Towards Performance Robustness for Microservices

Divyanshu Saxena, Gaurav Vipat, Jiaxin Lin, Jingbo Wang,
Işıl Dillig, Sanjay Shakkottai, Aditya Akella

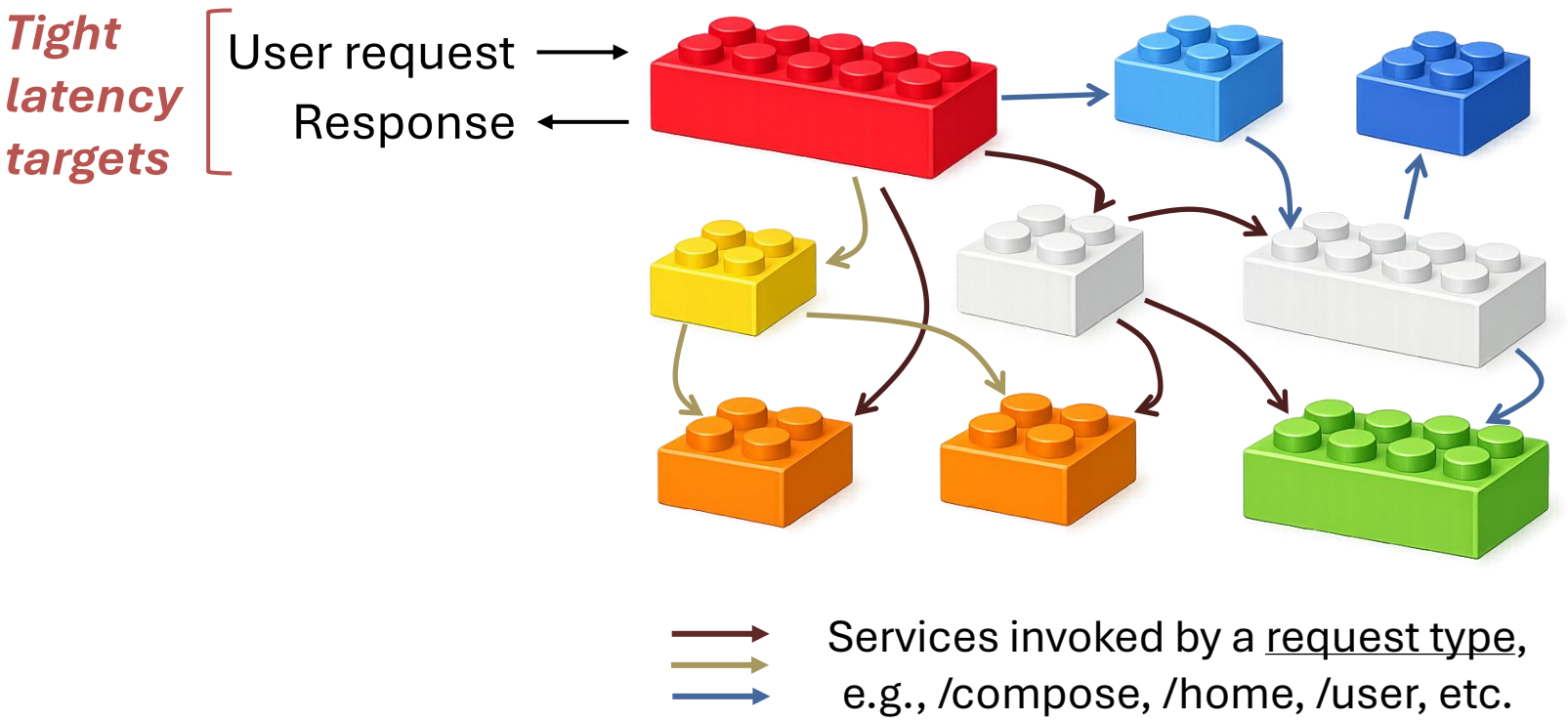
LDOS

TEXAS
The University of Texas at Austin

CORNELL
UNIVERSITY

PURDUE
UNIVERSITY

A Primer on Microservice Applications



Simplifies development but ensuring good performance becomes challenging.

Latent Factors Affect Microservice Performance

Sudden bursts in arrival rates
20% increase⟶1.7X p99 latency
3X arrival rates within 5 minutes*

Co-located background jobs
5.3X spike in p99 latency due to CPU contention

Concurrent request types
Shared microservices can become bottlenecks

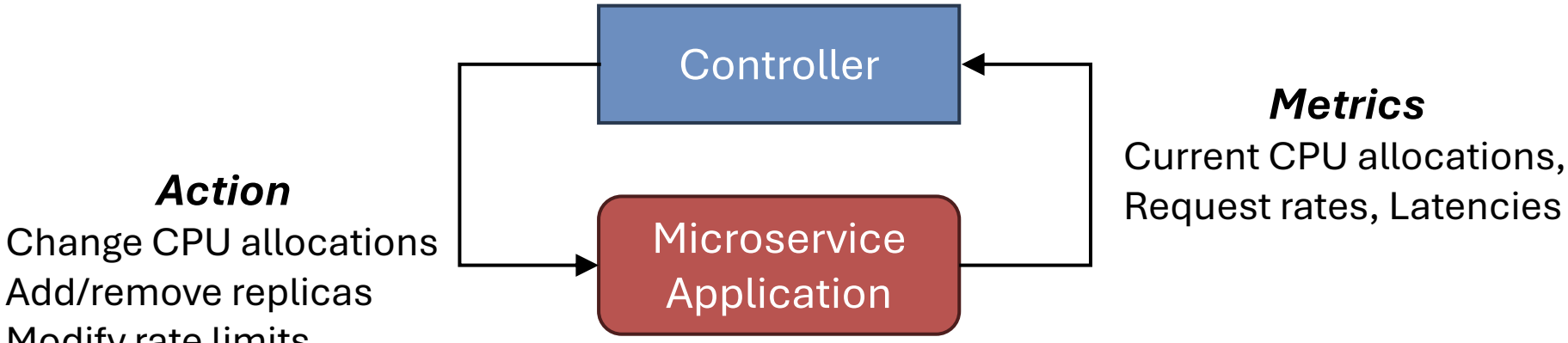
*Luo et al. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. SoCC'21

Latent Factors Affect Microservice Performance

Sudden bursts in arrival rates
20% increase⟶1.7X p99 latency
3X arrival rates within 5 minutes*

Co-located background jobs
5.3X spike in p99 latency due to CPU contention

Concurrent request types
Shared microservices can become bottlenecks



Controllers to manage microservices

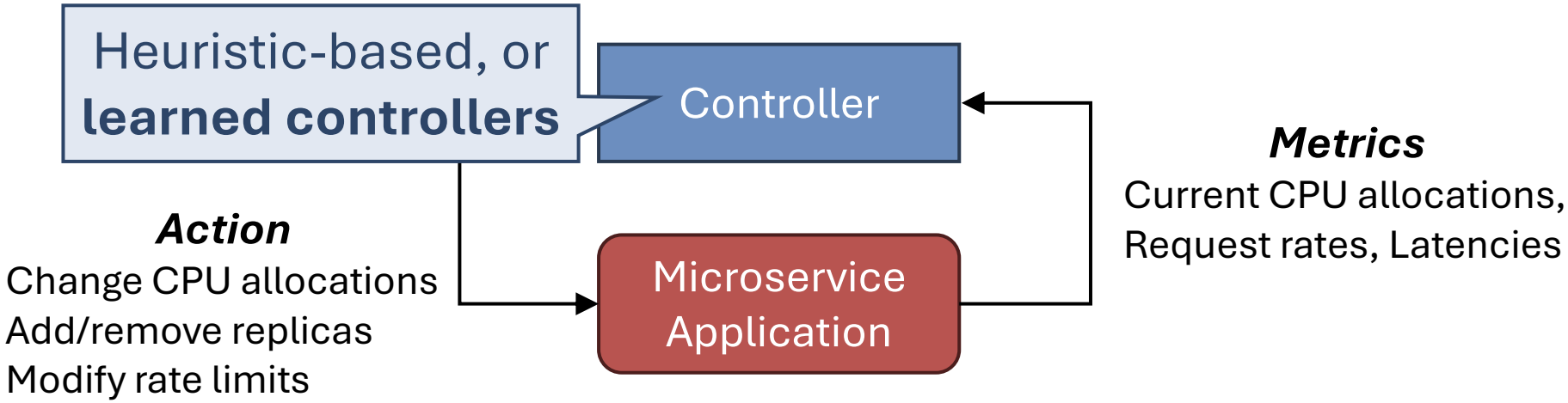
*Luo et al. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. SoCC'21

Latent Factors Affect Microservice Performance

Sudden bursts in arrival rates
20% increase⟶1.7X p99 latency
3X arrival rates within 5 minutes*

