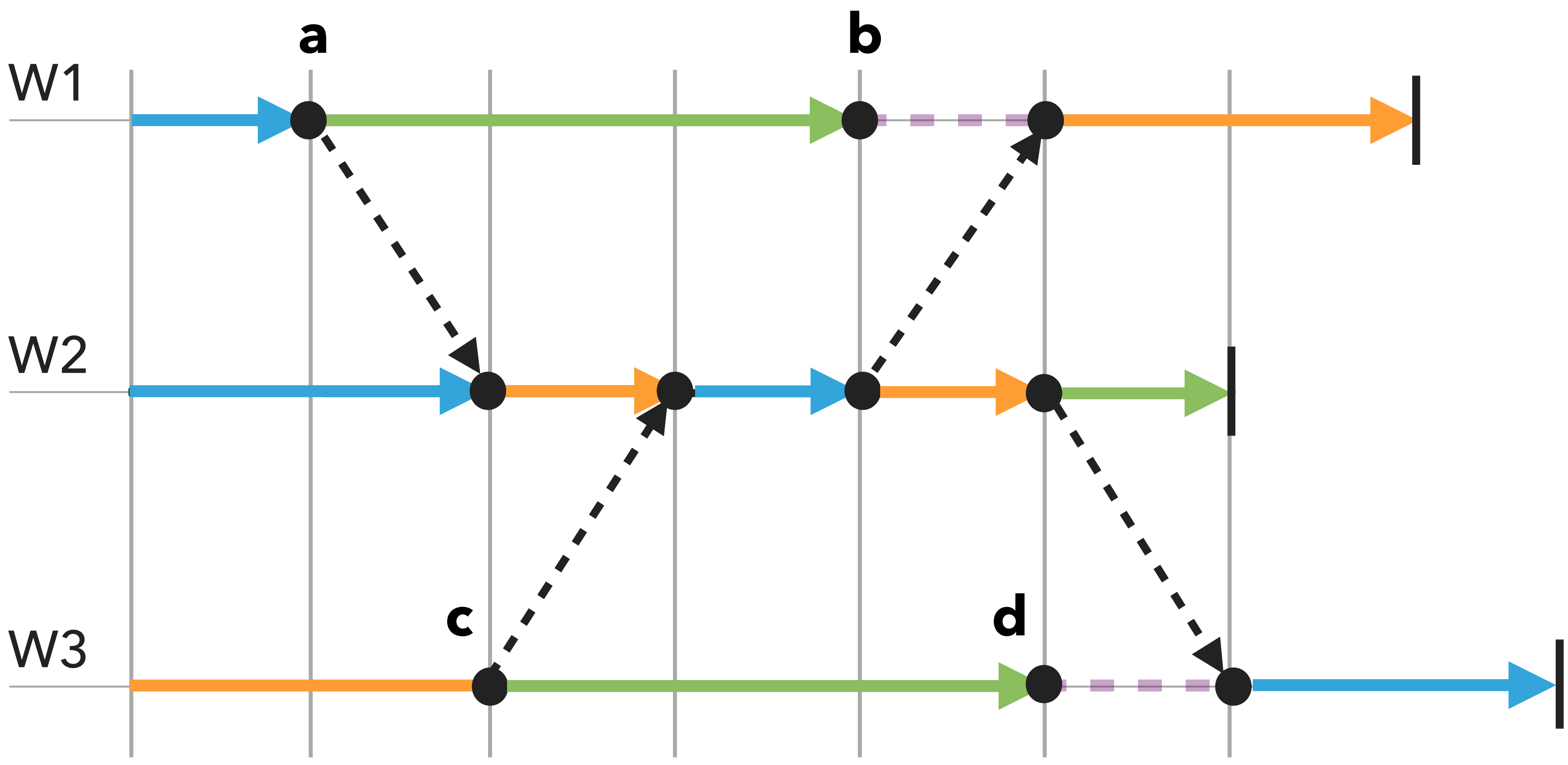
CRITICAL PATH



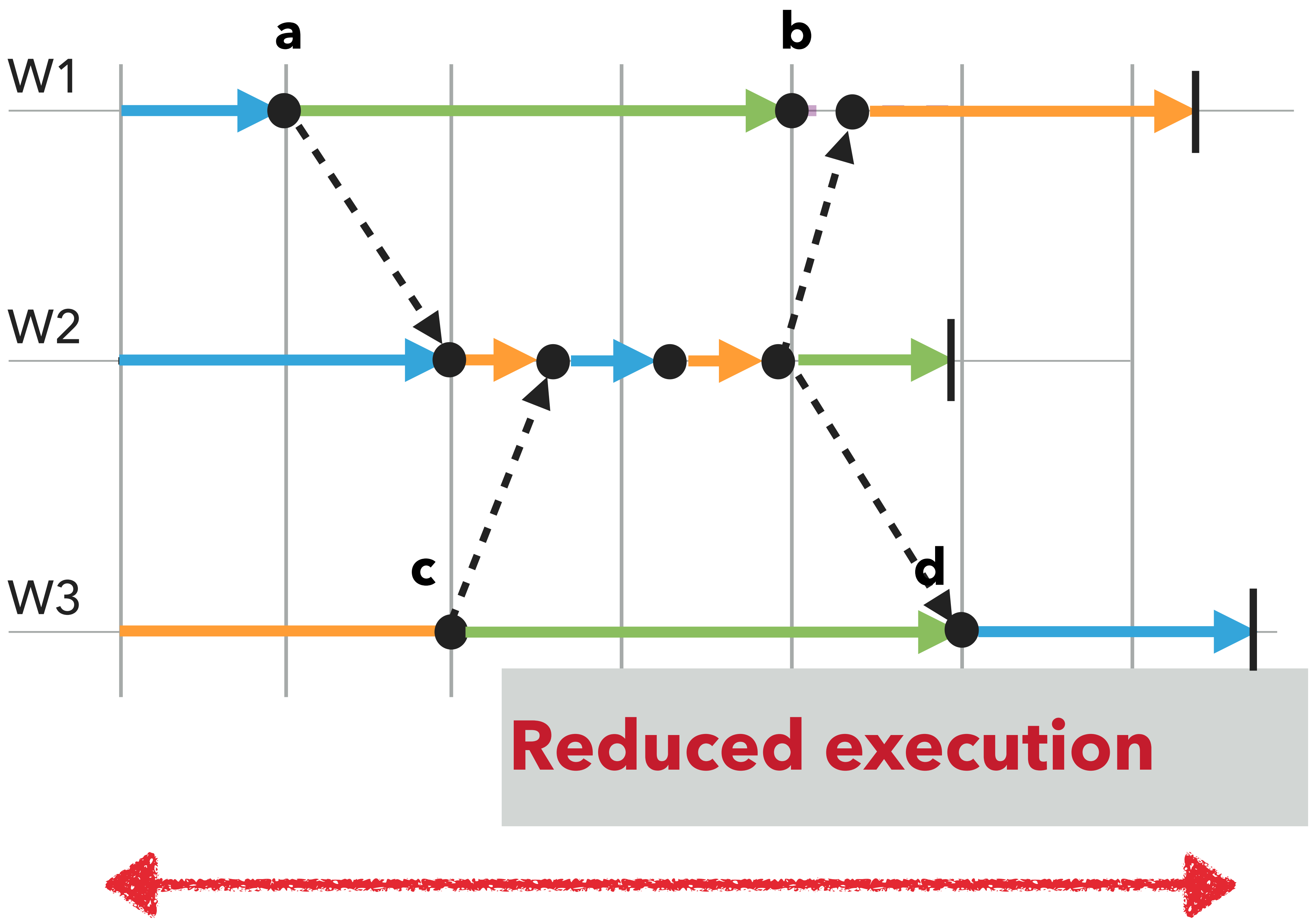
The **longest** path in the execution history

(not considering waiting activities)