Co-located background jobs
5.3X spike in p99 latency due to CPU contention

Concurrent request types
Shared microservices can become bottlenecks

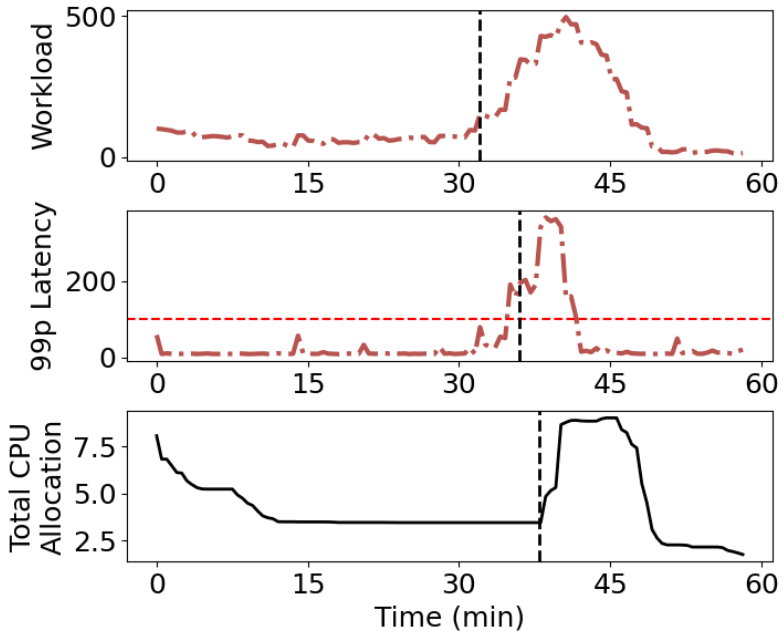


Controllers to manage microservices

*Luo et al. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. SoCC'21

Learned Controllers Fail to Address Latent Factors

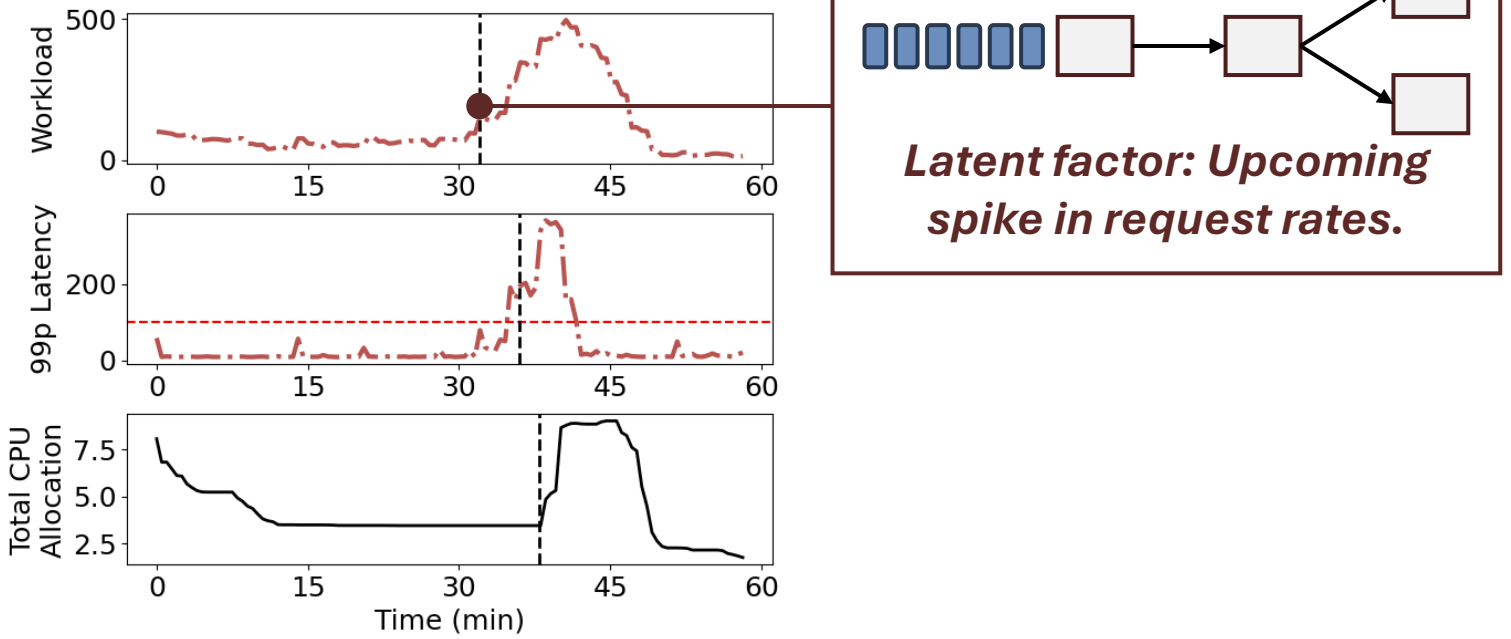
Autothrottle* uses current request rates and latencies to allocate CPU resources…
…but does not ramp up CPU allocations quickly enough.



*Wang et al. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. NSDI'24

Learned Controllers Fail to Address Latent Factors

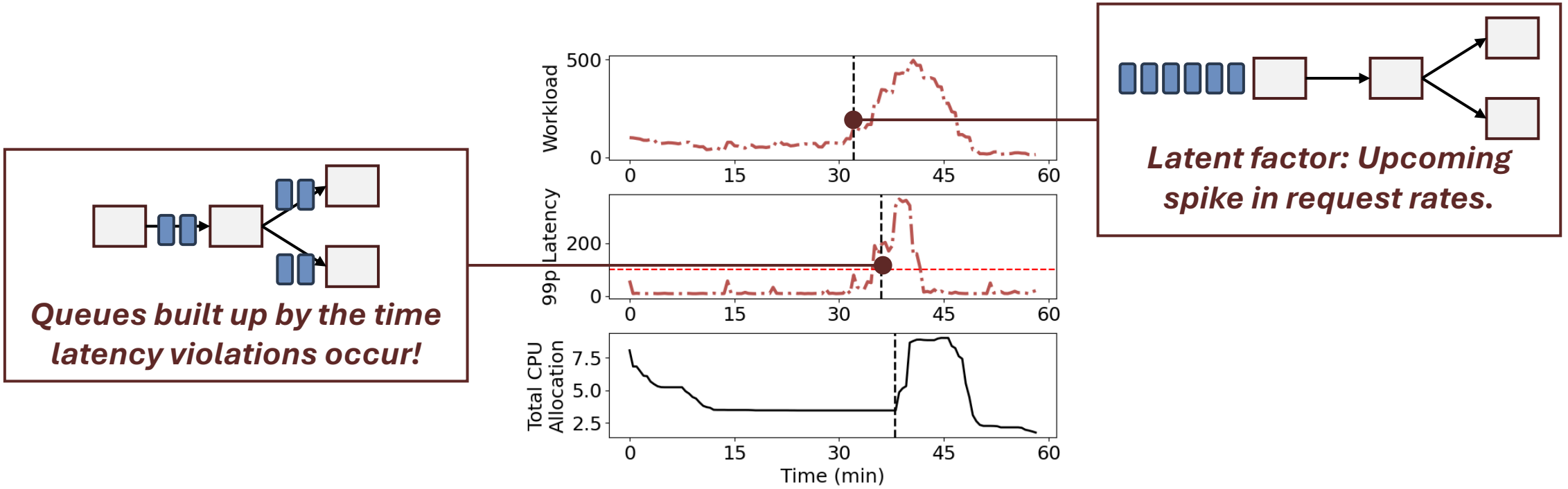
Autothrottle* uses current request rates and latencies to allocate CPU resources…
…but does not ramp up CPU allocations quickly enough.



*Wang et al. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. NSDI'24

Learned Controllers Fail to Address Latent Factors

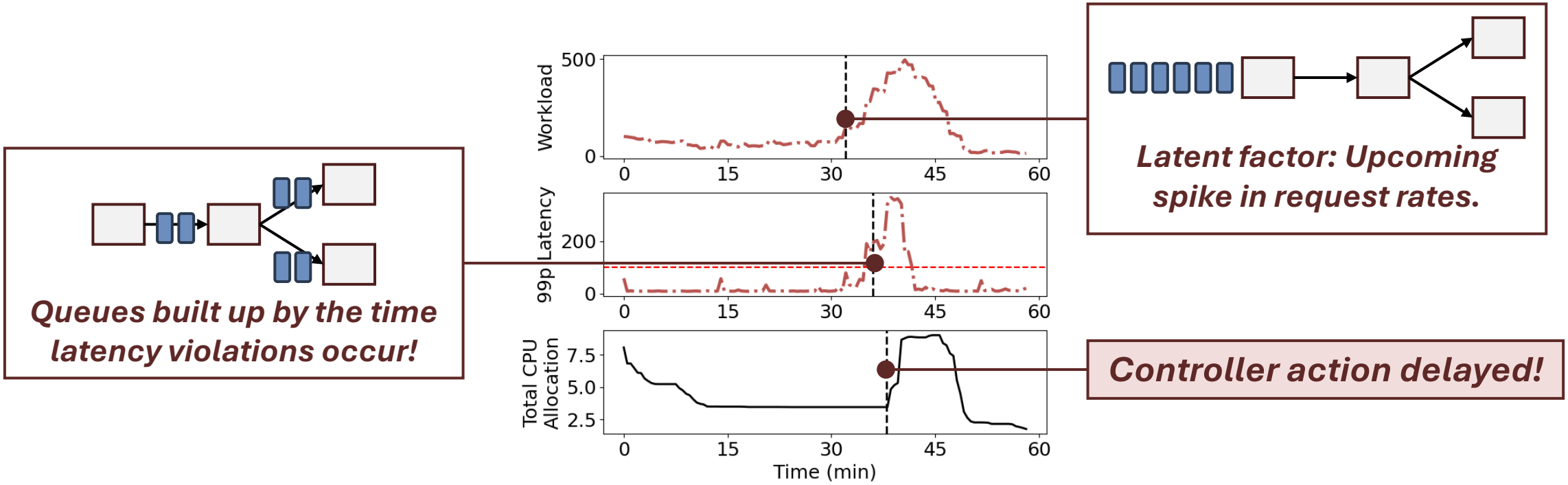
Autothrottle* uses current request rates and latencies to allocate CPU resources…
…but does not ramp up CPU allocations quickly enough.



*Wang et al. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. NSDI'24

Learned Controllers Fail to Address Latent Factors

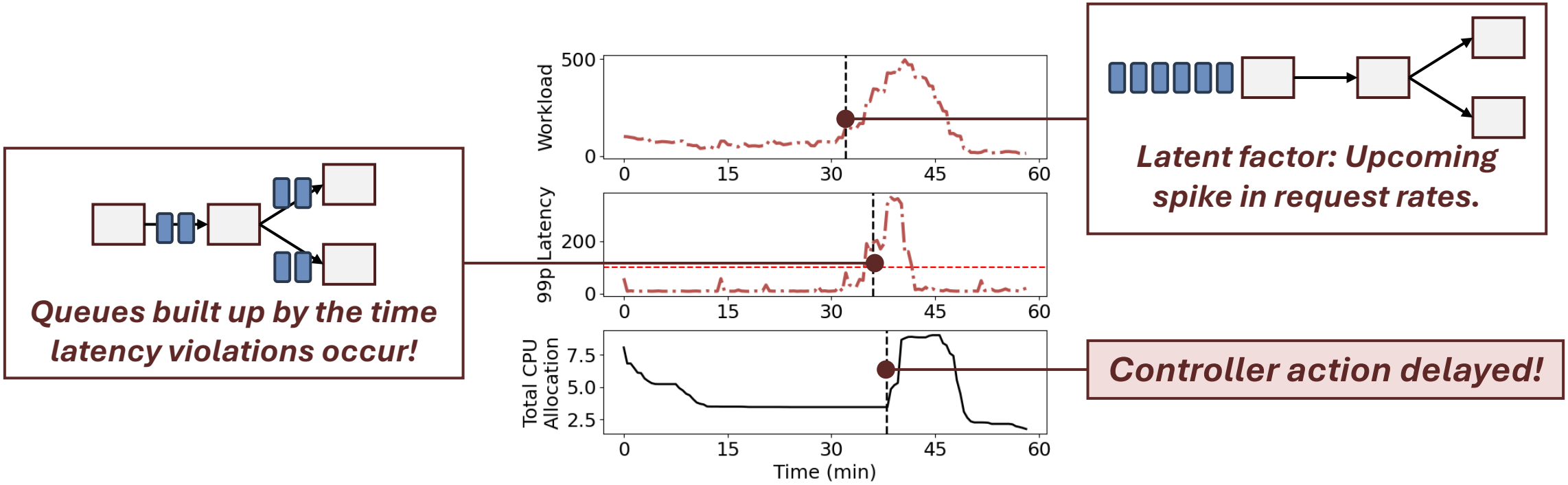
Autothrottle* uses current request rates and latencies to allocate CPU resources…
…but does not ramp up CPU allocations quickly enough.