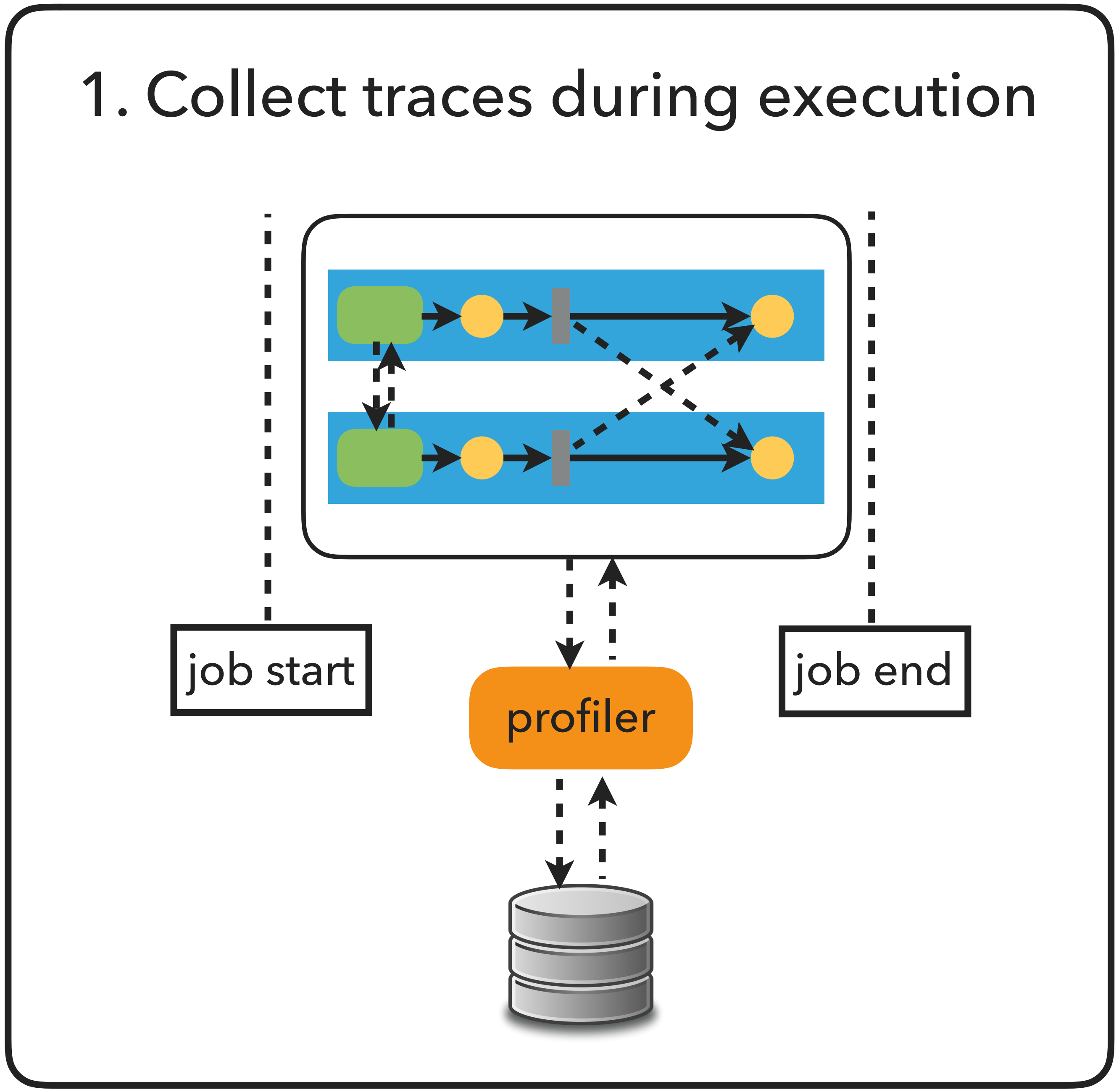
CRITICAL PATH

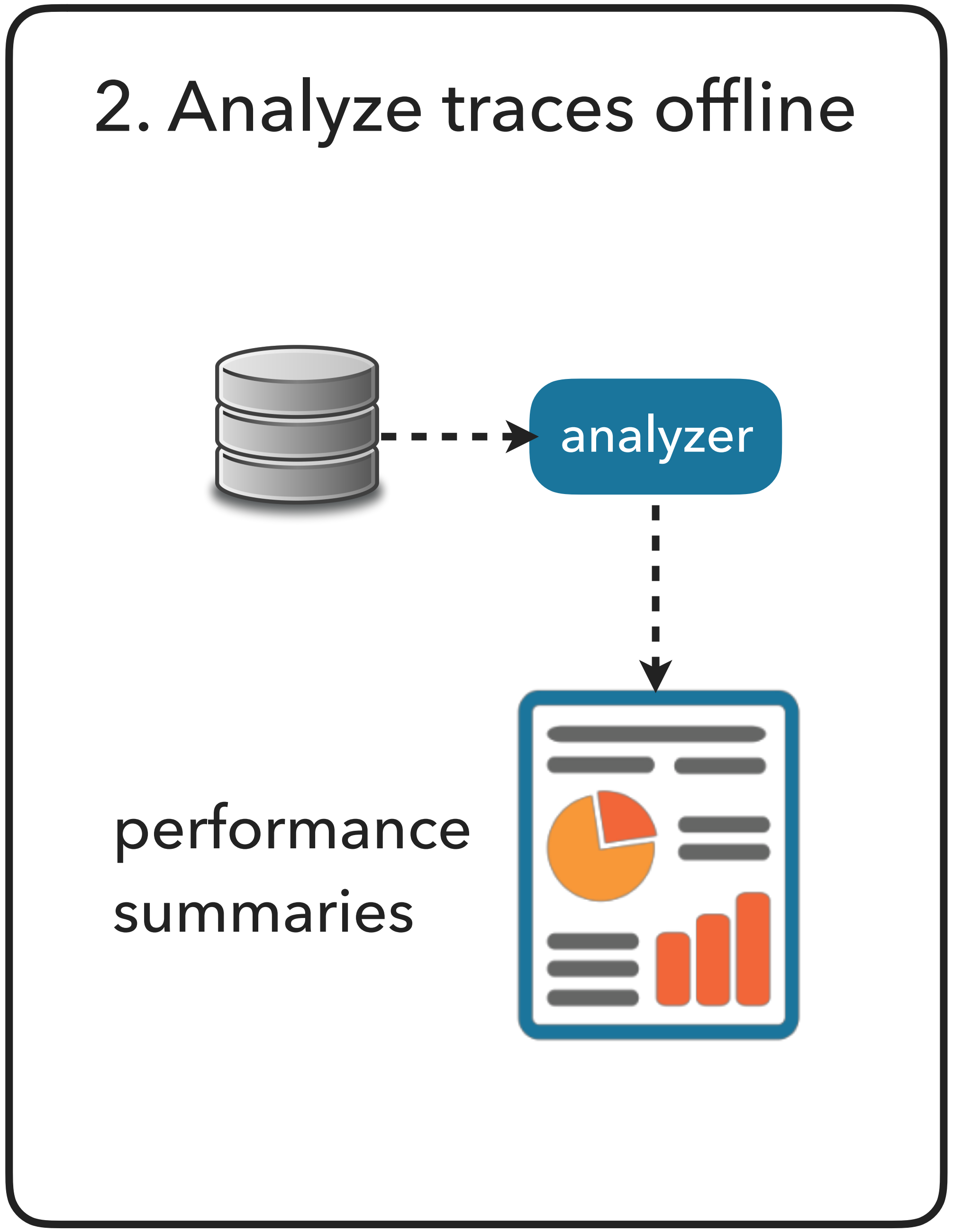


CRITICAL PATH



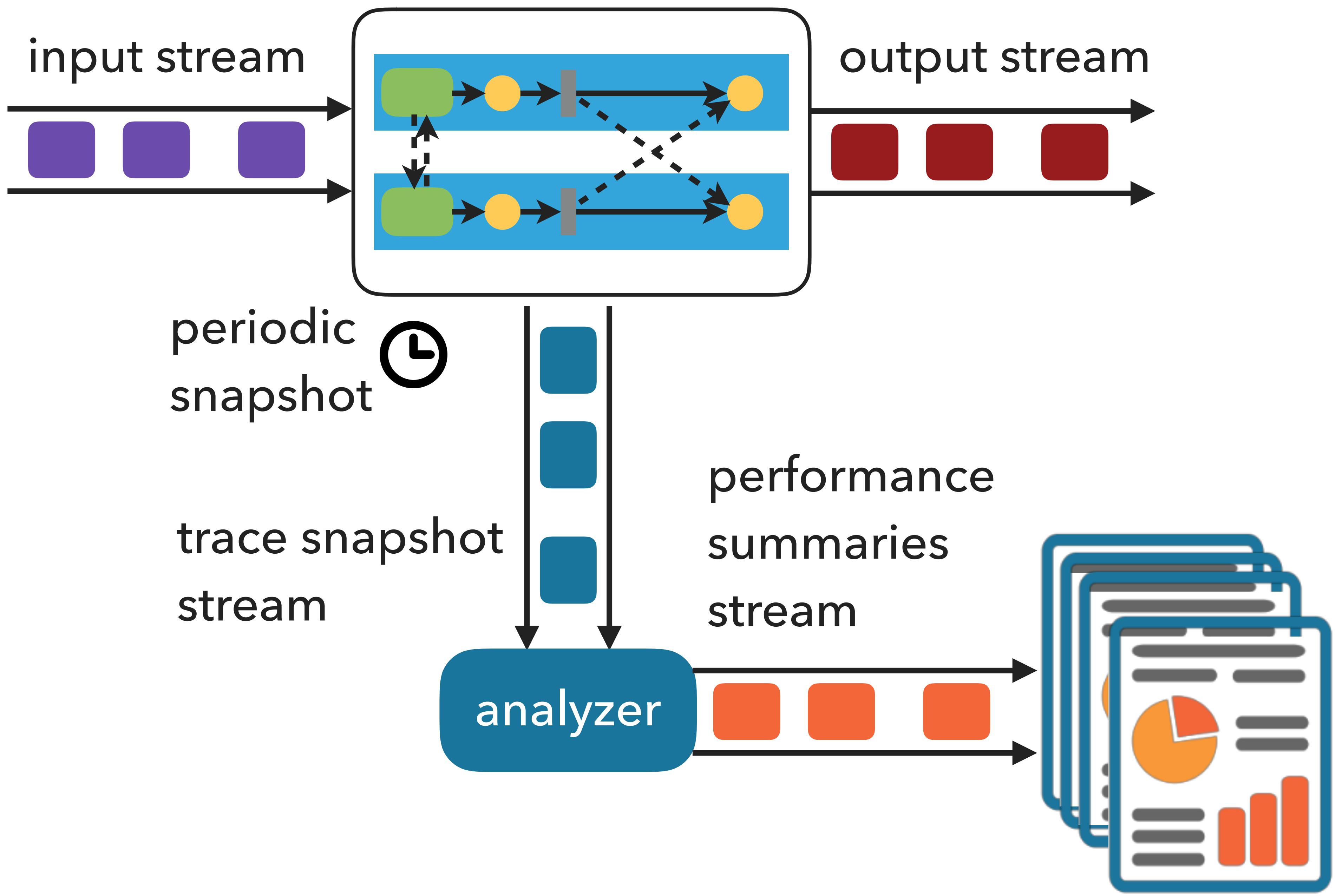
POST-MORTEM CRITICAL PATH ANALYSIS



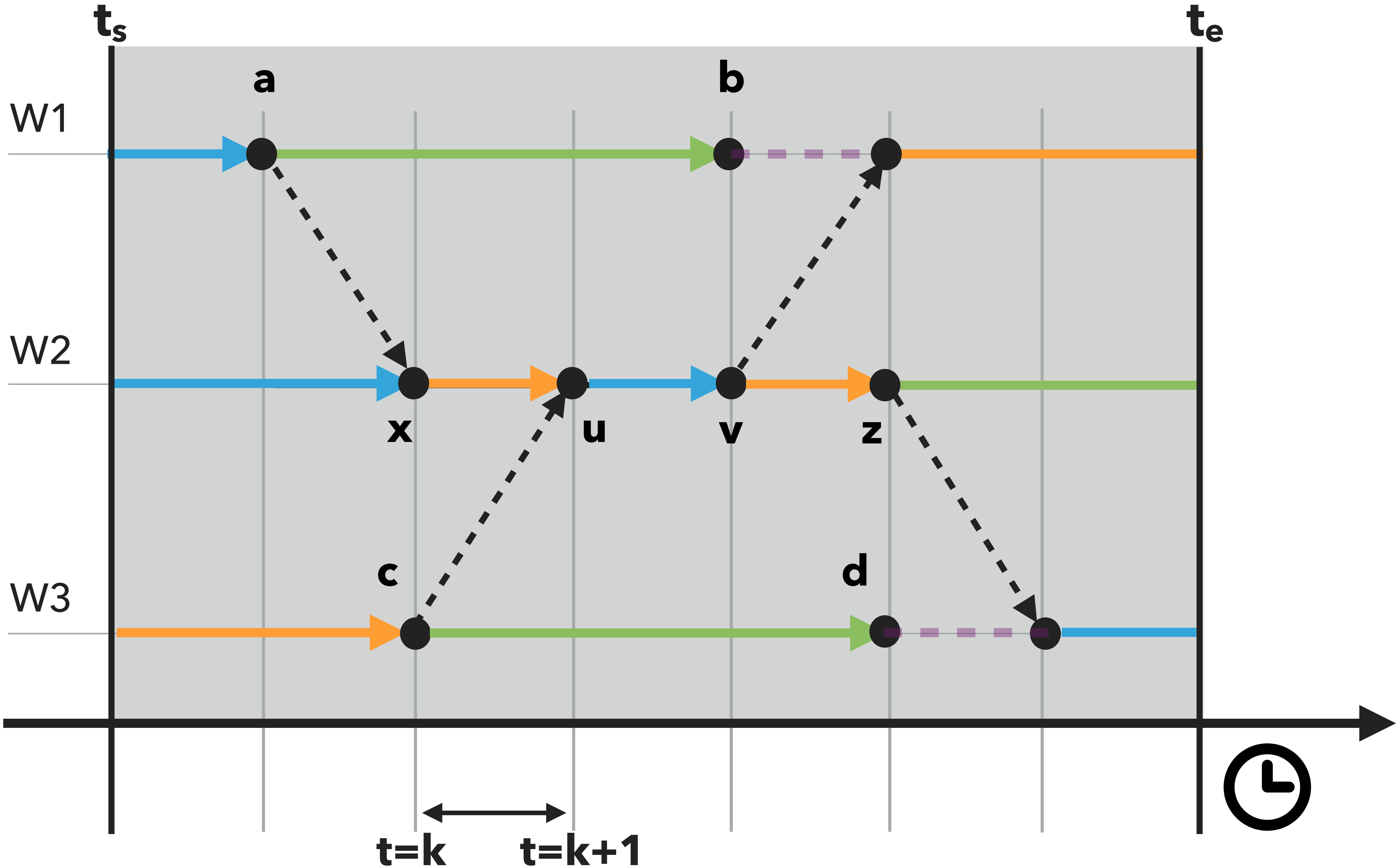


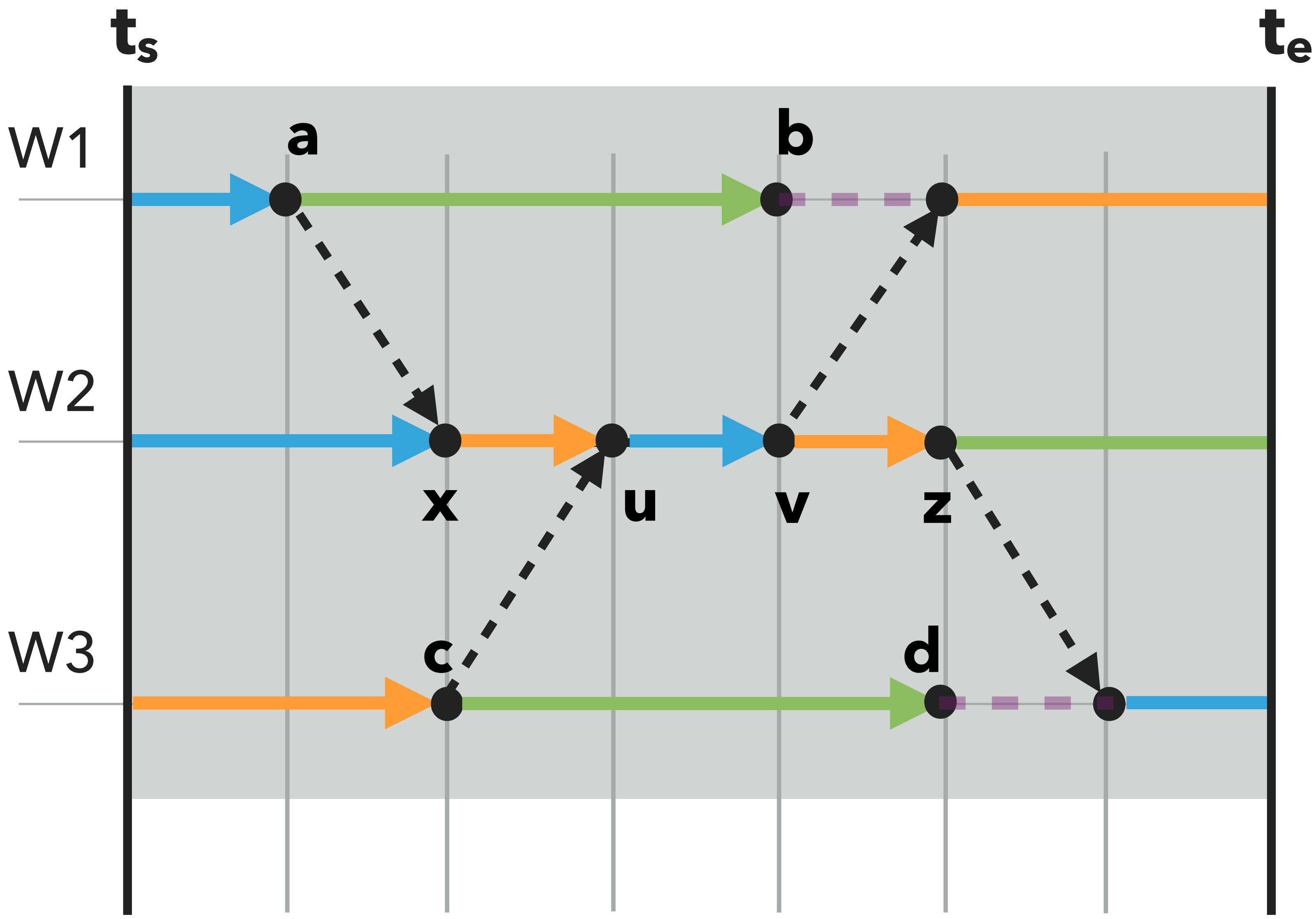
ONLINE CRITICAL PATH ANALYSIS

ONLINE ANALYSIS OF TRACE SNAPSHOTS

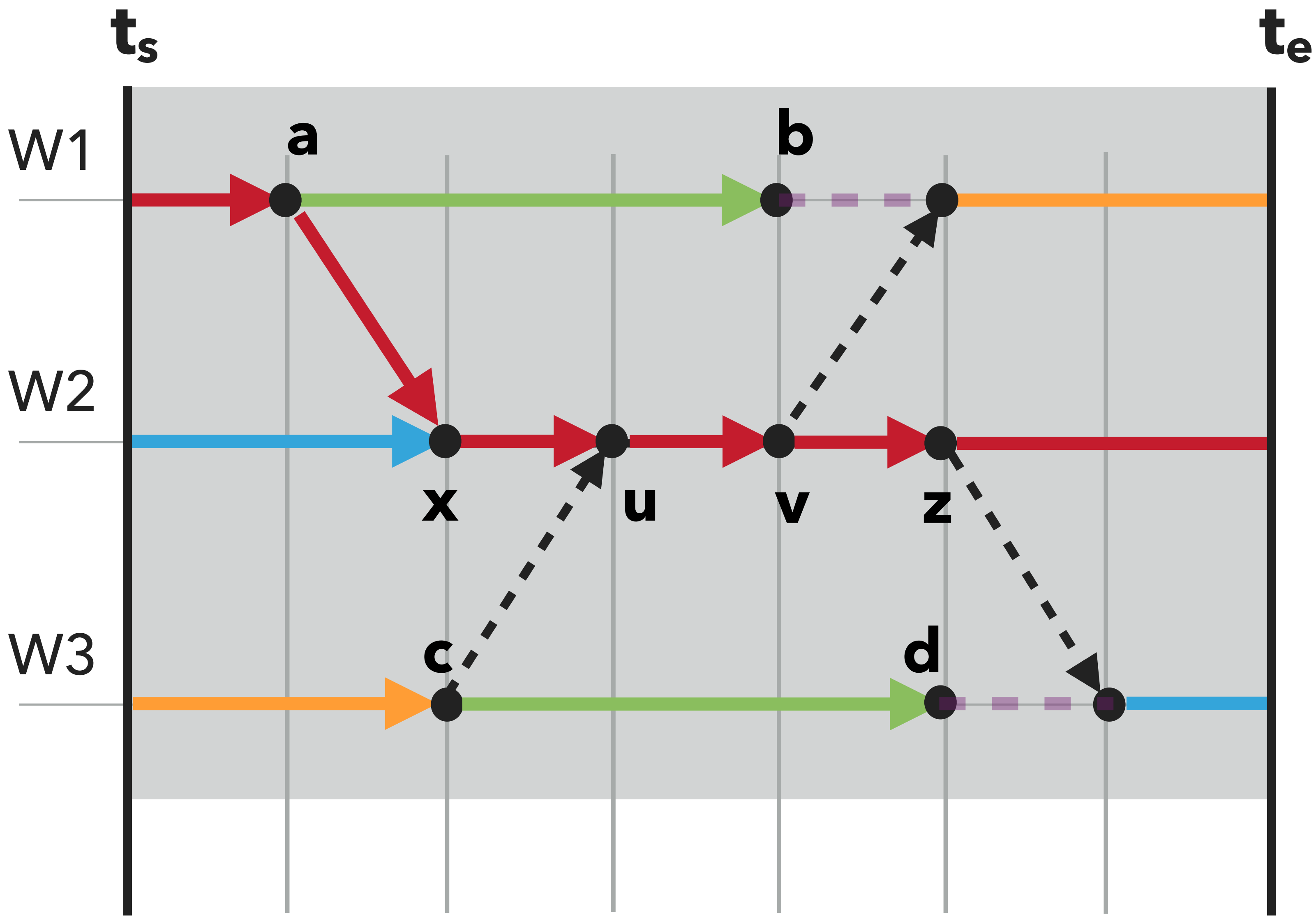


PROGRAM ACTIVITY GRAPH SNAPSHOT

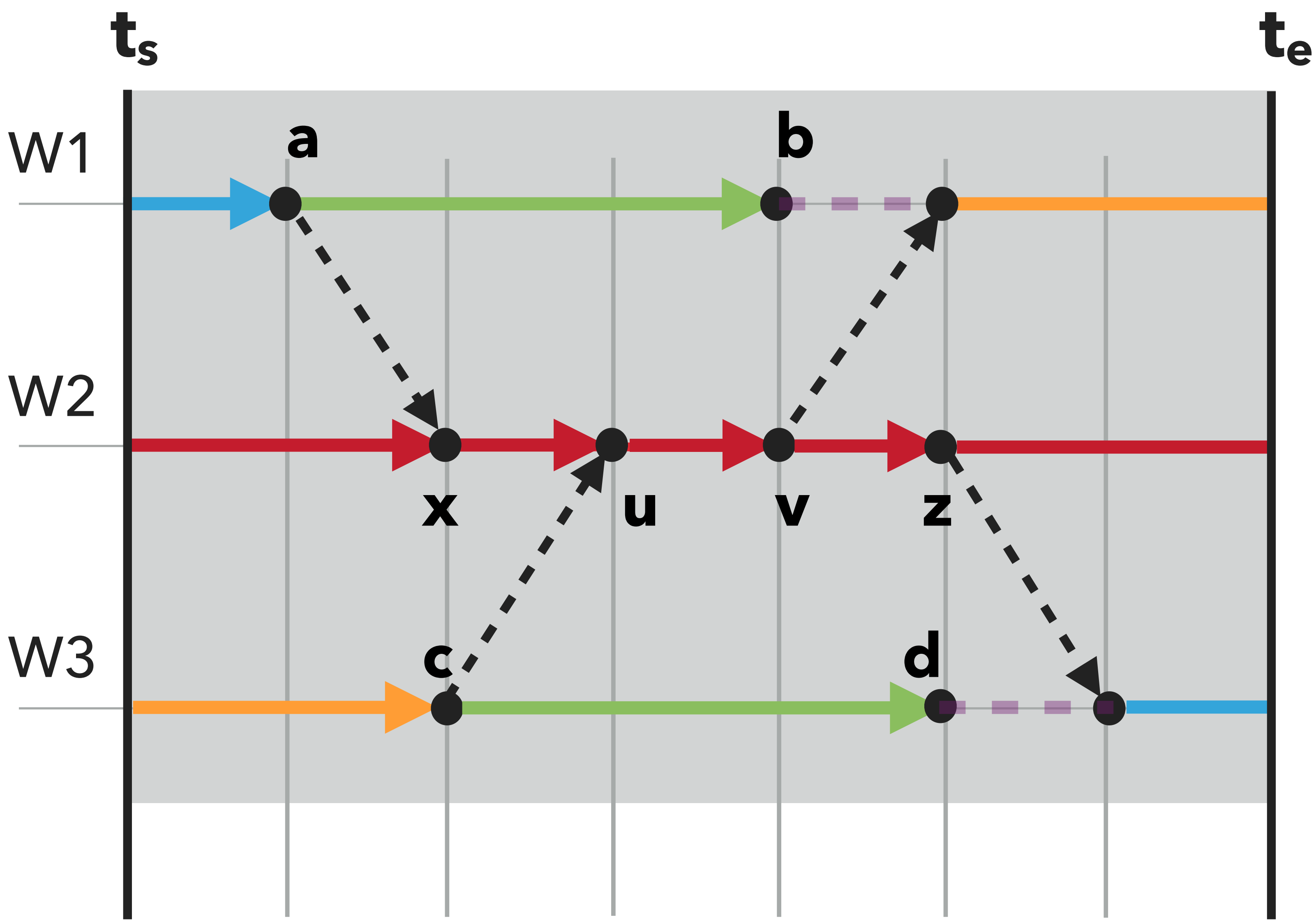




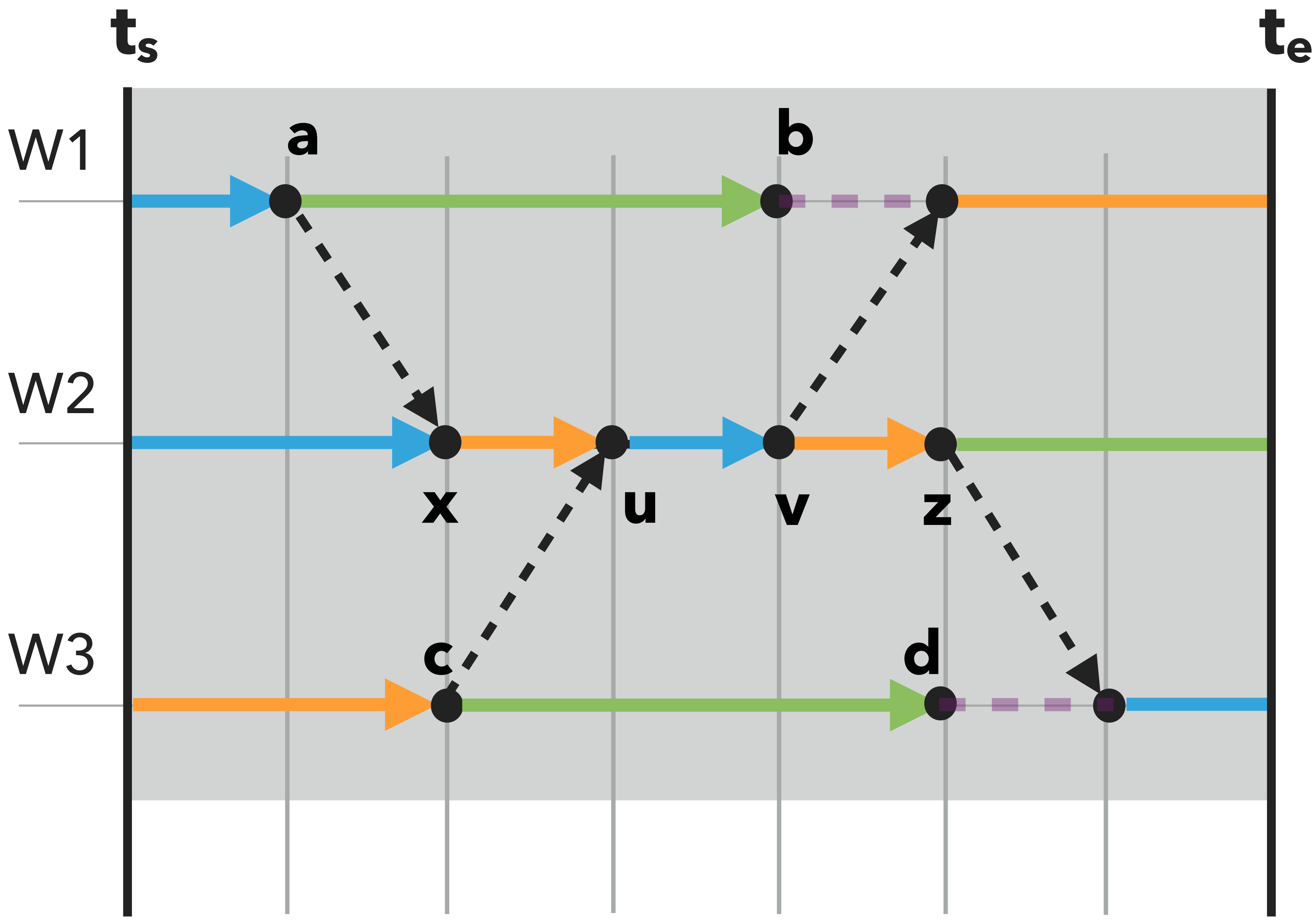
- All paths have the same length: $t_e - t_s$



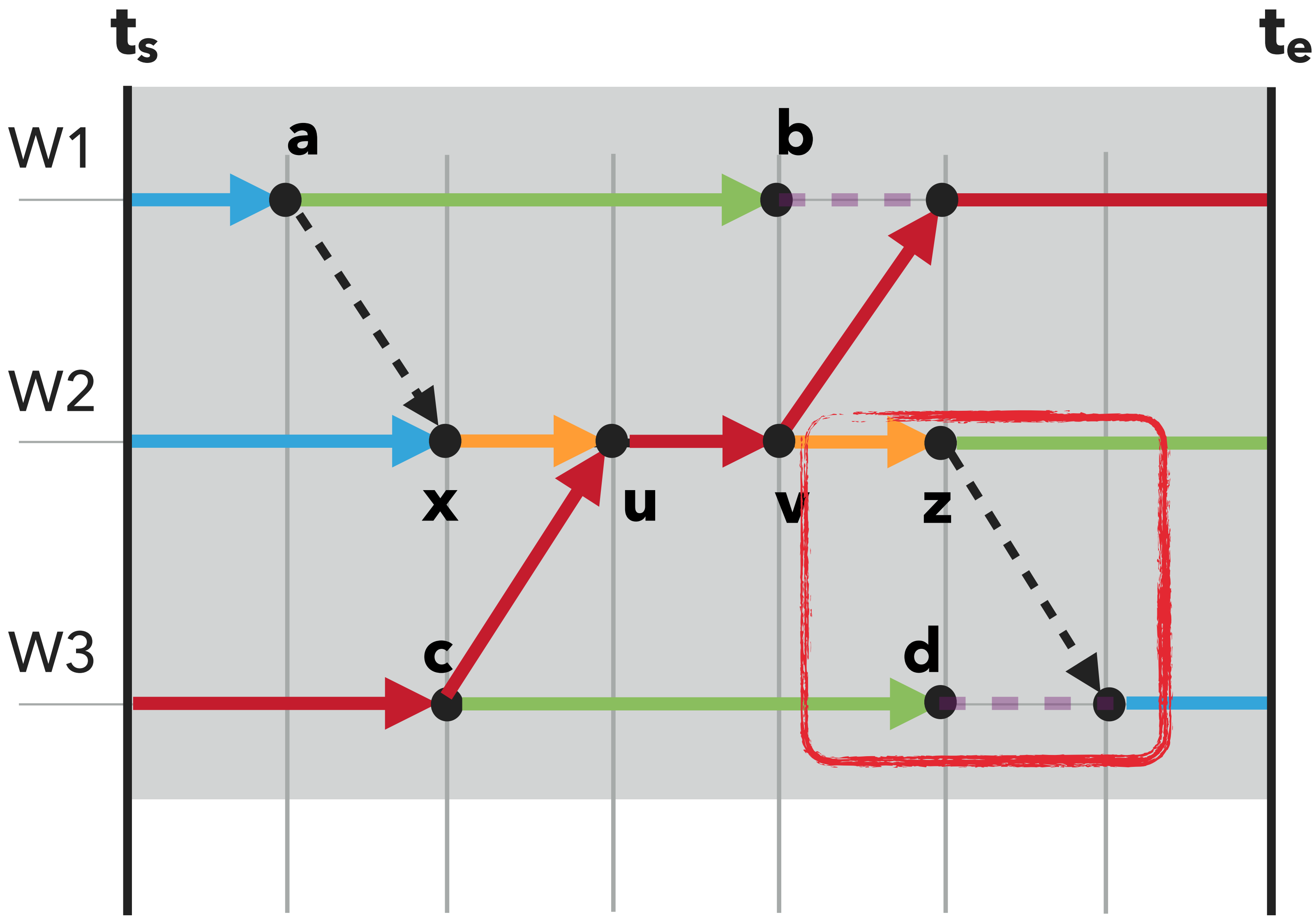
- All paths have the same length: $t_e - t_s$



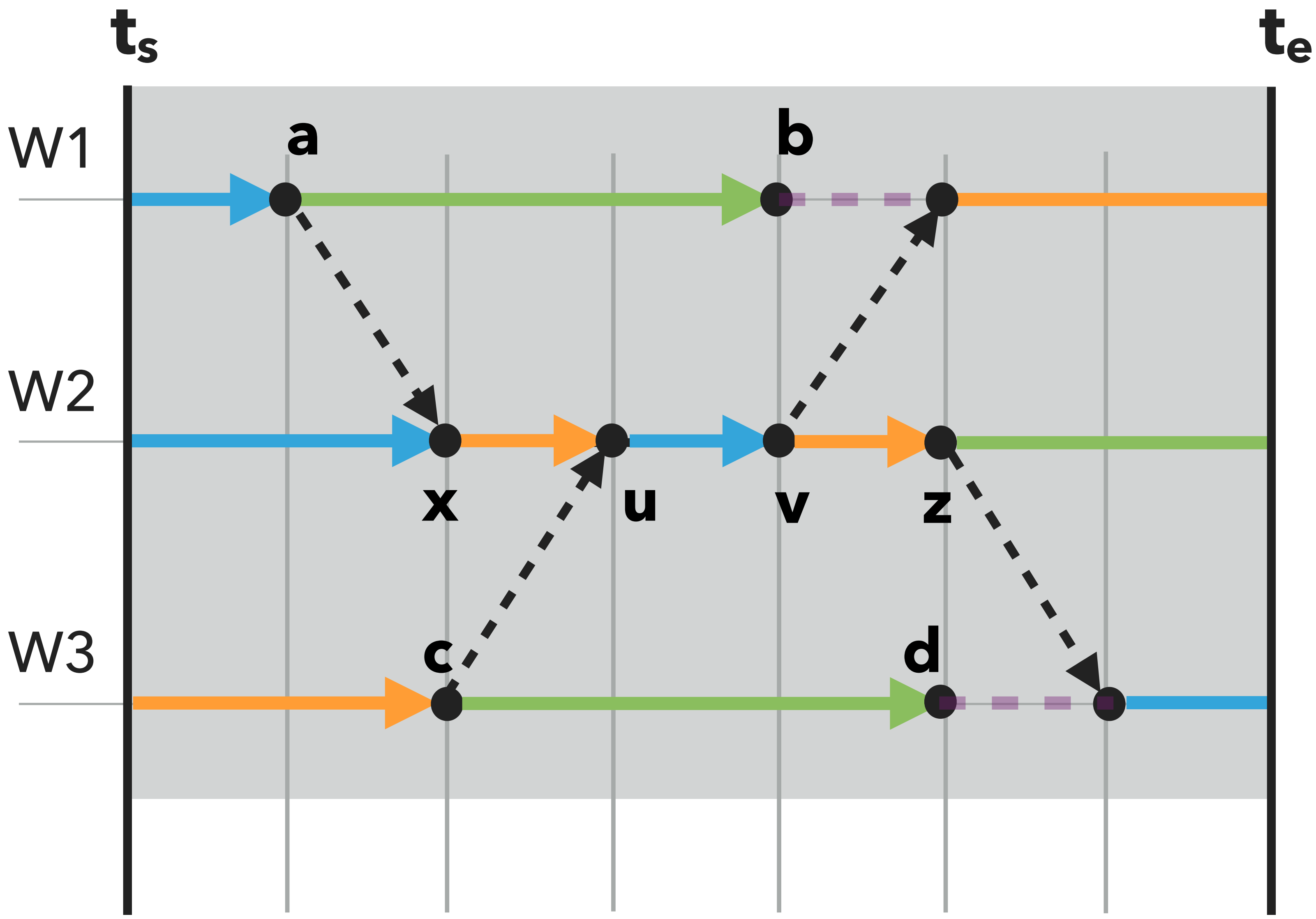
- All paths have the same length: $t_e - t_s$



- All paths have the same length: $t_e - t_s$
- Choosing a random path might miss critical activities