*Wang et al. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. NSDI'24

Learned Controllers Fail to Address Latent Factors

Autothrottle* uses current request rates and latencies to allocate CPU resources…
…but does not ramp up CPU allocations quickly enough.



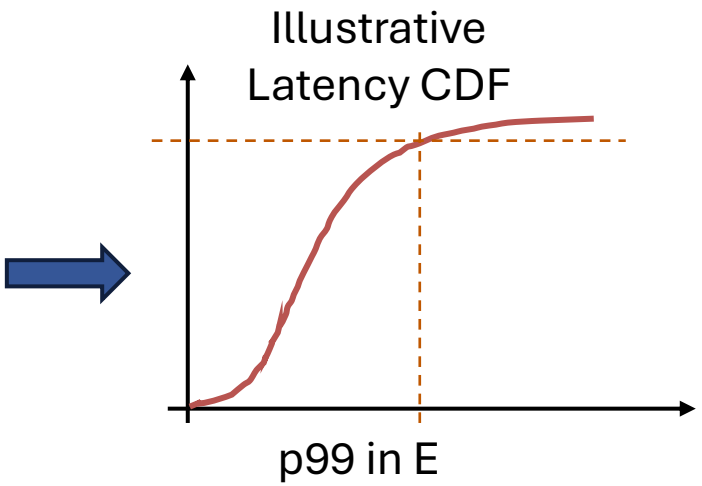
Controllers are unaware of how latent factors evolve and impact performance!

*Wang et al. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. NSDI'24

How to achieve ***performance robustness*** –
consistently meet latency targets despite latent factors?

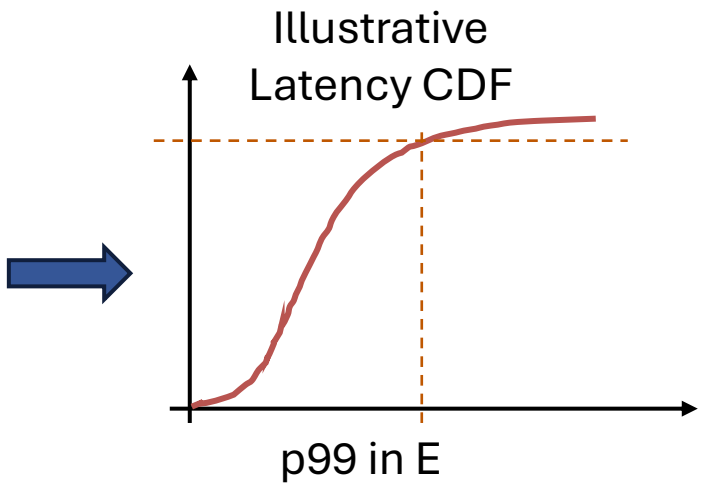
Key Idea: Anticipate Performance Under Perturbations

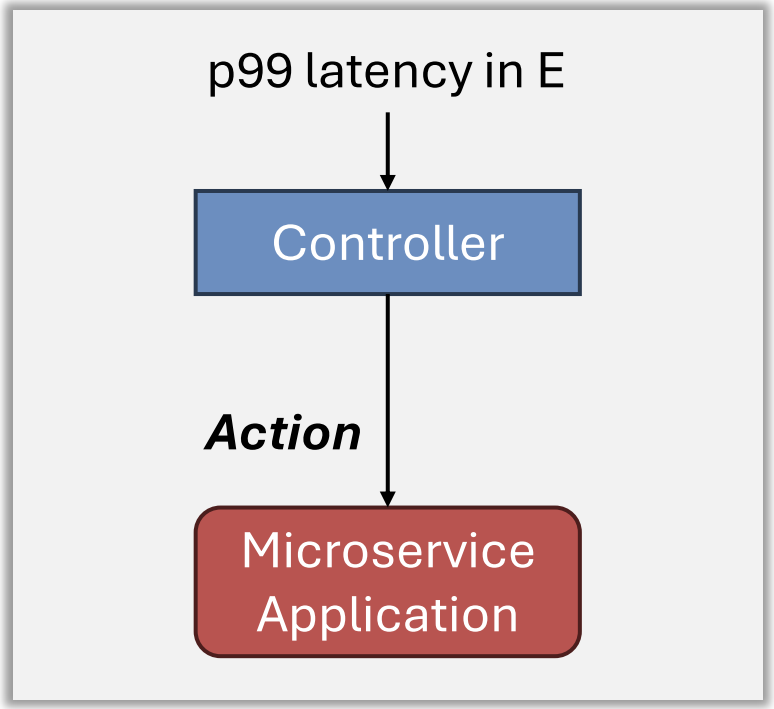
E
Current
Environment



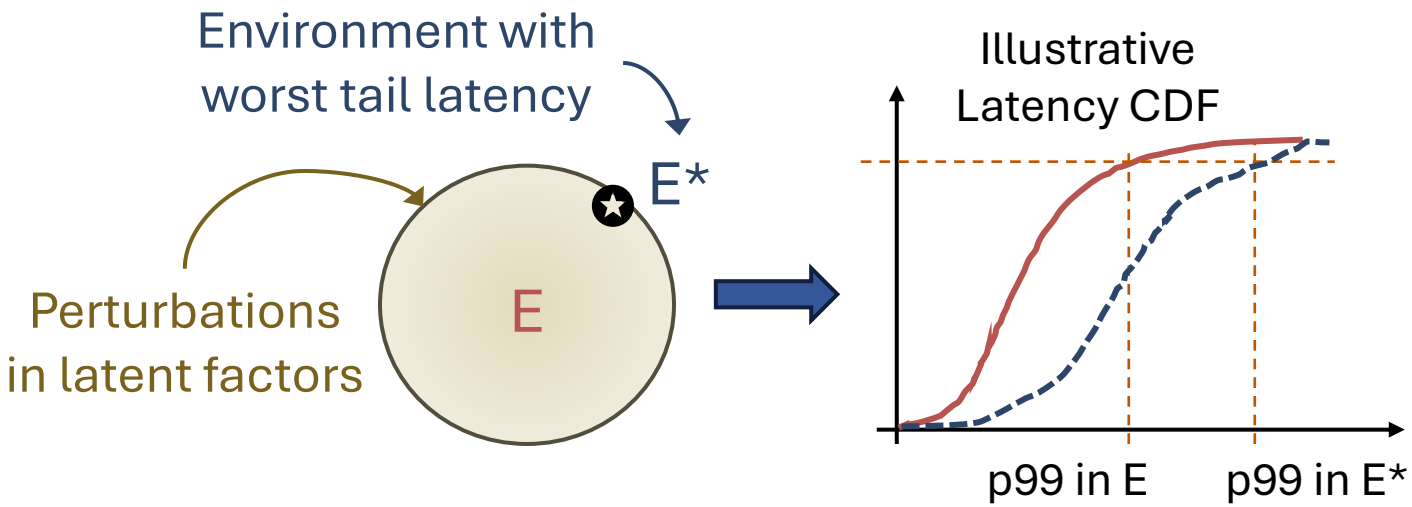
Key Idea: Anticipate Performance Under Perturbations