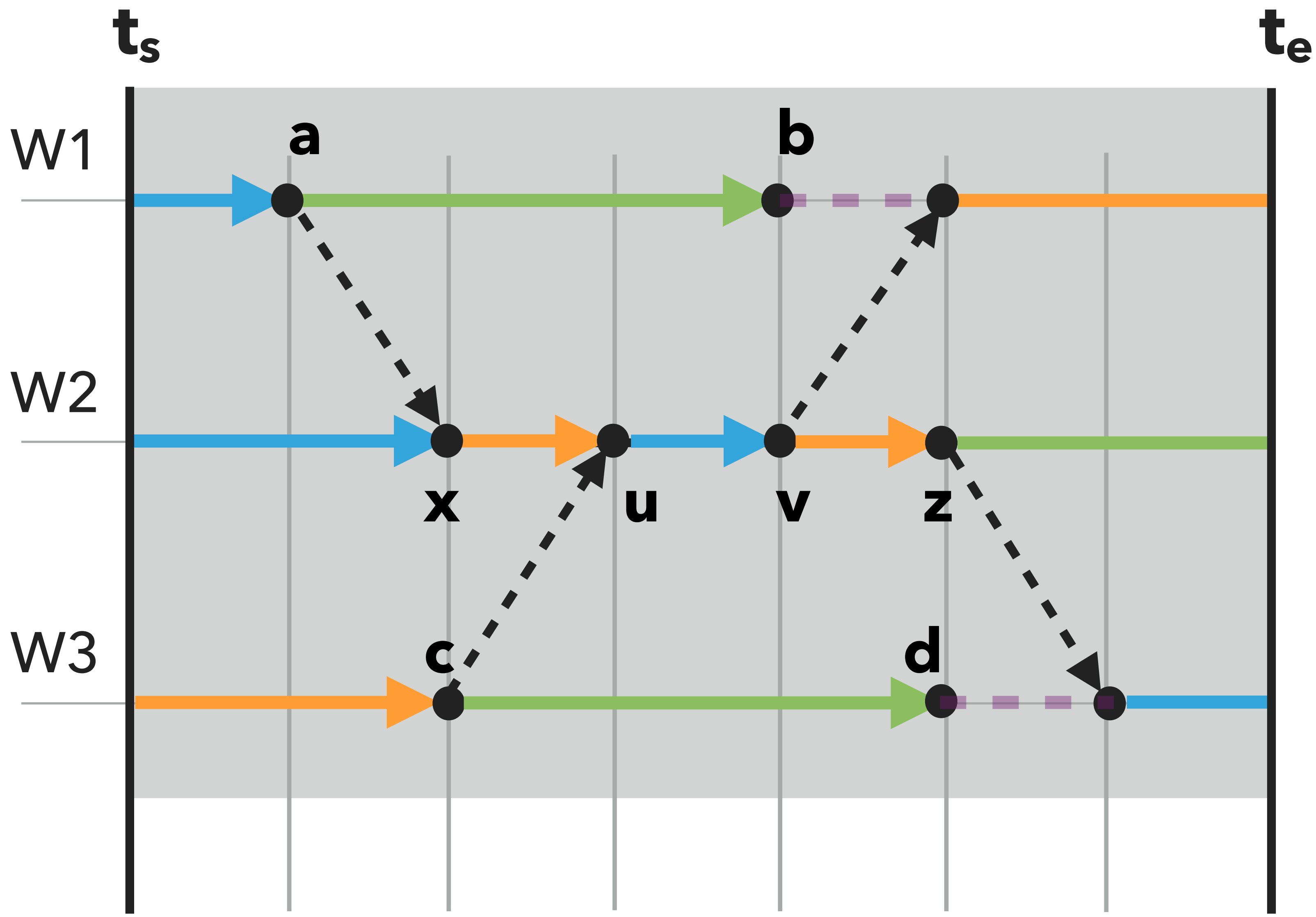


- All paths have the same length: $t_e - t_s$
- Choosing a random path might miss critical activities



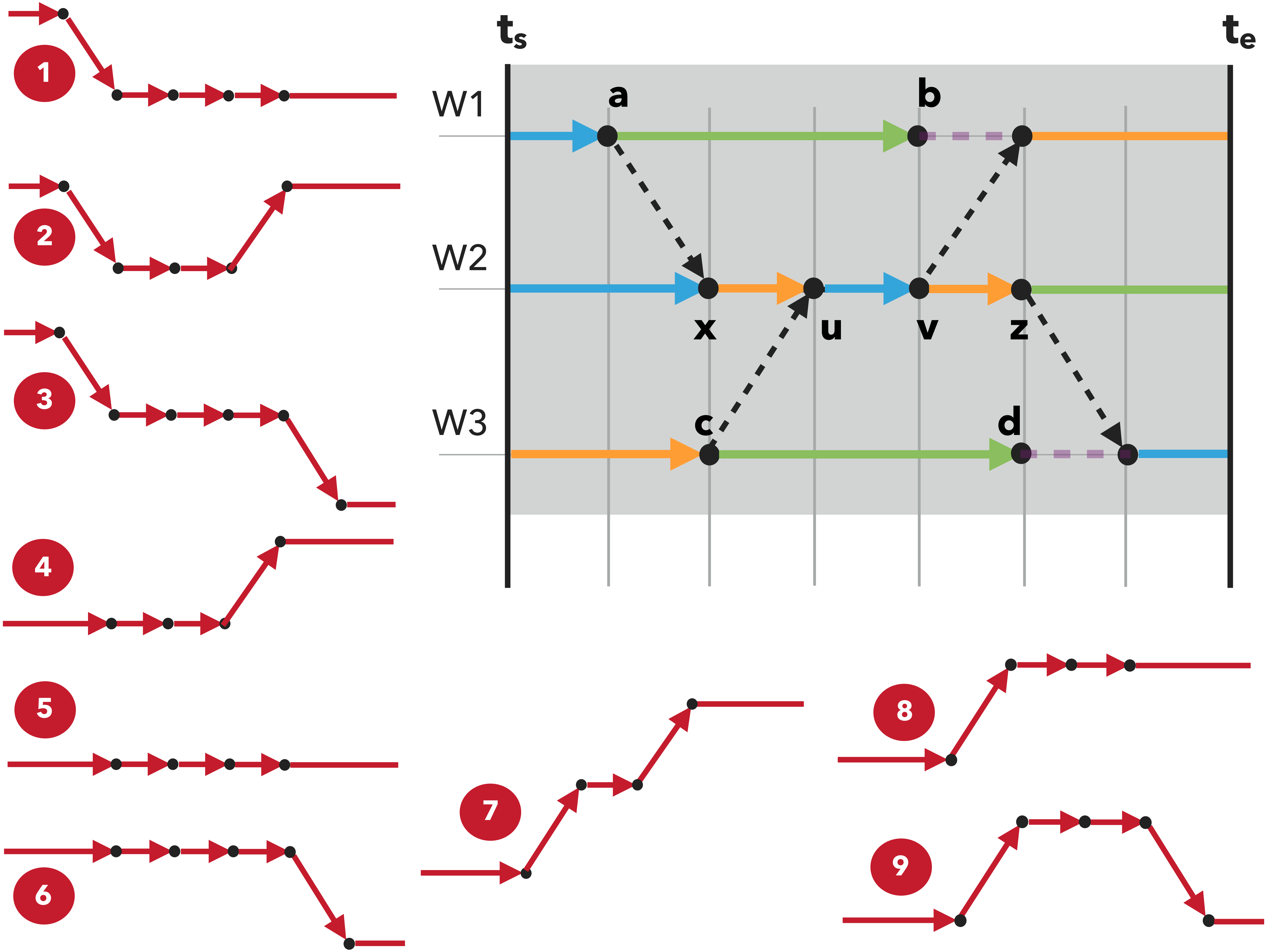
- All paths have the same length: $t_e - t_s$
- Choosing a random path might miss critical activities
- Enumerating all paths is impractical

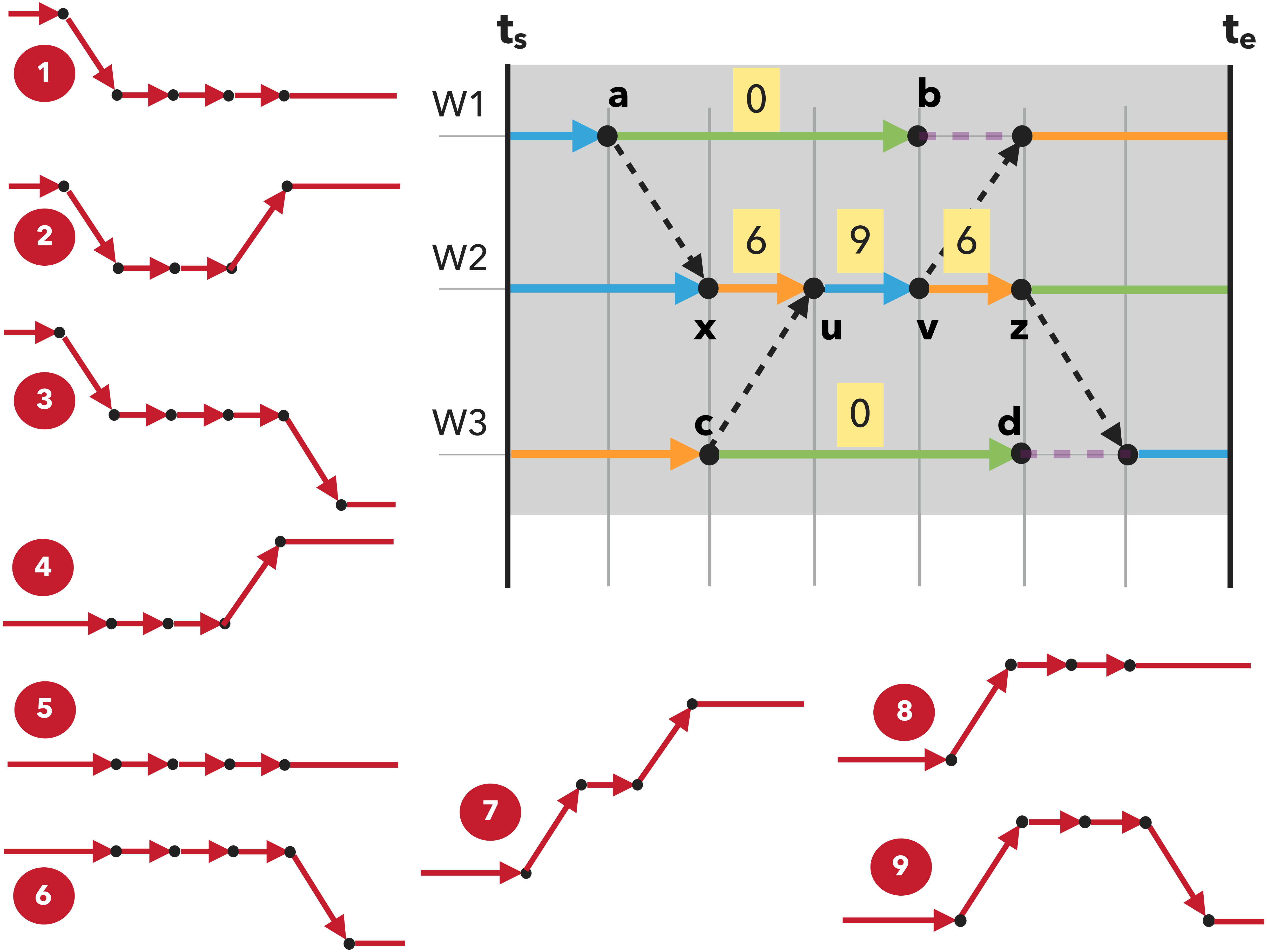
All paths are potentially part of the evolving critical path



How to rank activities with regard to criticality?

Intuition: the more paths an activity appears on the more probable it is that this activity is critical


t_s
t_e
W1
W2
W3
a
b
x
u
v
z
c
d
1
2
3
4
5
6
7
8
9


1
2
3
4
5
6
7
8
9
ts
te
W1
W2
W3
a
b
x
u
v
z
c
d
0
6
9
6
0

CRITICAL PARTICIPATION (CP METRIC)

An estimation of the activity's participation in the critical path

centrality: the number of paths this activity appears on

activity duration: edge weight

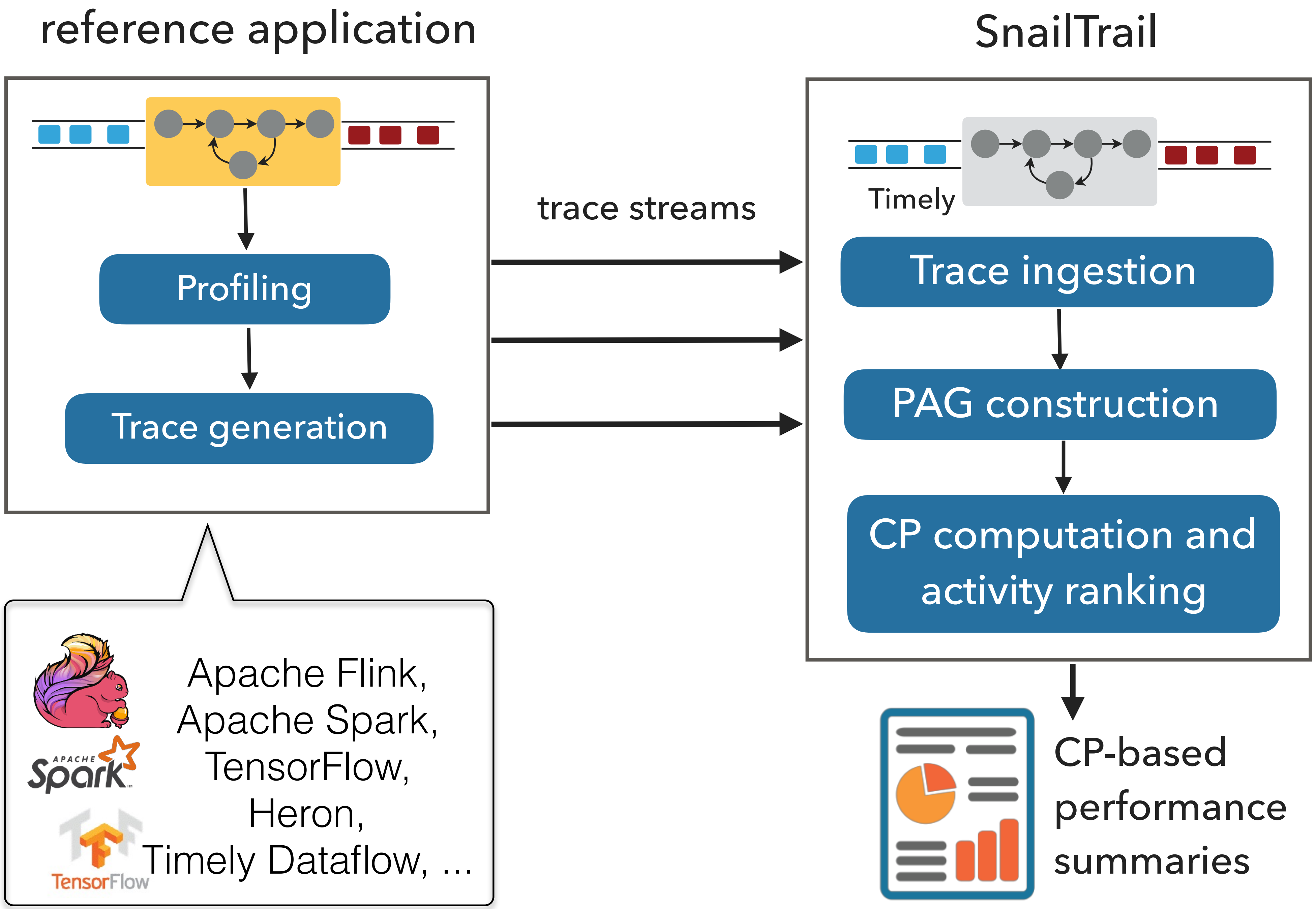
$$CP_a = \frac{C(a) \cdot a_w}{N(t_e - t_s)}$$

total number of paths in the snapshot

Can be computed without path enumeration!