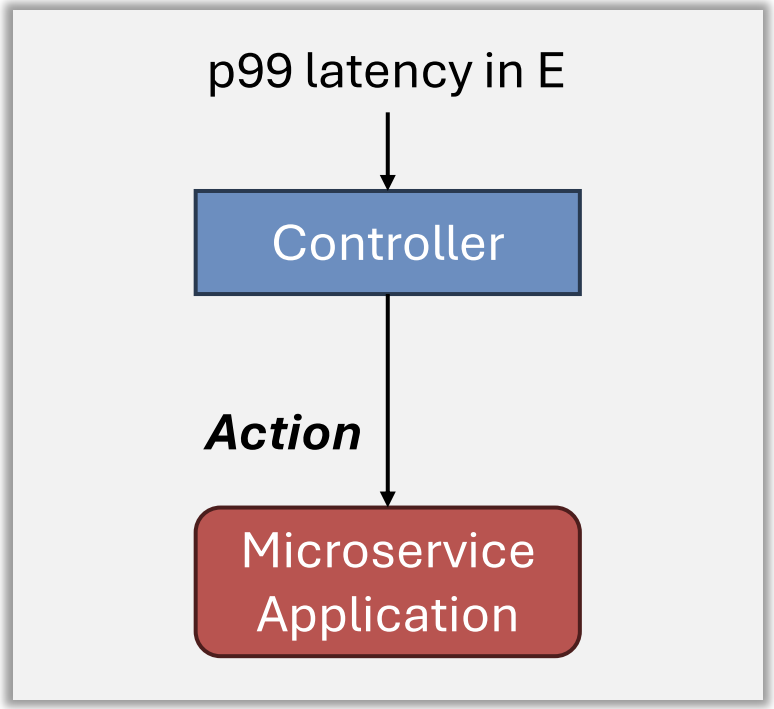
E
Current
Environment



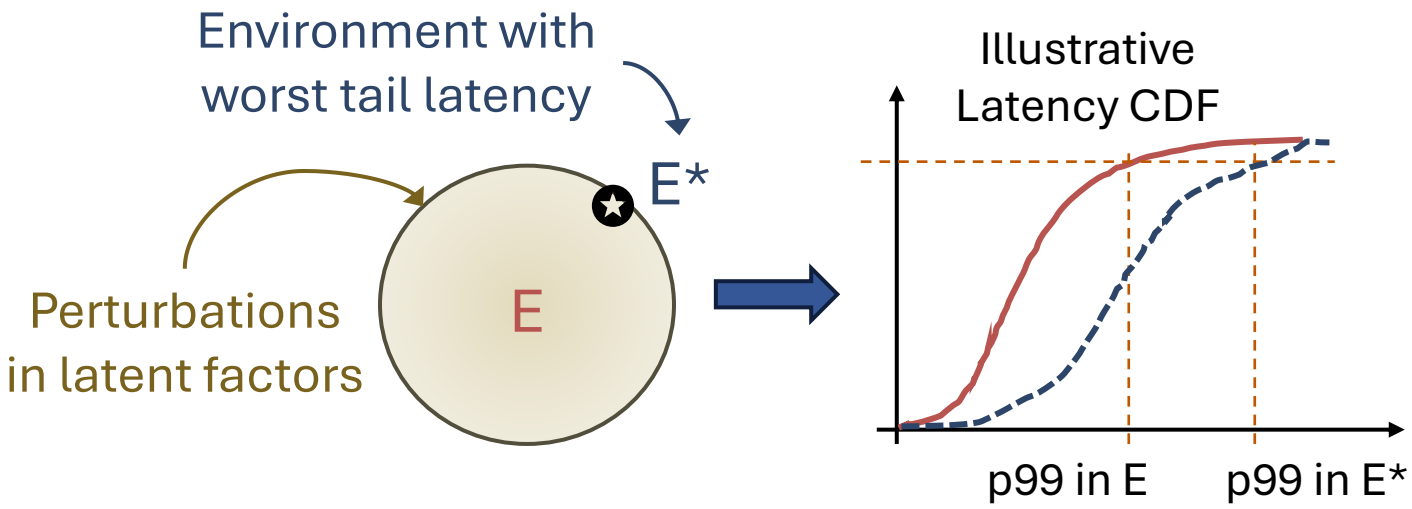


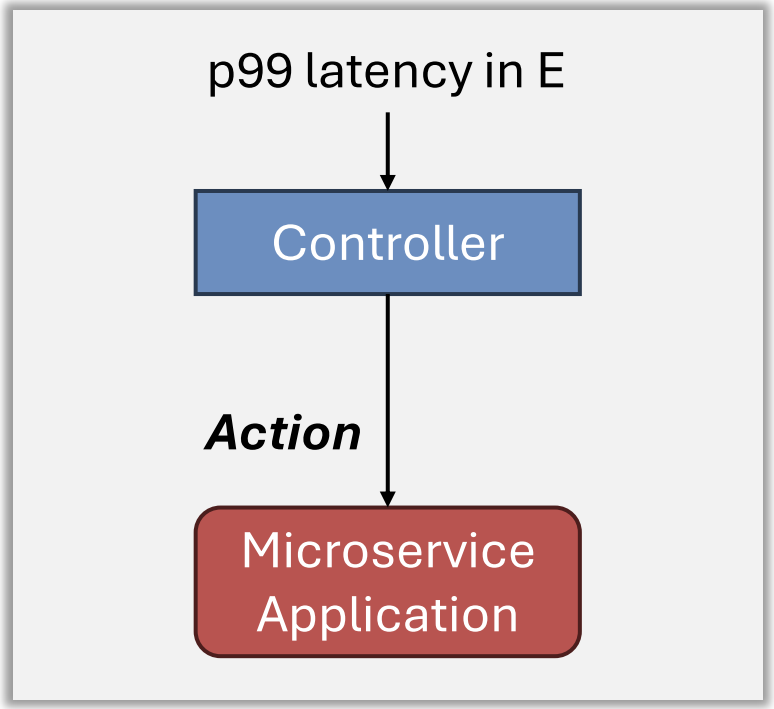
Key Idea: Anticipate Performance Under Perturbations





Key Idea: Anticipate Performance Under Perturbations



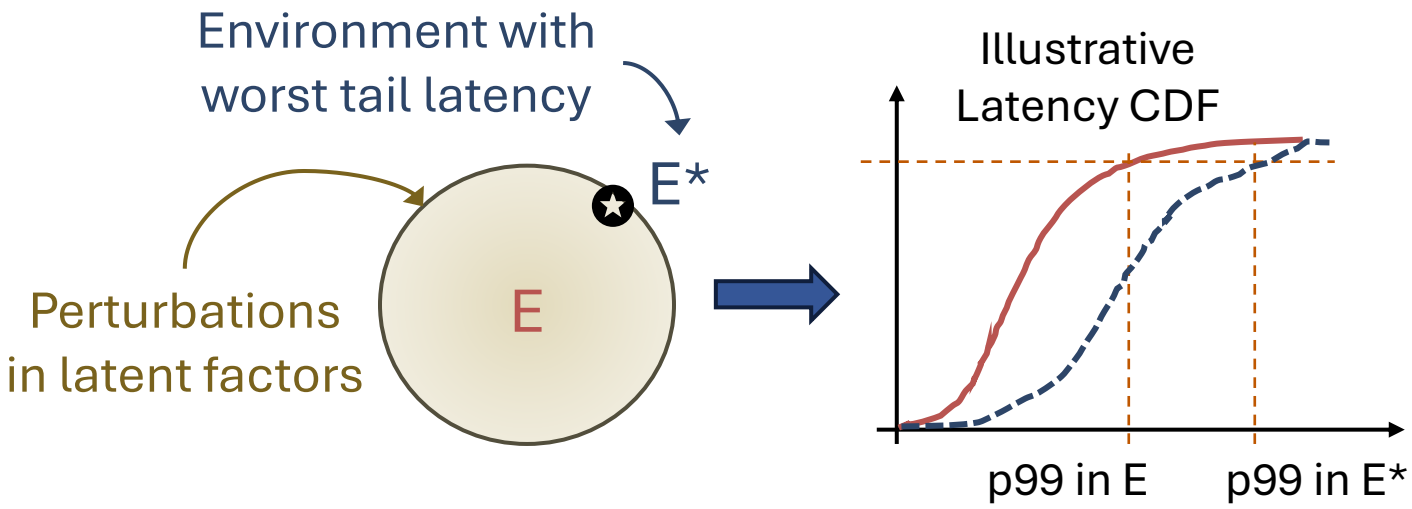


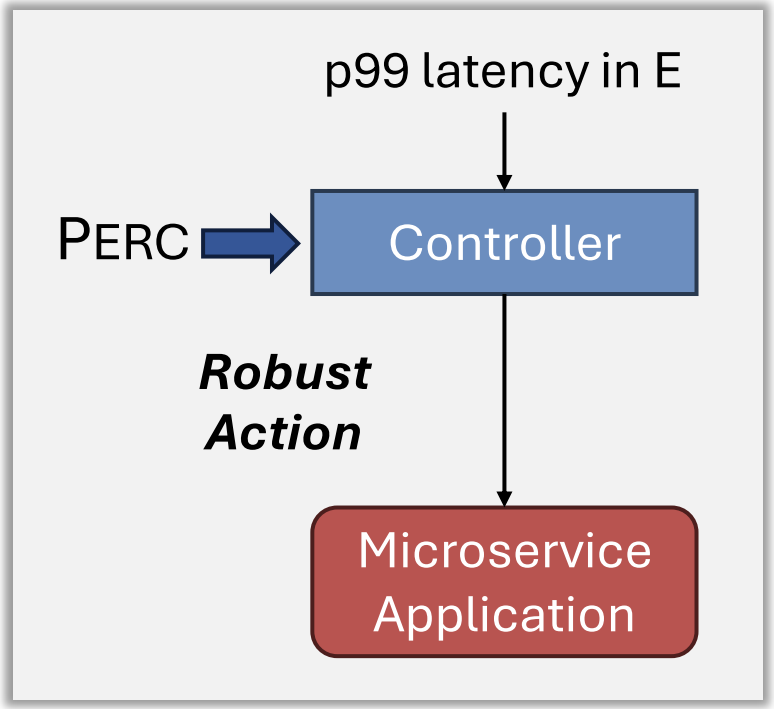
Performance Robustness Certificate (PERC)

Worst tail latency under environment perturbations.

$\text{PERC} = \max\{l_{99}(E') \mid E' \in \Delta_E\}$

Key Idea: Anticipate Performance Under Perturbations





Performance Robustness Certificate (PERC)

Worst tail latency under environment perturbations.

$\text{PERC} = \max\{l_{99}(E') \mid E' \in \Delta_E\}$

Galileo: Train Controllers for Robust Performance

1. A performance reasoning model to compute PERCs

2. Train controllers to use PERCs

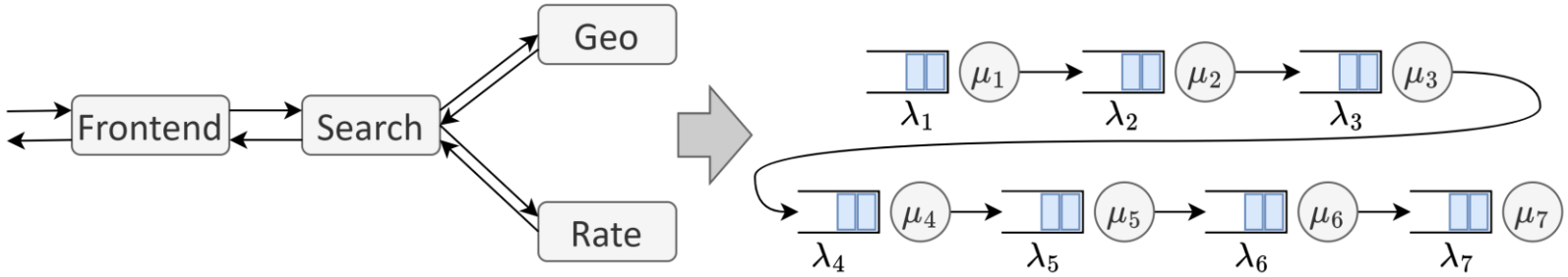
Galileo: Train Controllers for Robust Performance

1. A performance reasoning model to compute PERCs

- *What it is?* An analytical model that captures latent factors.
- *How it works?* Compute PERCs by simulating impact on performance.