ONLINE PERFORMANCE ANALYSIS WITH SNAILTRAIL

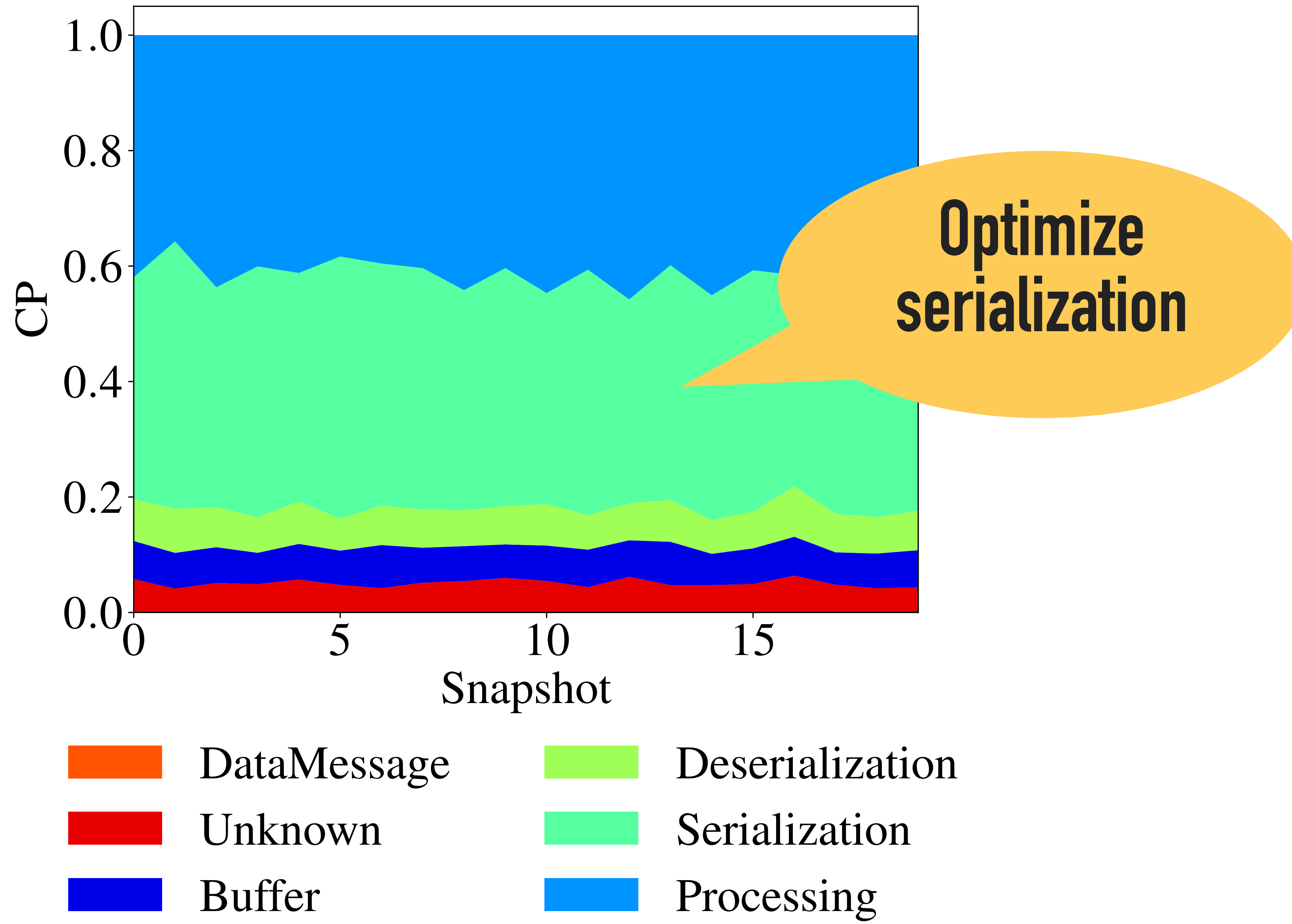
SNAILTRAIL IN ACTION



SNAILTRAIL CP-BASED SUMMARIES

- Activity Summary
  - *which **activity type** is a bottleneck?*

Activity Summary

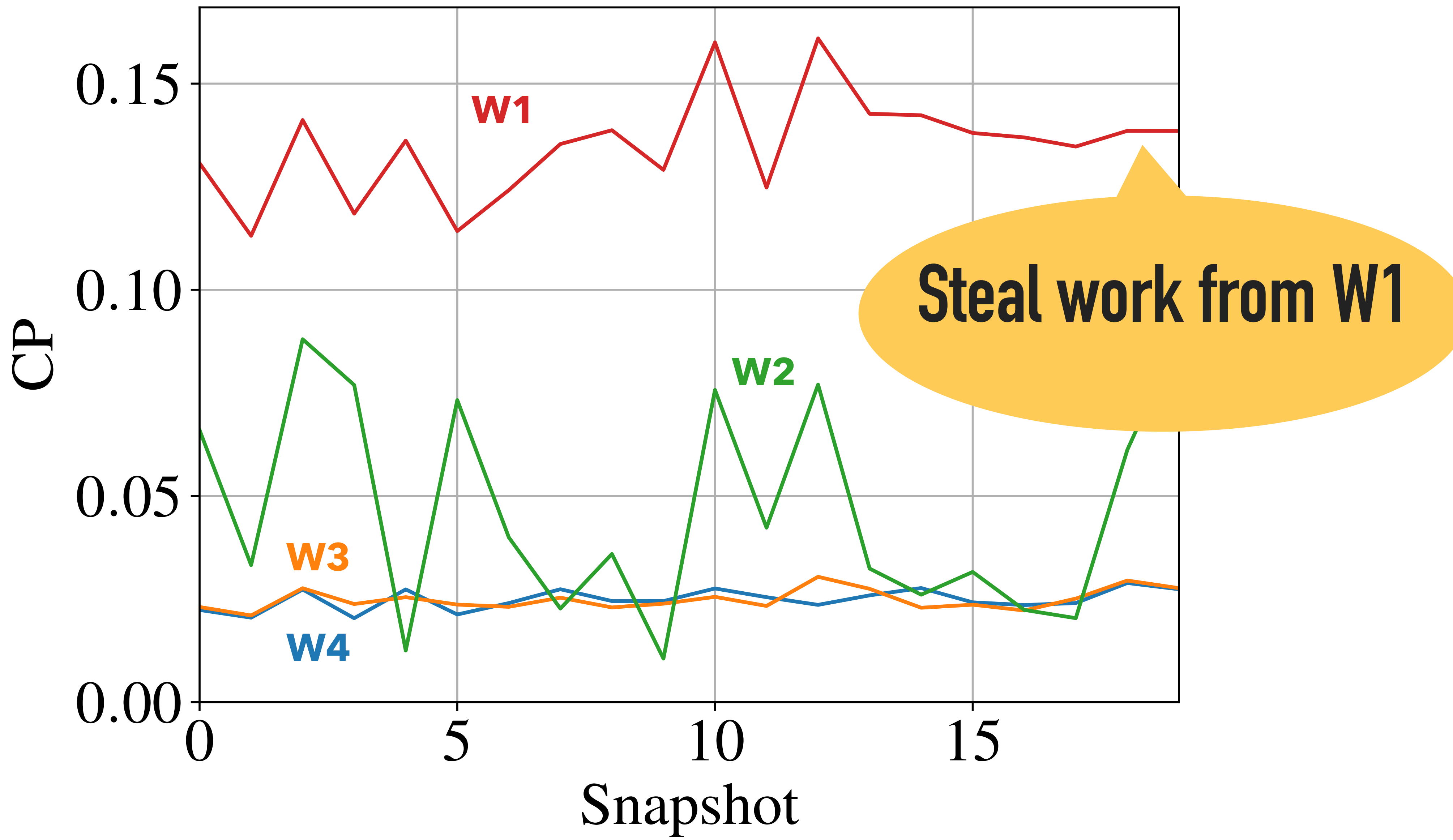


Apache Flink: Dhalion WordCount Benchmark, 4 workers, 1s snapshots

SNAILTRAIL CP-BASED SUMMARIES

- Activity Summary
  - *which **activity type** is a bottleneck?*
- Straggler Summary
  - *which **worker** is a bottleneck?*

Straggler Summary

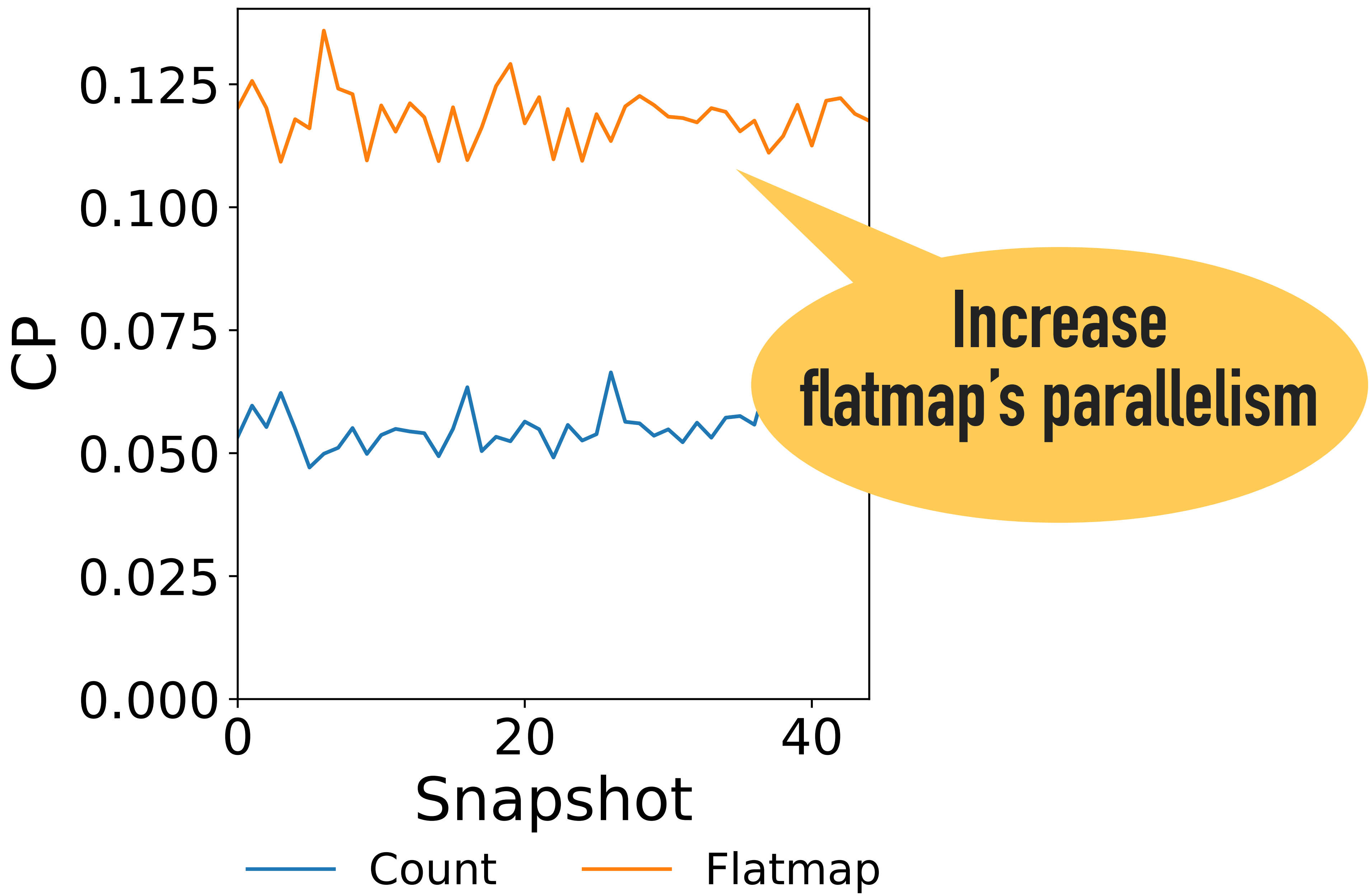


Apache Flink: Dhalion WordCount Benchmark, 4 workers, 1s snapshots

SNAILTRAIL CP-BASED SUMMARIES

- Activity Summary
  - *which **activity type** is a bottleneck?*
- Straggler Summary
  - *which **worker** is a bottleneck?*
- Operator Summary
  - *which **operator** is a bottleneck?*

Operator Summary

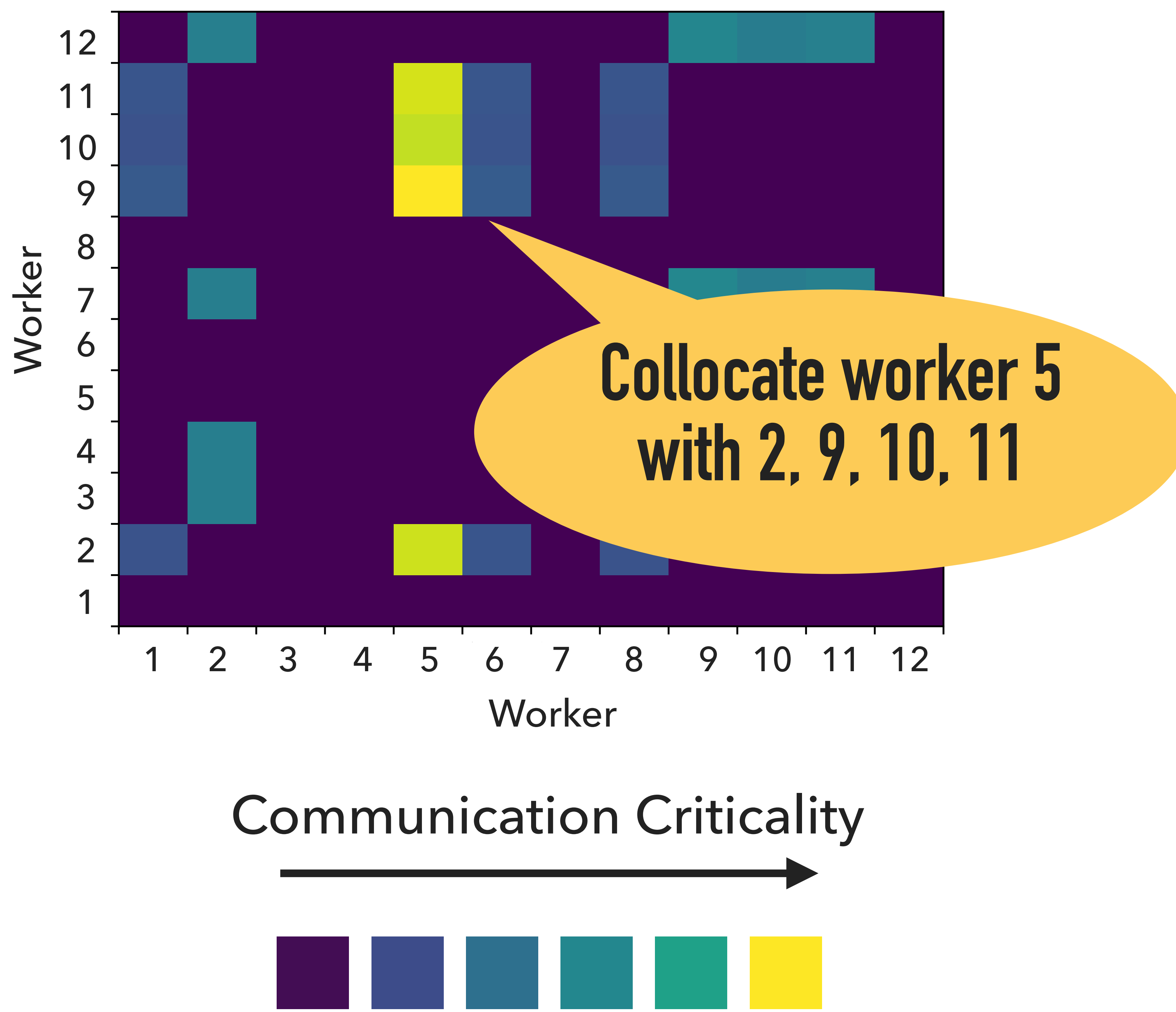


Apache Flink: Dhalion WordCount Benchmark, 10 workers, 1s snapshots

SNAILTRAIL CP-BASED SUMMARIES

- Activity Summary
  - *which **activity type** is a bottleneck?*
- Straggler Summary
  - *which **worker** is a bottleneck?*
- Operator Summary
  - *which **operator** is a bottleneck?*
- Communication Summary
  - *which communication **channels** are bottlenecks?*