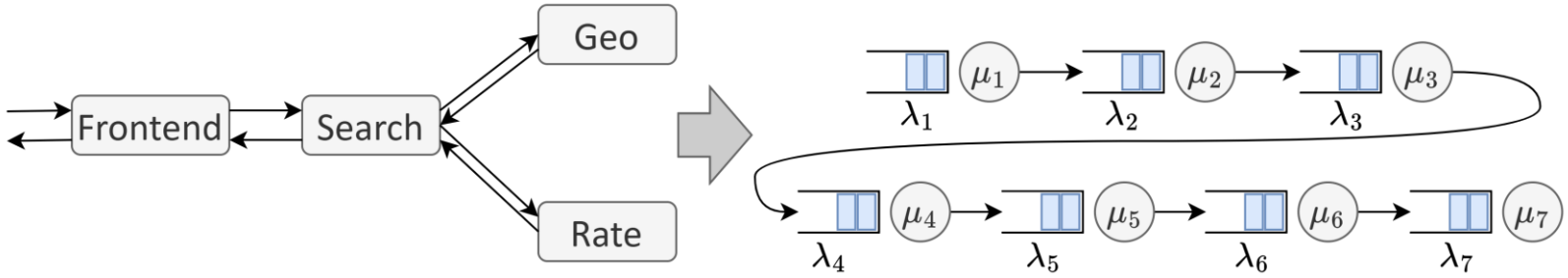
2. Train controllers to use PERCs

Performance Reasoning Rooted in Queueing Theory



Path of each request of a given type ⟶ network of $\mathrm{M/M/1\text{-}PS}$ queues

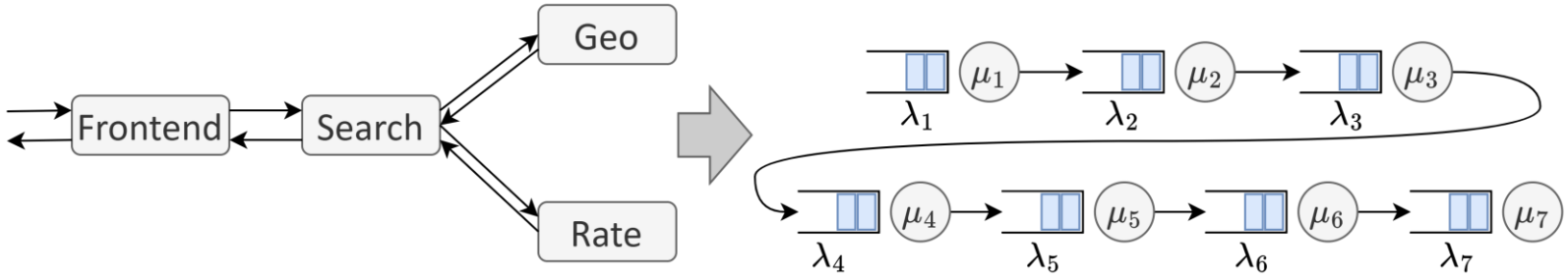
Performance Reasoning Rooted in Queueing Theory



Why queueing theory?

Matches microservice processing

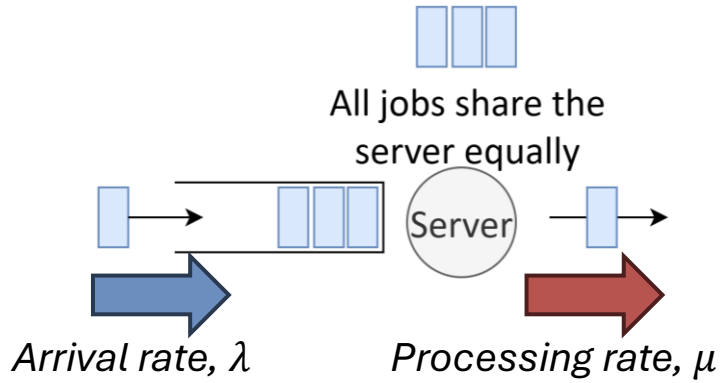
Performance Reasoning Rooted in Queueing Theory



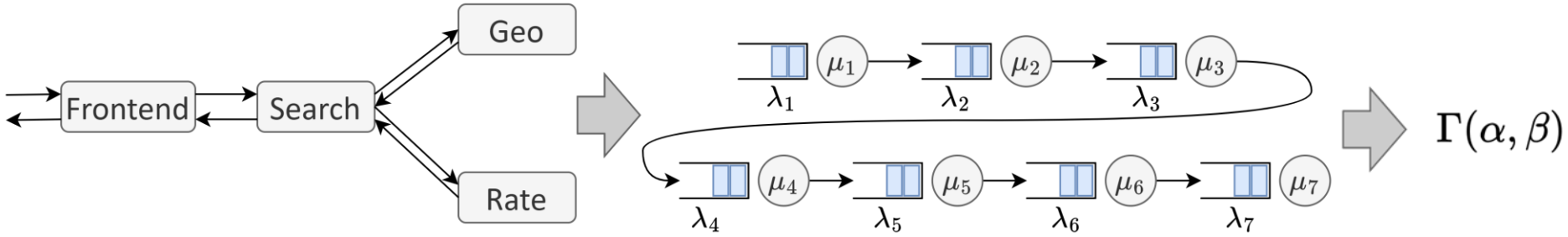
Why queueing theory?

Matches microservice processing

At each hop



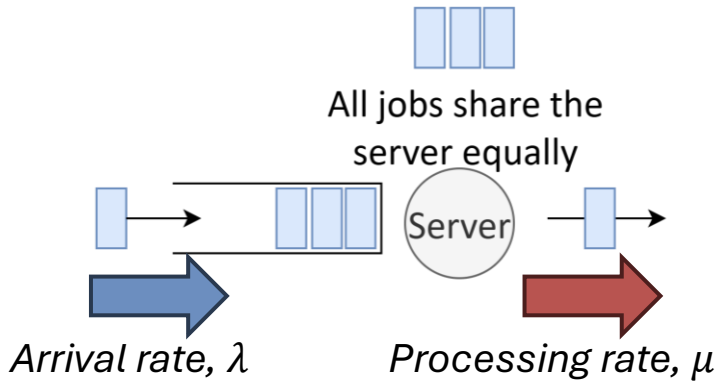
Performance Reasoning Rooted in Queueing Theory



Why queueing theory?

Matches microservice processing

At each hop

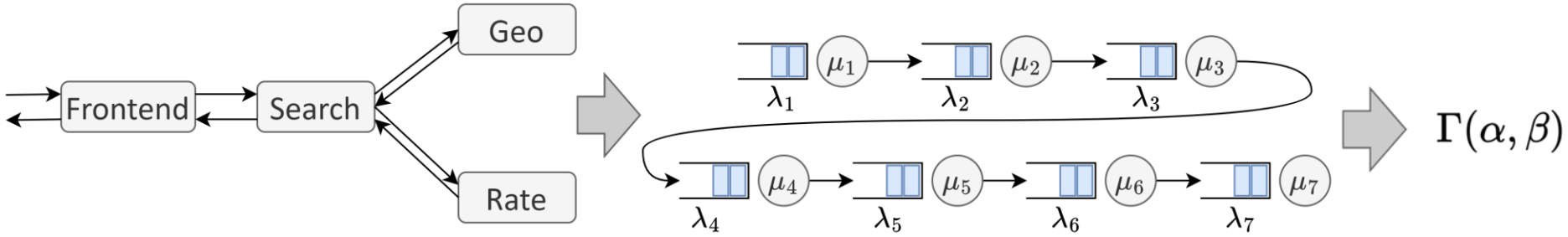


For the network of queues

End-to-end latency distribution:
$\boldsymbol{Pr(W = t) \sim \Gamma(\alpha, \beta)}$

Empirically: 4-19% average error in tail latency estimates!!

Performance Reasoning Rooted in Queueing Theory



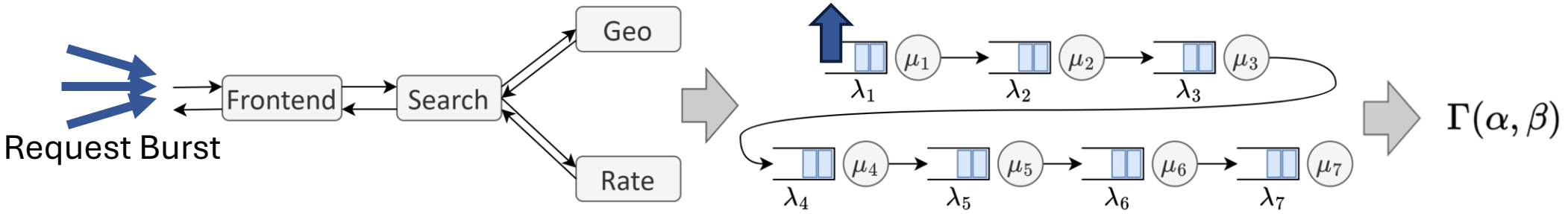
Why queueing theory?

Matches microservice processing

Perturbations captured as changes to model parameters

Any change in latent factors → Changes processing or arrival rates → Changes resulting (α, β)

Performance Reasoning Rooted in Queueing Theory



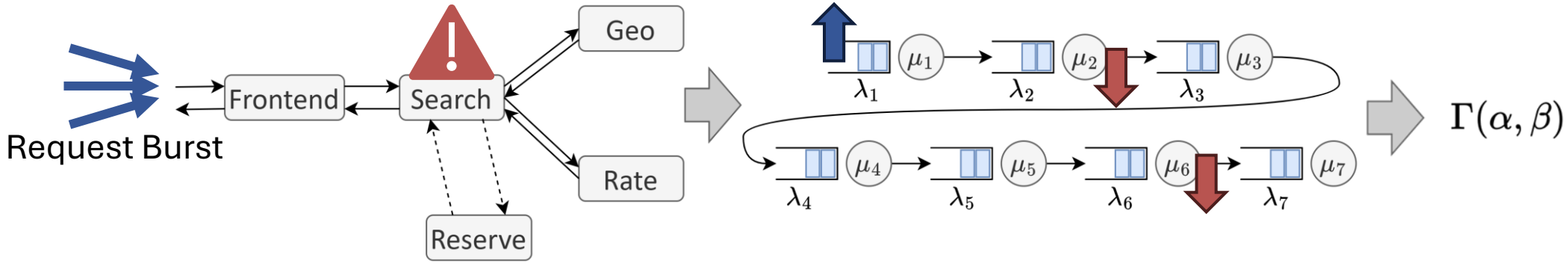
Why queueing theory?