Communication Summary



github.com/strymon-system/snailtrail

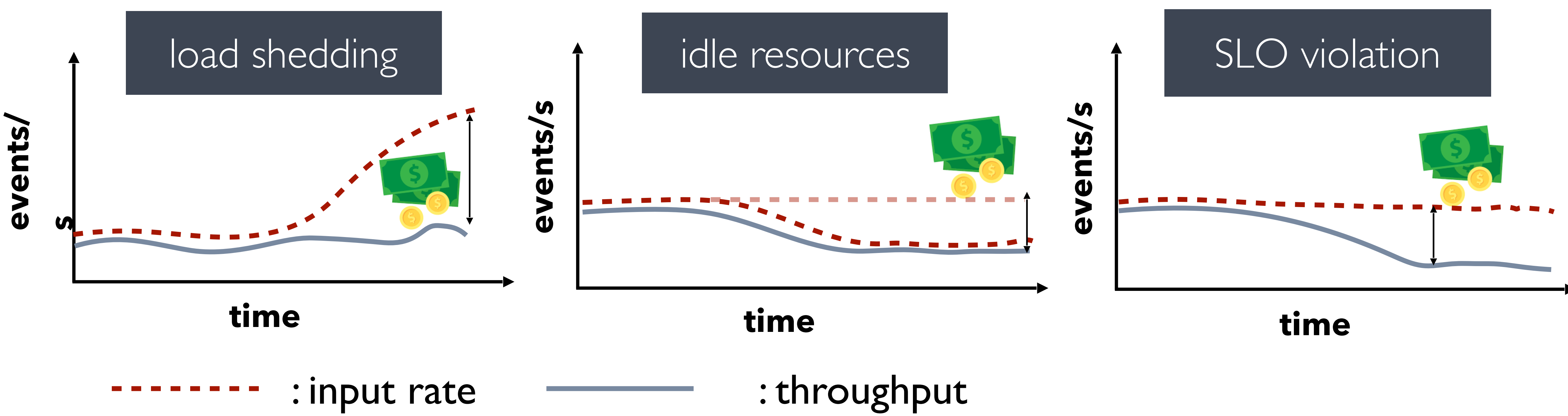
Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows

Vasiliki Kalavri, John Liagouris, Moritz Hoffmann,
Desislava Dimitrova, Matthew Forshaw, Timothy Roscoe

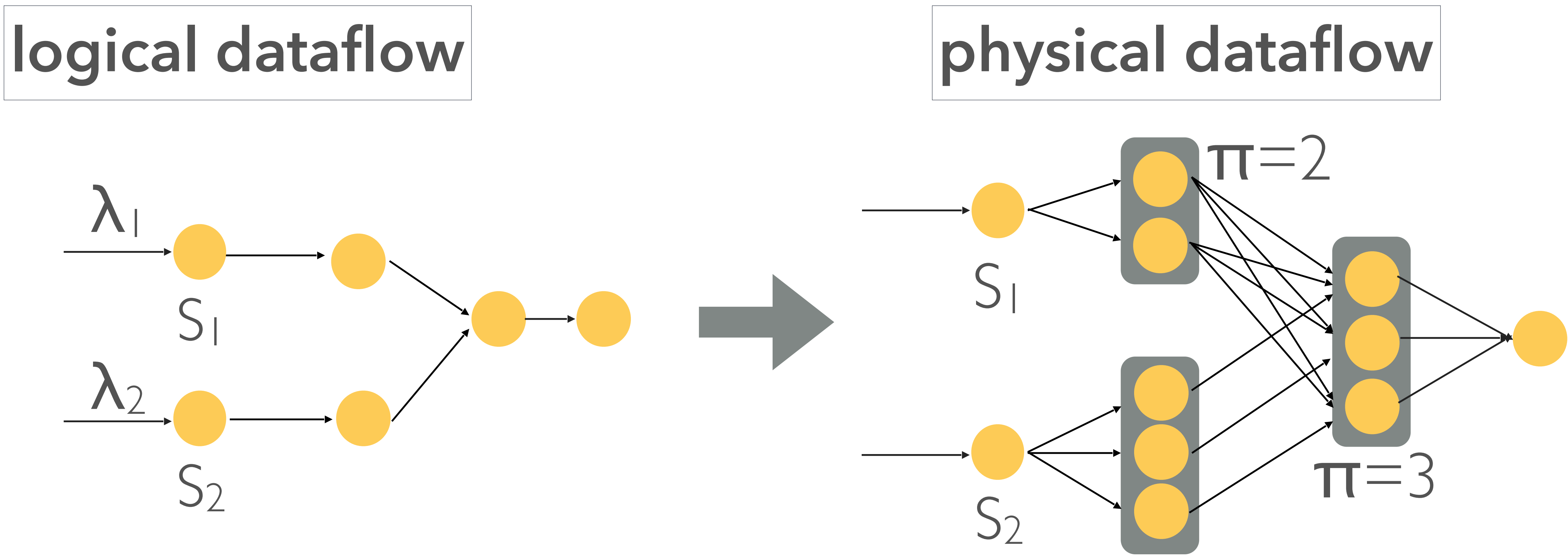
Systems Group, ETH Zurich

OSDI | 18

Any streaming job will inevitably become over- or under-provisioned in the future

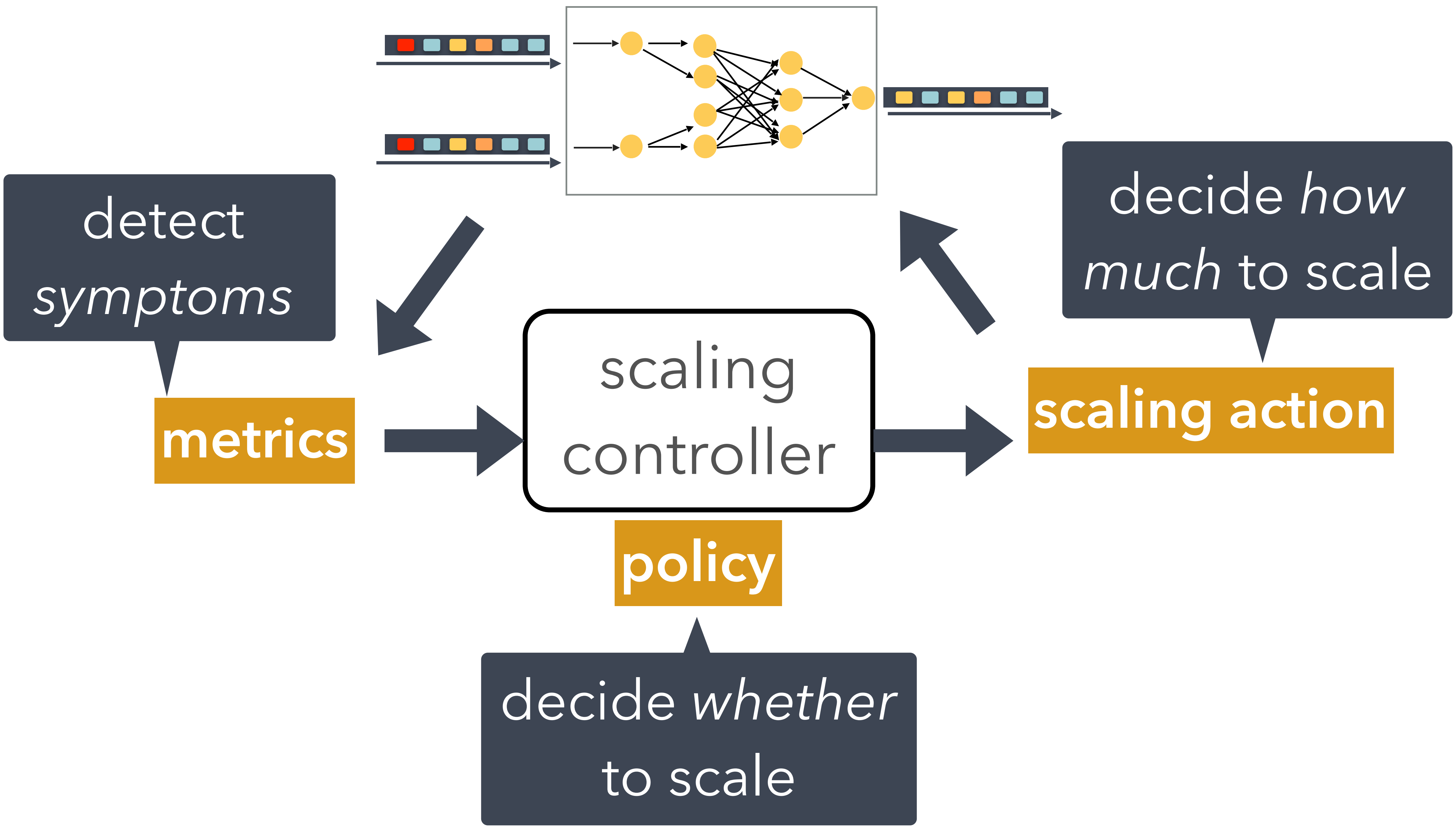


THE SCALING PROBLEM



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

AUTOMATIC SCALING OVERVIEW



EXISTING SCALING MODELS: QUEUING THEORY

- **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

EXISTING SCALING MODELS: CONTROL THEORY

- **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

EXISTING SCALING MODELS: RULES AND THRESHOLDS

- **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

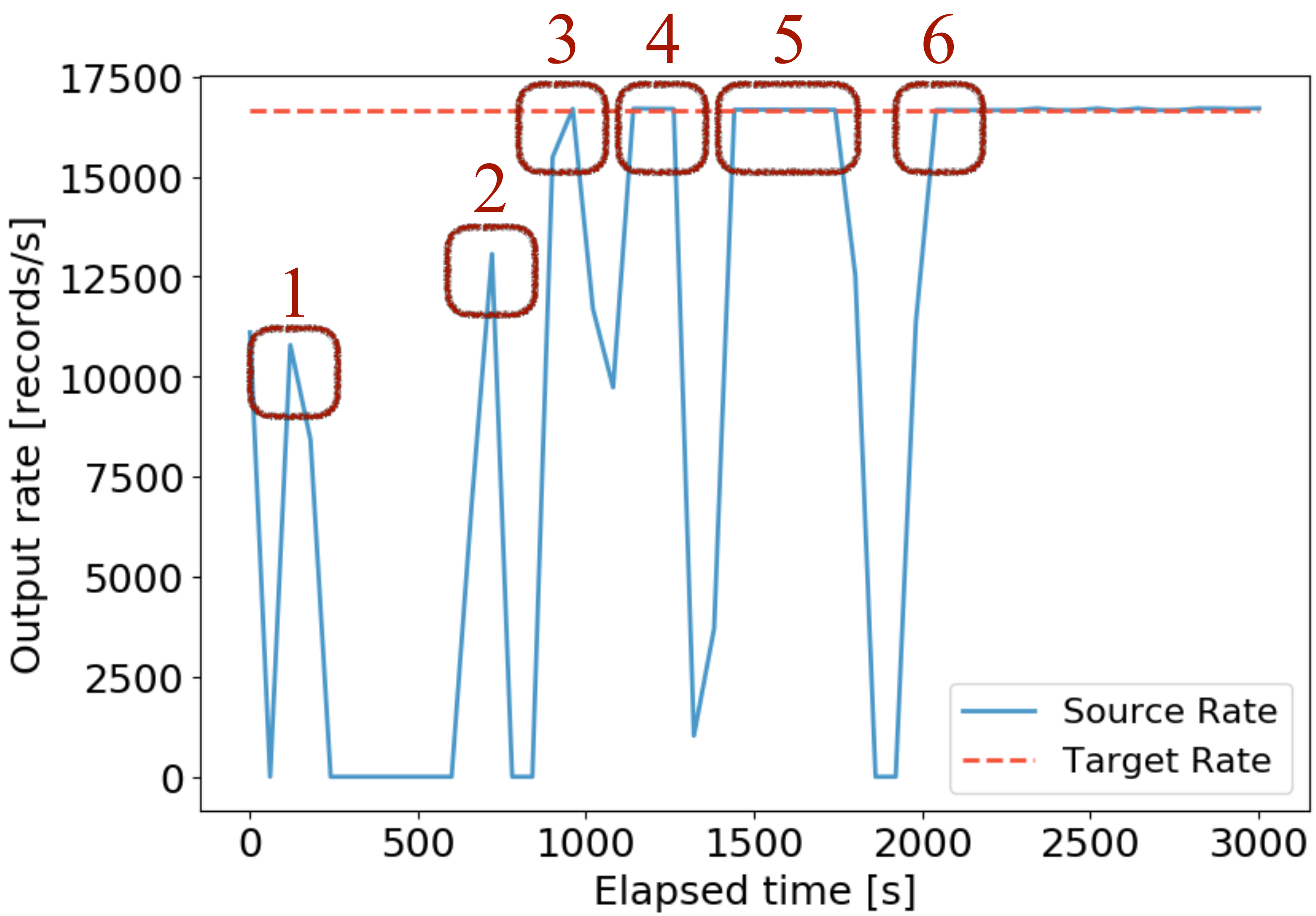
Noisy, sensitive to interference, misleading