Matches microservice processing

Perturbations captured as changes to model parameters

Any change in latent factors → Changes processing or arrival rates → Changes resulting (α, β)

Performance Reasoning Rooted in Queueing Theory



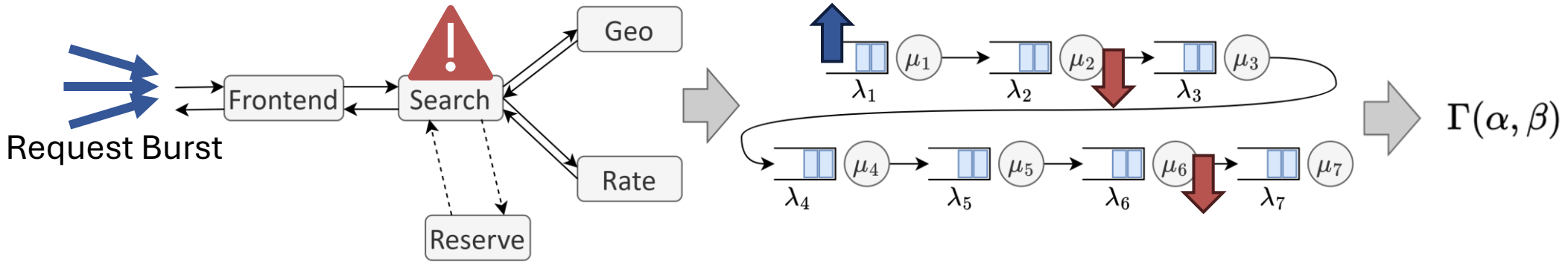
Why queueing theory?

Matches microservice processing

Perturbations captured as changes to model parameters

Any change in latent factors → Changes processing or arrival rates → Changes resulting (α, β)

Performance Reasoning Rooted in Queueing Theory



Why queueing theory?

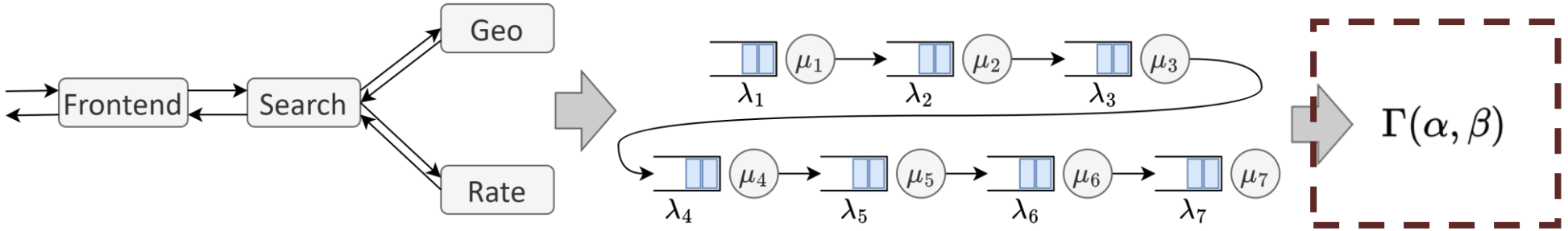
Matches microservice processing

Perturbations captured as changes to model parameters

Any change in latent factors → Changes processing or arrival rates → Changes resulting (α, β)

In our model, environment can take two types of actions:
1. Change processing rate at a queue.
2. Change arrival rate at a queue.

Performance Reasoning Rooted in Queueing Theory



Why queueing theory?

Matches microservice processing

Perturbations captured as changes to model parameters

Easy to calibrate using live data

Fit (α, β) using end-to-end latency measurements.
No additional instrumentation needed!

Encapsulates performance in a two-parameter model.
Can be fit using limited data!

Galileo: Train Controllers for Robust Performance

1. A performance reasoning model to compute PERCs

- *What it is?* An analytical model that captures latent factors.
- *How it works?* Compute PERCs by simulating impact on performance.

2. Train controllers to use PERCs

Simulating the Impact of Latent Factors

$$\text{PERC} = \max\{l_{99}(E') \mid E' \in \Delta_E\}$$

Simulating the Impact of Latent Factors

Given by $\Gamma(\alpha, \beta)$ distribution

$$\text{PERC} = \max\{l_{99}(E') \mid E' \in \Delta_E\}$$

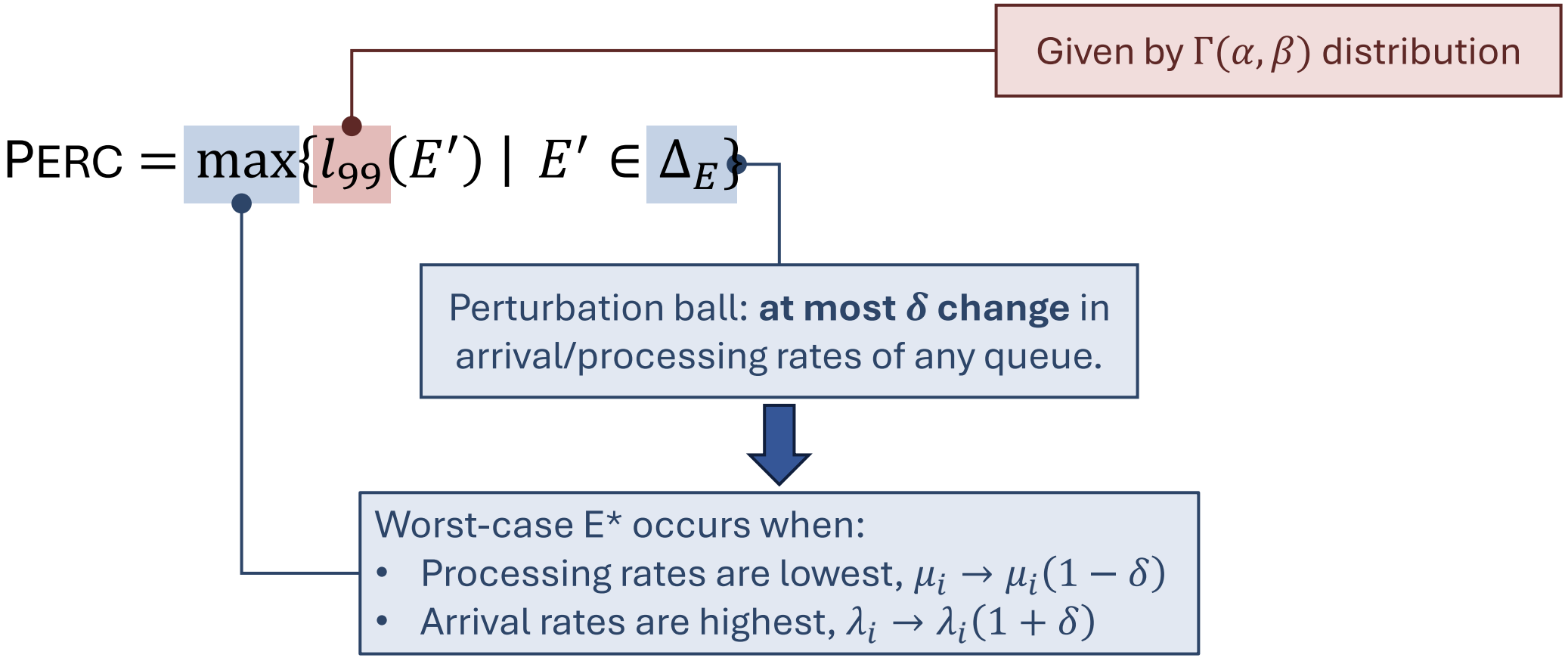
Simulating the Impact of Latent Factors

$$\text{PERC} = \max\{l_{99}(E') \mid E' \in \Delta_E\}$$

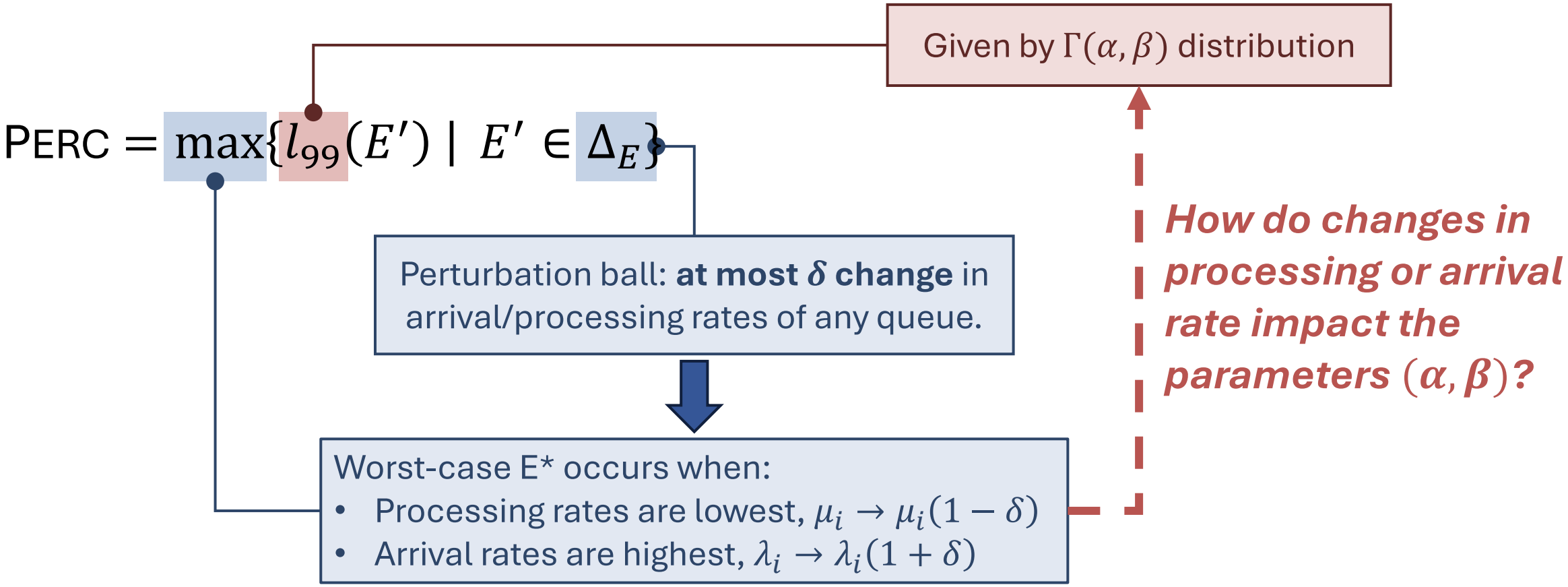
Given by $\Gamma(\alpha, \beta)$ distribution

Perturbation ball: **at most δ change** in arrival/processing rates of any queue.

Simulating the Impact of Latent Factors

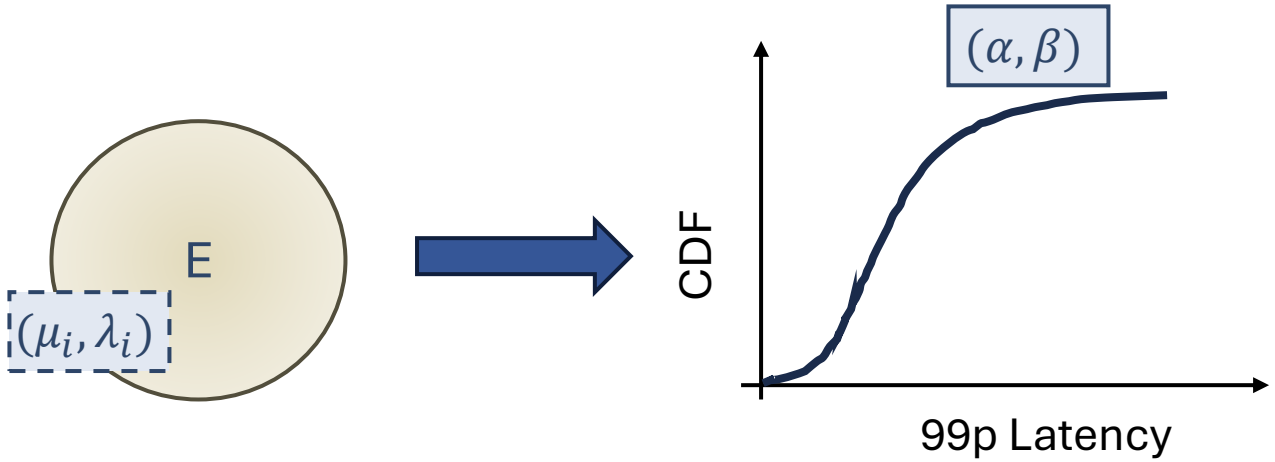


Simulating the Impact of Latent Factors



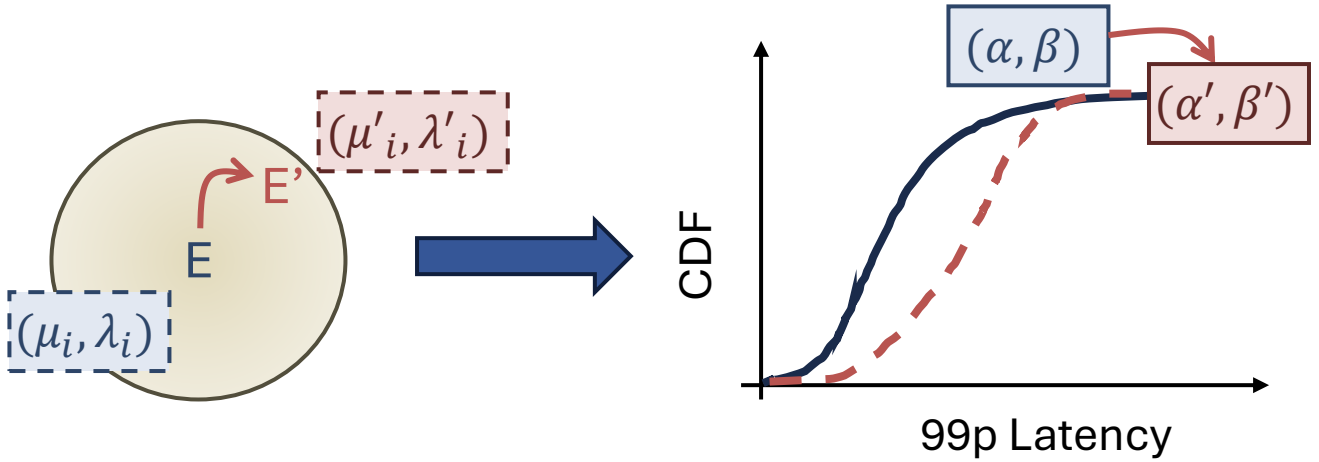
Gradient Estimation for Impact on (α, β)

Gradient Estimation for Impact on (α, β)



Gradient Estimation for Impact on (α, β)

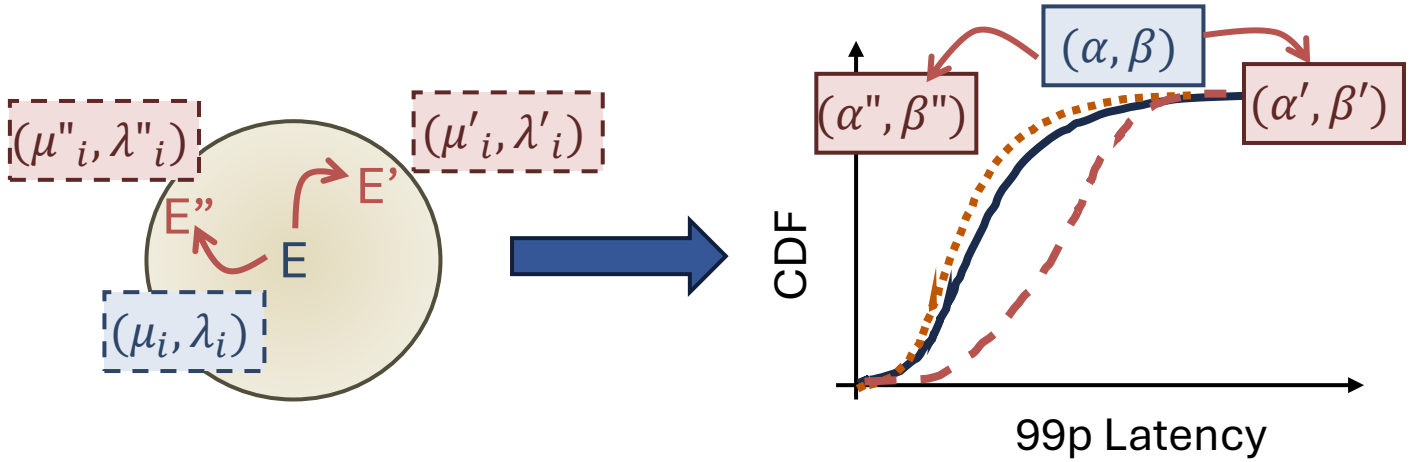
1. Tweak arrival/processing rate and measure updated (α', β').



Gradient Estimation for Impact on (α, β)

1. Tweak arrival/processing rate and measure updated (α', β').

2. Repeat over multiple tweaks to compute the gradients $(\nabla\alpha, \nabla\beta)$.

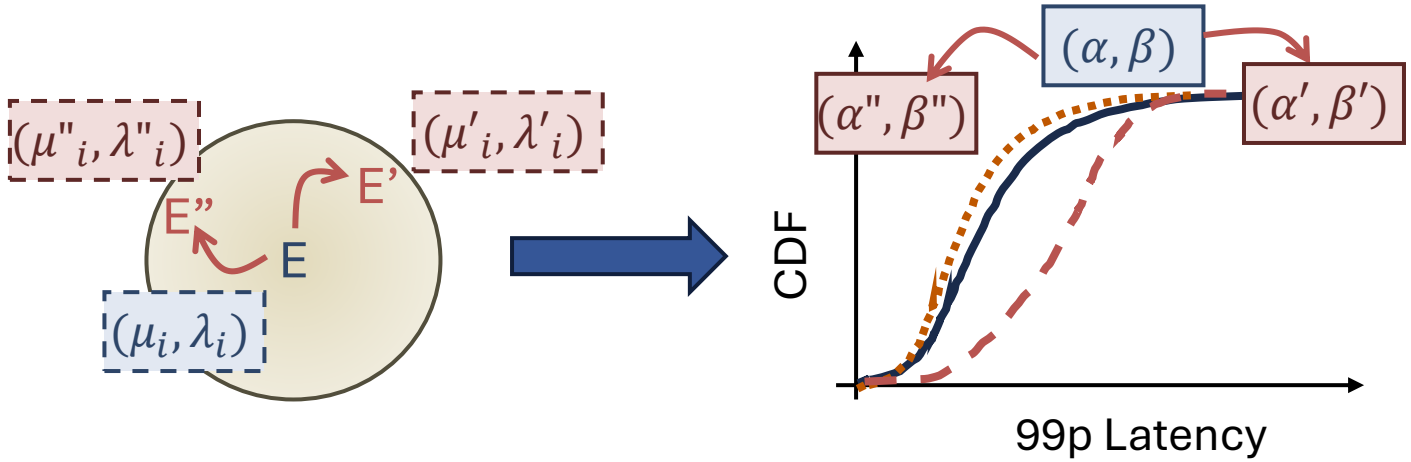


Gradient Estimation for Impact on (α, β)

1. Tweak arrival/processing rate and measure updated (α', β').

2. Repeat over multiple tweaks to compute the gradients $(\nabla\alpha, \nabla\beta)$.

3. *Changing μ or λ changes:*
$(\alpha, \beta) \rightarrow (\alpha + \eta\partial\alpha, \beta + \eta\partial\beta)$.