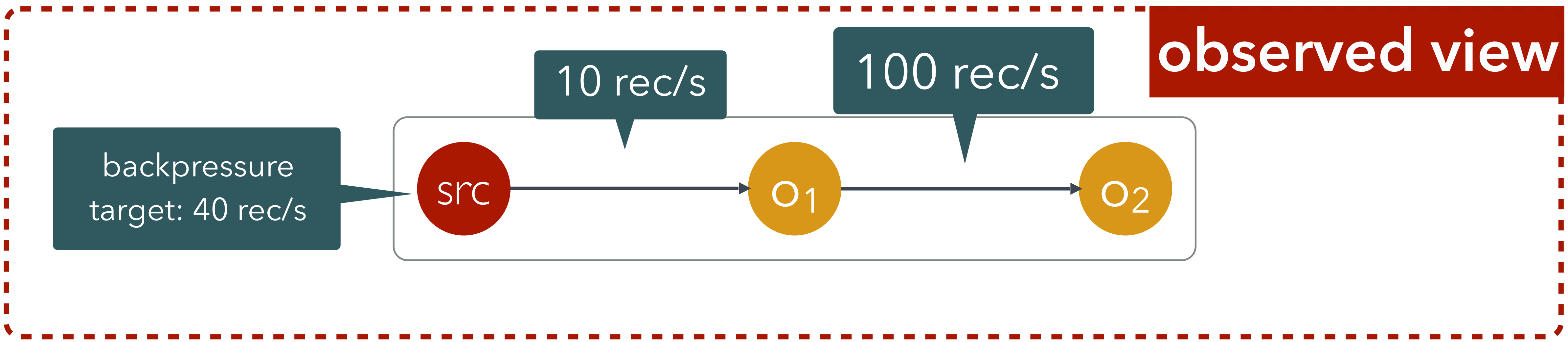
Easy-to-obtain

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

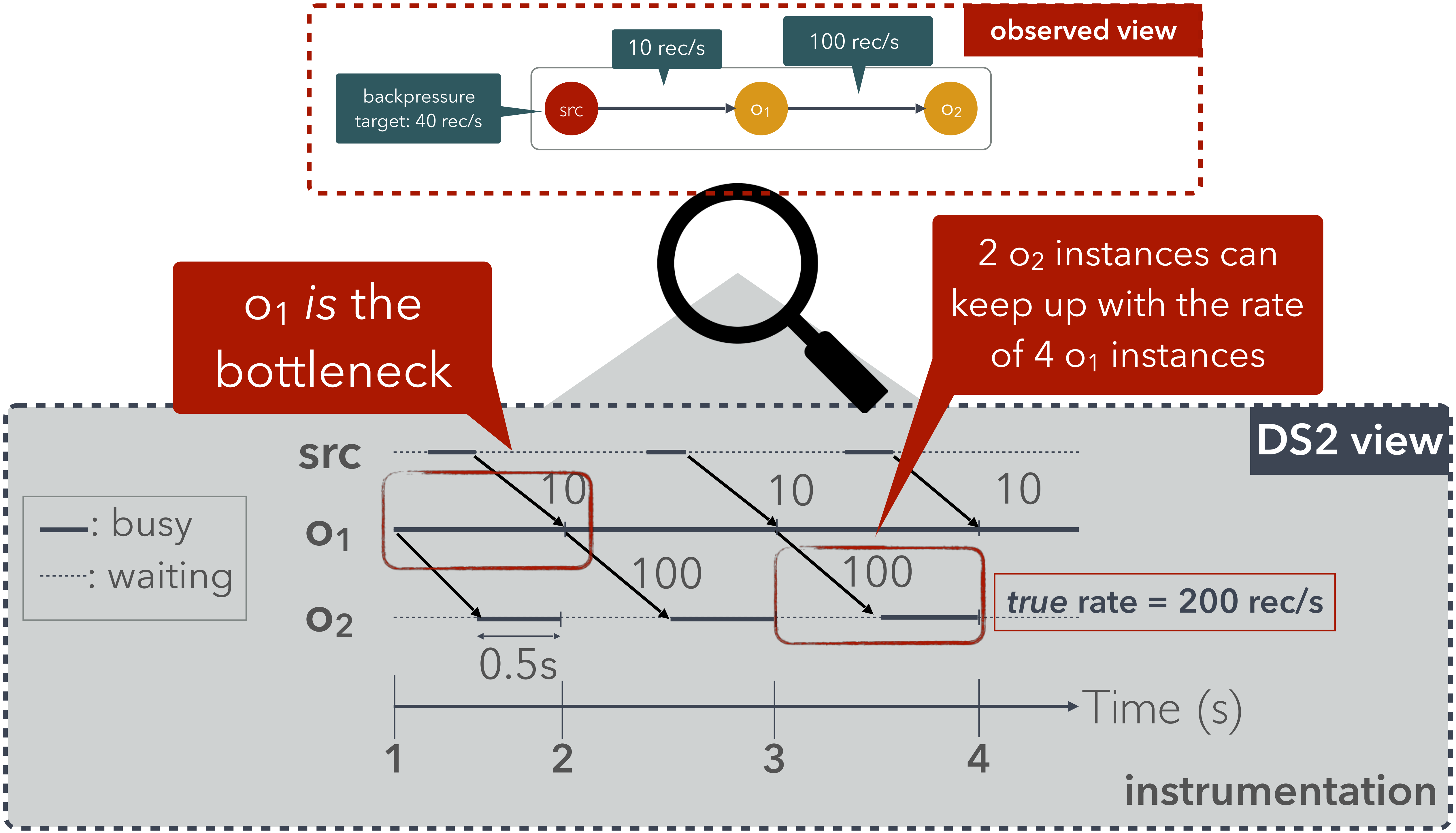




Which operator is the bottleneck?

What if we scale o_1 x 4?

How much to scale o_2?


observed view
10 rec/s
100 rec/s
backpressure
target: 40 rec/s
src
o1
o2
o1 is the bottleneck
2 o2 instances can keep up with the rate of 4 o1 instances
DS2 view
—: busy
······: waiting
src
o1
o2
10
10
10
100
100
true rate = 200 rec/s
0.5s
Time (s)
1
2
3
4
instrumentation

THE DS2 MODEL: INSTRUMENTATION AND DATAFLOW DEPENDENCIES

- 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 itself
  - 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

THE DS2 MODEL: USEFUL TIME

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**

THE DS2 MODEL: TRUE RATES

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

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

THE DS2 MODEL: OPTIMAL PARALLELISM

Optimal parallelism per operator

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

captures upstream operators

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

current parallelism of operator i

Recursively computed as:

True output rate of source j

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

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

DS2 MODEL PROPERTIES

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

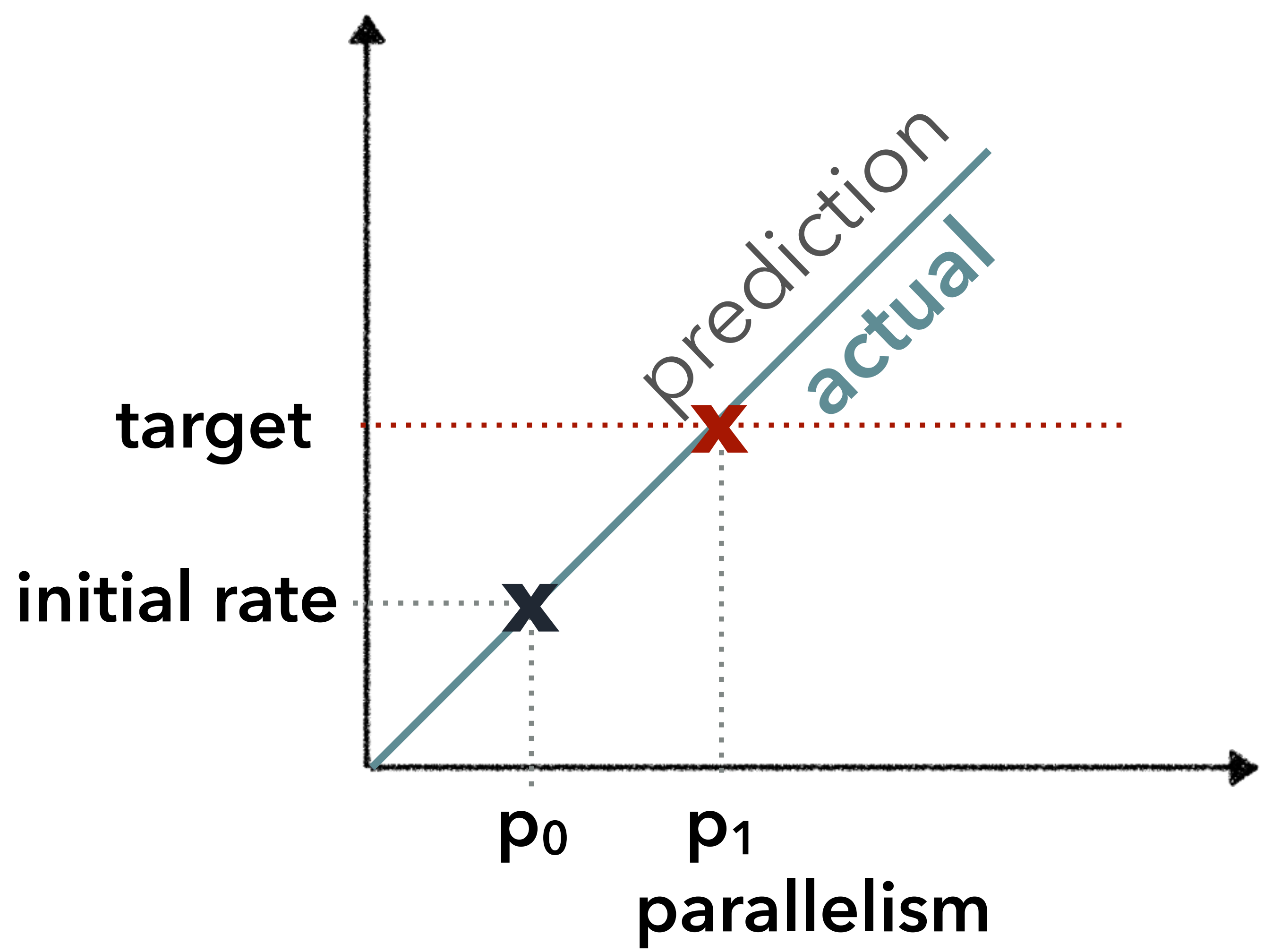
- scale-up does not cause over-provisioning (**no overshoot**)
- scale-down does not cause under-provisioning (**no undershoot**)

Ideal scaling acts as un **upper bound** when scaling up and as a **lower bound** when scaling down:

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

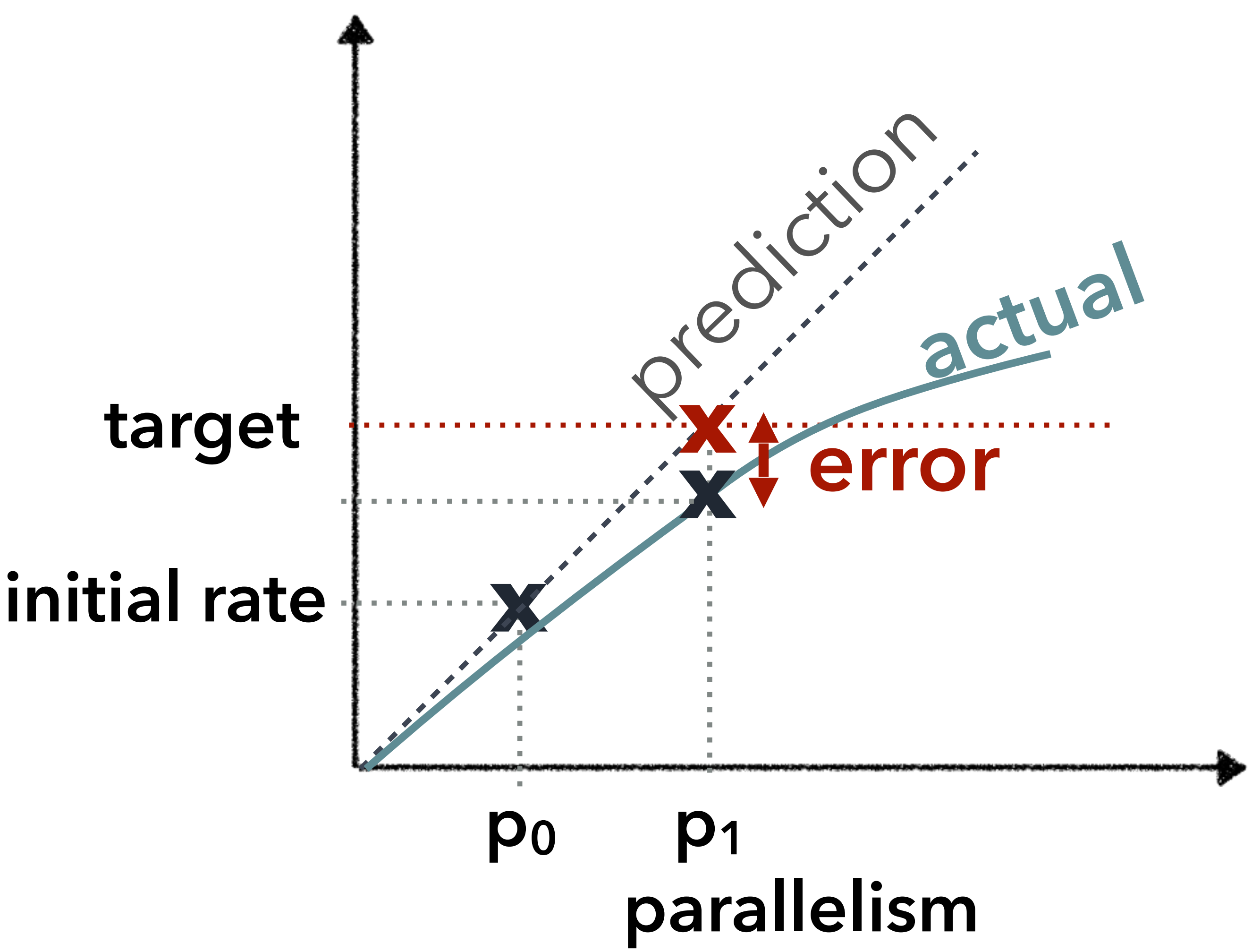
CONVERGENCE STEPS

If the actual scaling is **linear**, convergence takes **one** step



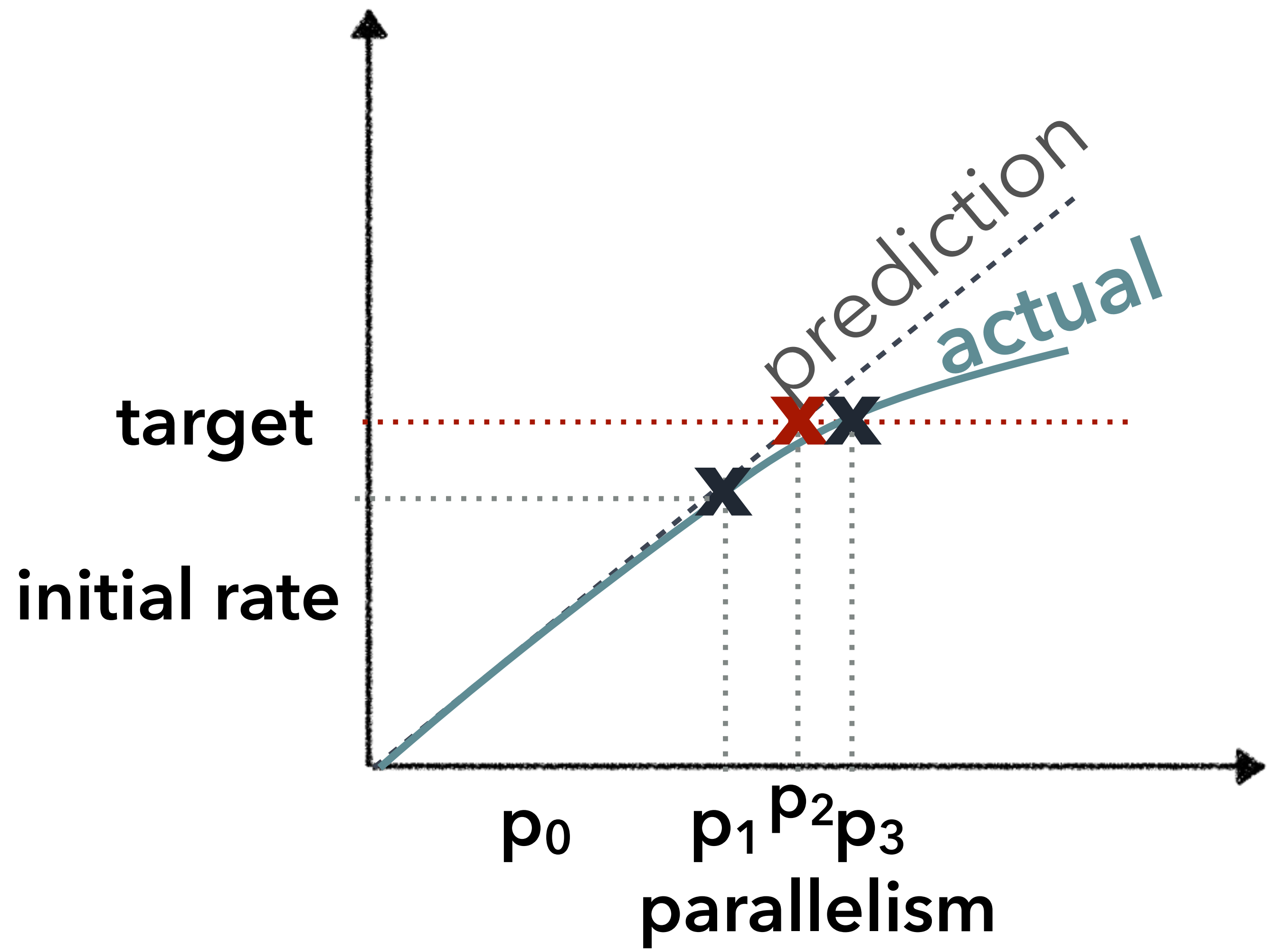
CONVERGENCE STEPS

when the actual scaling is **sub-linear**, convergence takes **more than one** steps

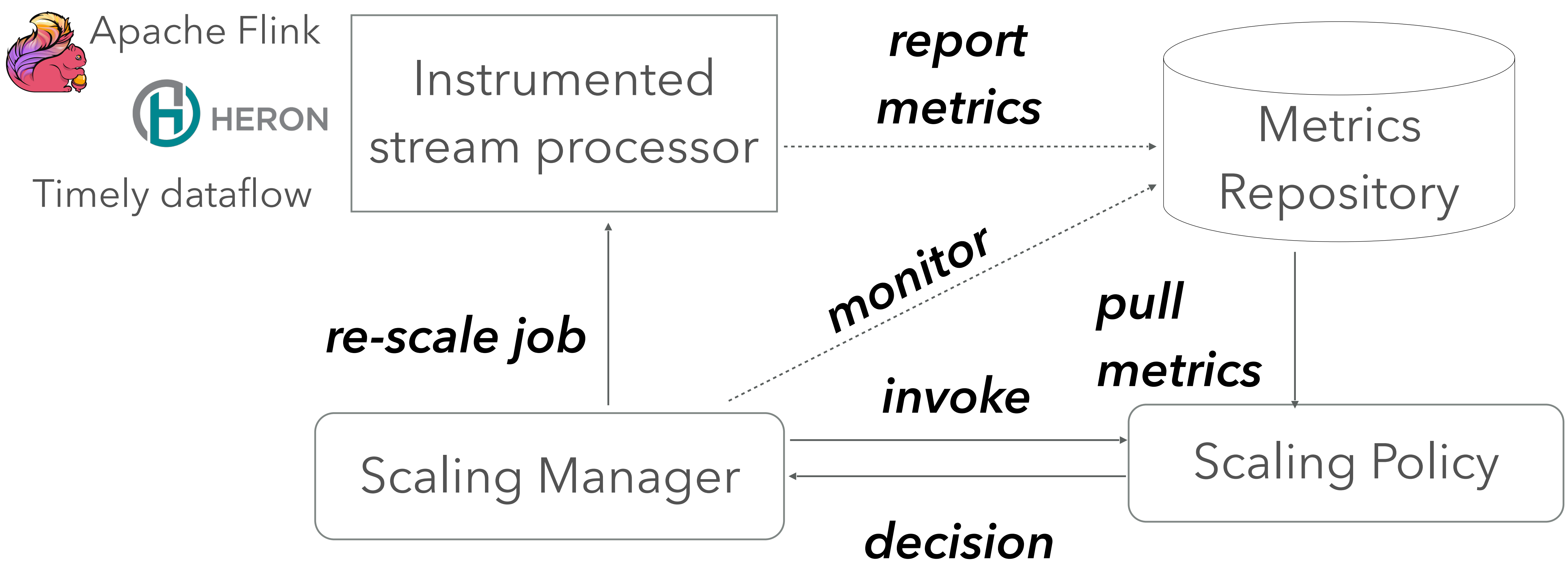


CONVERGENCE STEPS

In our experiments, DS2 took **up to three steps** to converge for complex queries.

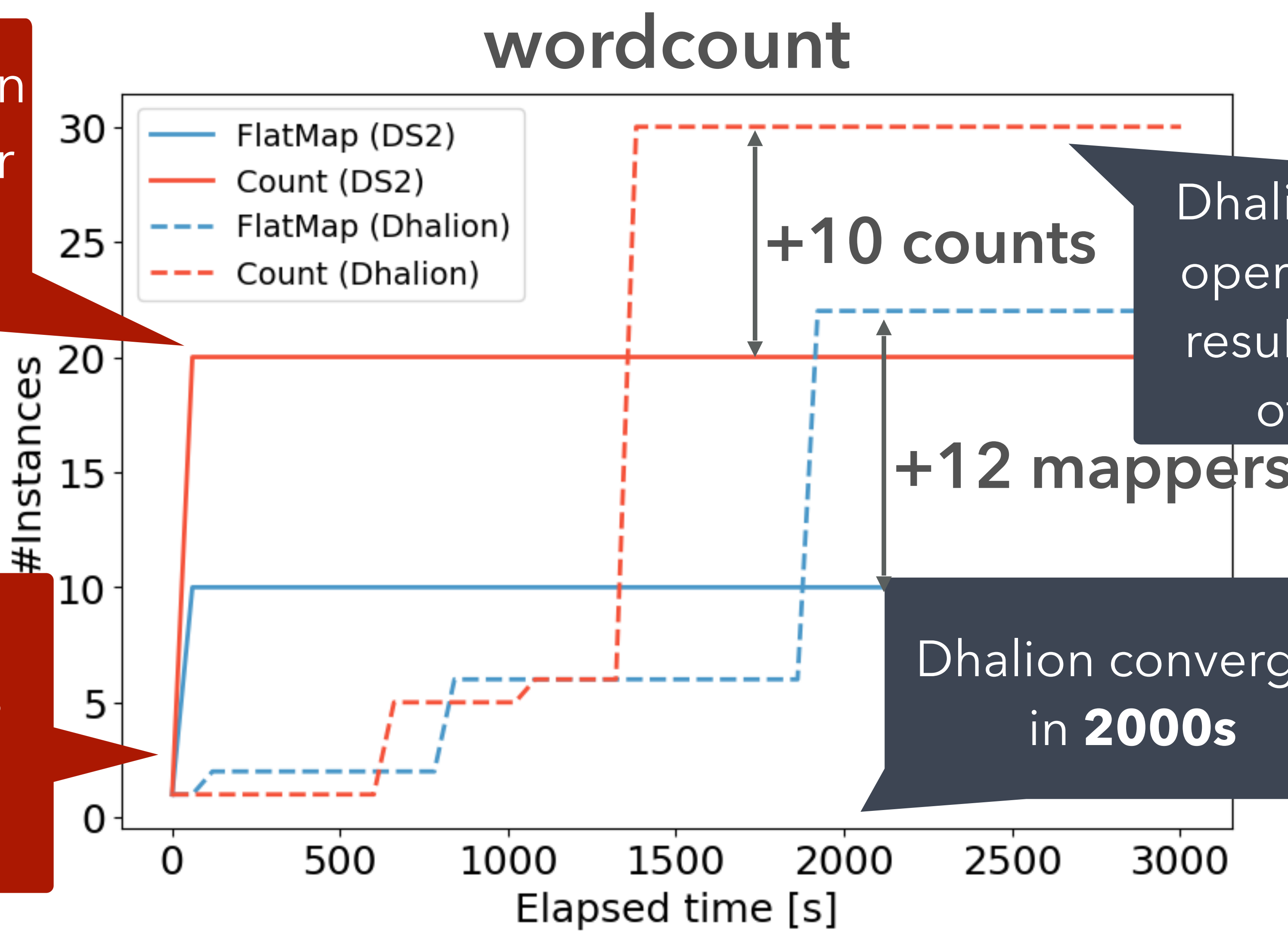


DS2 operates online in a reactive setting



DS2 VS. DHALION ON HERON

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



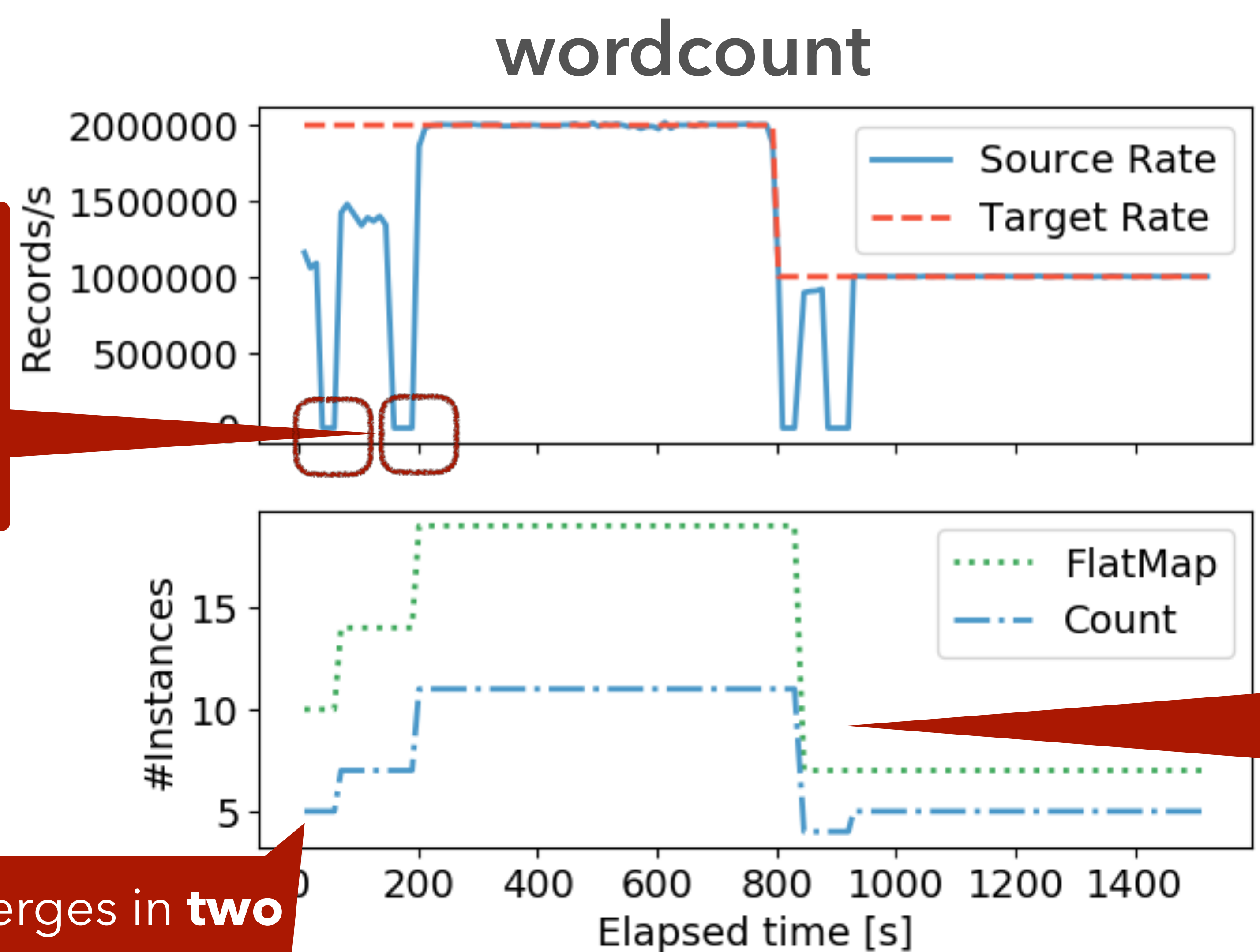
Dhalion scales one operator at a time, resulting to a total of **six steps**

DS2 converges in **60s**, i.e. as soon as it receives the Heron metrics

Dhalion converges in **2000s**

Target rate: 16.700 rec/s

DS2 ON APACHE FLINK



Apache Flink savepoint and reconfiguration takes ~**30s**

DS2 reacts **3s** after the target rate has changed

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

Target rate: 2.000.000 rec/s

CONVERGENCE – NEXMARK

at most **3 steps**

initial parallelism	Q1: flatmap	Q2: filter	Q3: incremental join	Q5: tumbling window join	Q8: sliding window	Q11: session window
8 =>	12 => **16**	11 => 13 => **14**	16 => **20**	14 => 15 => **16**	**10**	12 => 22 => **28**
12 =>	**16**	**14**	18 => **20**	**16**	**10**	22 => **28**
16 =>	**16**	12 => **14**	**20**	**16**	8 => **10**	26 => **28**
20 =>	**16**	13 => **14**	**20**	14 => **16**	8 => **10**	**28**
24 =>	**16**	**14**	**20**	14 => **16**	8 => **10**	**28**
28 =>	**16**	**14**	**20**	13 => **16**	8 => **10**	**28**

scale-up

scale-down

a single step for many queries and initial configurations

=> : scaling action

DS2 SUMMARY

metrics

externally observed

policy

threshold-based

scaling action

non-predictive, single-operator

true rates through instrumentation

dataflow *dependency* model

predictive, dataflow-wide actions

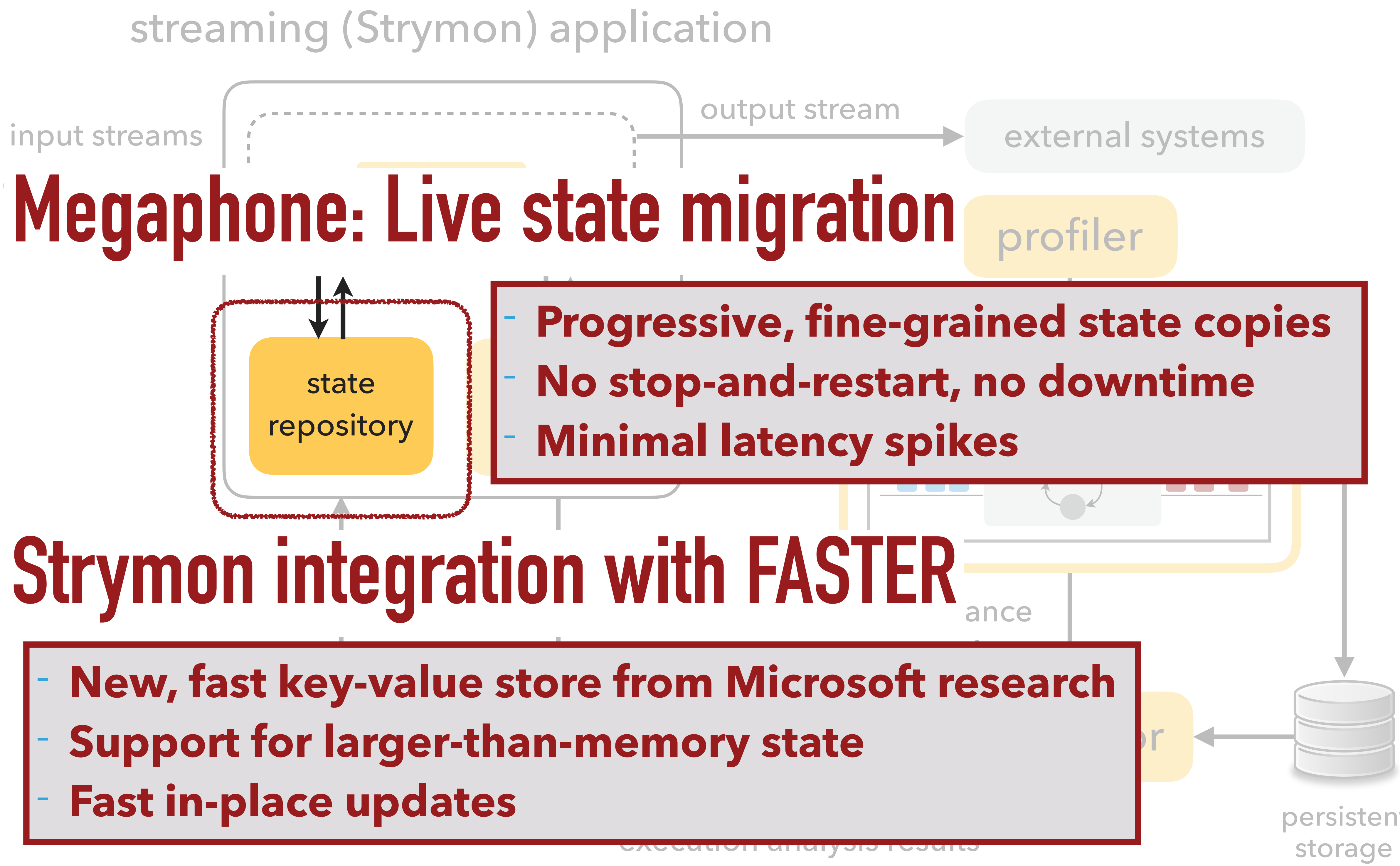
✓ no oscillations

✓ true rates as bounds to avoid over/under-shoot

✓ fast convergence

github.com/strymon-system/ds2

ONGOING WORK



Megaphone: Live state migration

- Progressive, fine-grained state copies
- No stop-and-restart, no downtime
- Minimal latency spikes

Strymon integration with FASTER

- New, fast key-value store from Microsoft research
- Support for larger-than-memory state
- Fast in-place updates

Towards self-managed, re-configurable streaming dataflow systems

Vasia Kalavri
kalavriv@inf.ethz.ch