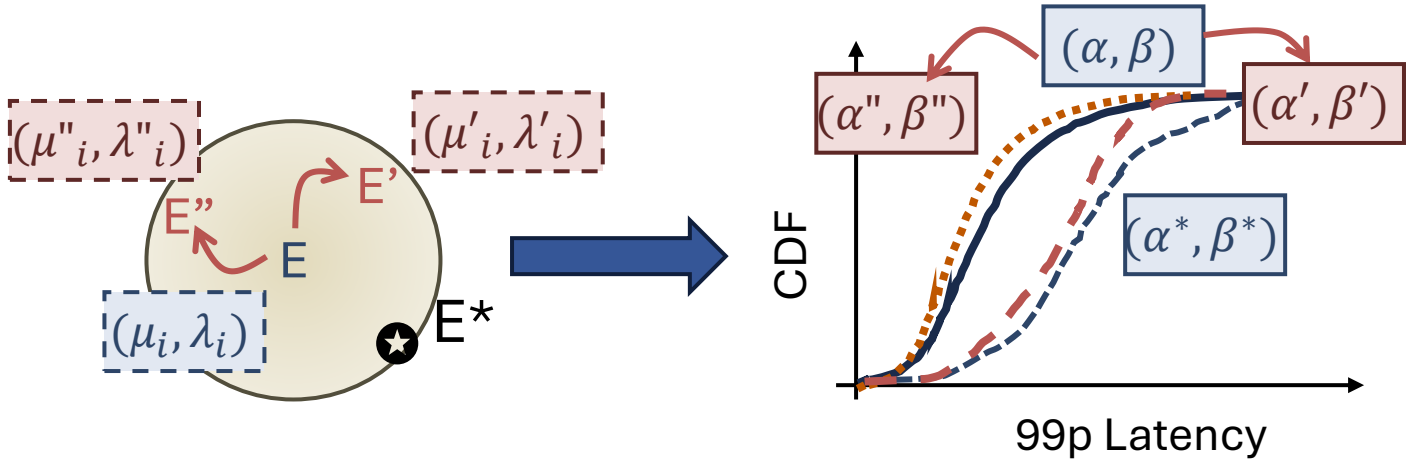
Gradient Estimation for Impact on (α, β)

1. Tweak arrival/processing rate and measure updated (α', β').

2. Repeat over multiple tweaks to compute the gradients $(\nabla\alpha, \nabla\beta)$.

3. *Changing μ or λ changes:*
$(\alpha, \beta) \rightarrow (\alpha + \eta\partial\alpha, \beta + \eta\partial\beta)$.

4. Use the semantics to get (α^*, β^*) by applying perturb actions for E*.



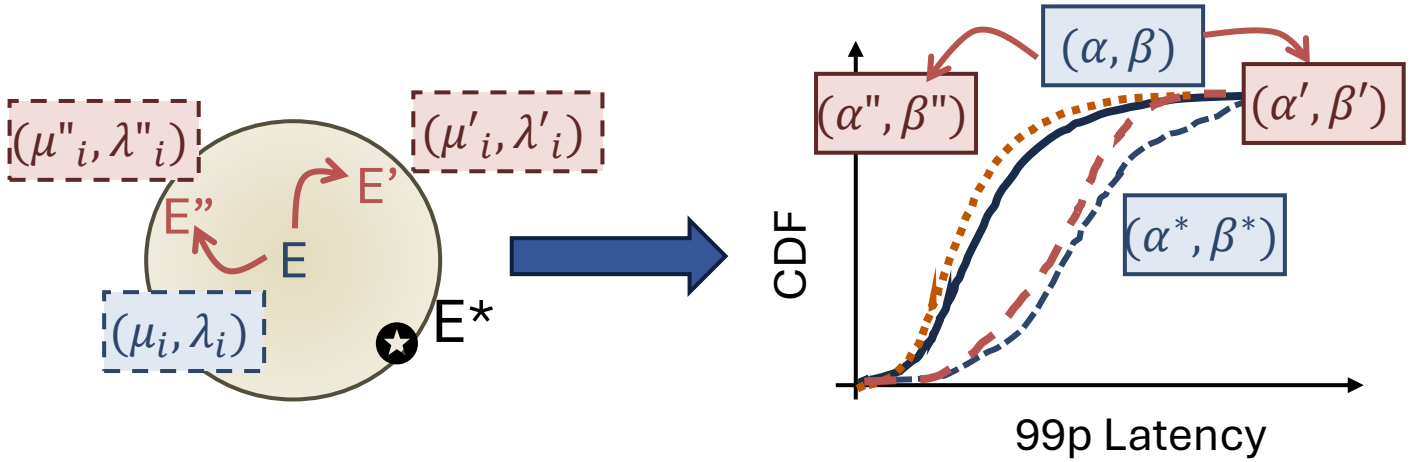
Gradient Estimation for Impact on (α, β)

1. Tweak arrival/processing rate and measure updated (α', β').

2. Repeat over multiple tweaks to compute the gradients $(\nabla\alpha, \nabla\beta)$.

3. ***Changing μ or λ changes:***
$(\alpha, \beta) \rightarrow (\alpha + \eta\partial\alpha, \beta + \eta\partial\beta)$.

4. Use the semantics to get (α^*, β^*) by applying perturb actions for E*.



Need **multiple samples** across **all dimensions!**

Use **compressive sensing*** to estimate gradients in fewer samples.

*Carleton, Vijaykumar, Saxena, et al. 2025. CONGO: Compressive Online Gradient Optimization. ICLR'25

Galileo: Train Controllers for Robust Performance

1. A performance reasoning model to compute PERCs

- *What it is?* An analytical model that captures latent factors.
- *How it works?* Compute PERCs by simulating impact on performance.

2. Train controllers to use PERCs

- Add a *robustness reward* function: “are PERCs within SLOs?”

Galileo: Train Controllers for Robust Performance

1. A performance reasoning model to compute PERCs

- *What it is?* An analytical model that captures latent factors.
- *How it works?* Compute PERCs by simulating impact on performance.

2. Train controllers to use PERCs

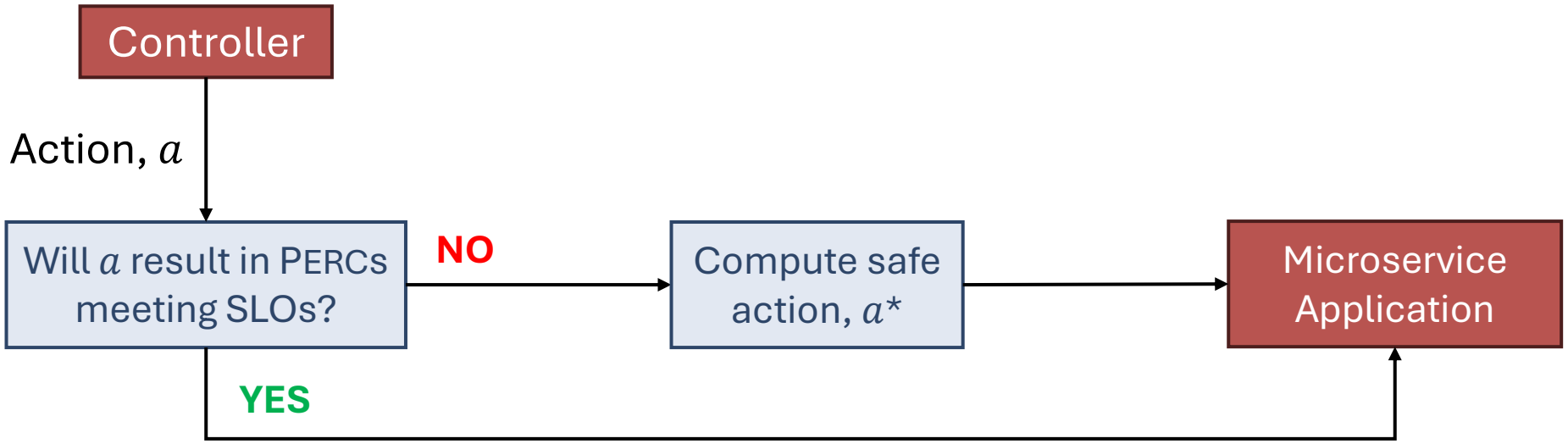
- Add a *robustness reward* function: “are PERCs within SLOs?”
- *Construct shields* to validate new controller actions

Shields to Train and Deploy Robust Controllers

Stochastic nature of learning algorithms ⟶ May not result in robust actions always!

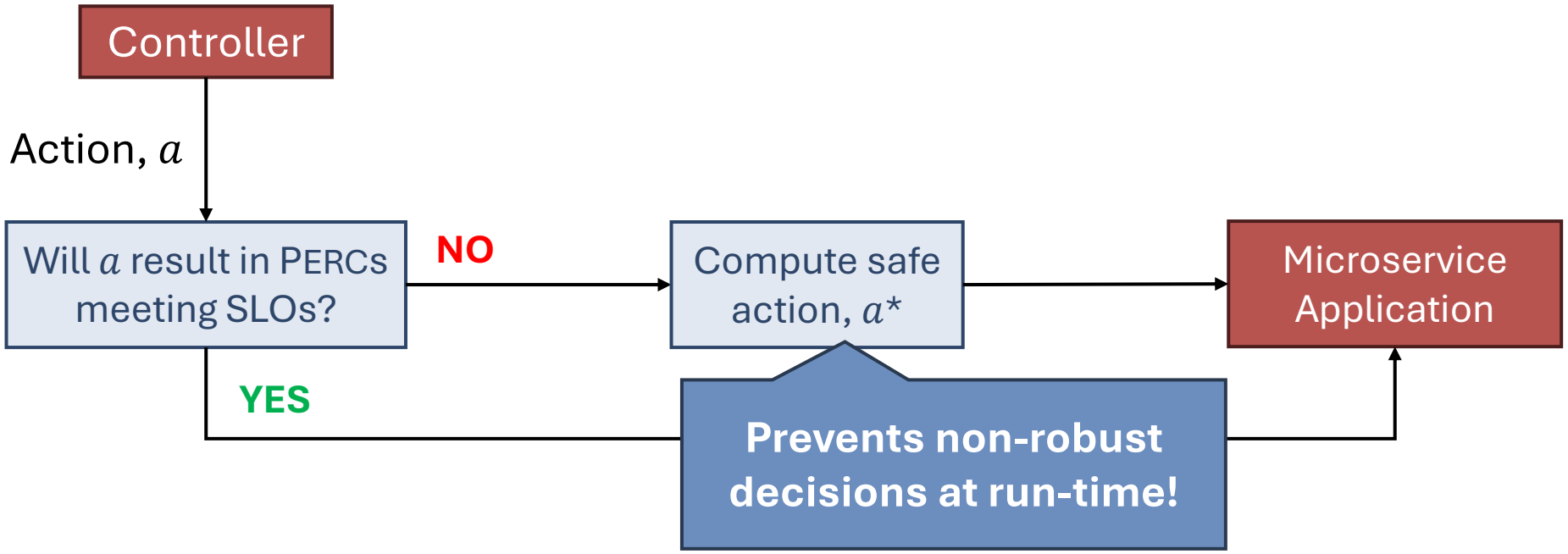
Shields to Train and Deploy Robust Controllers

⚠ Stochastic nature of learning algorithms ⟶ May not result in robust actions always!



Shields to Train and Deploy Robust Controllers

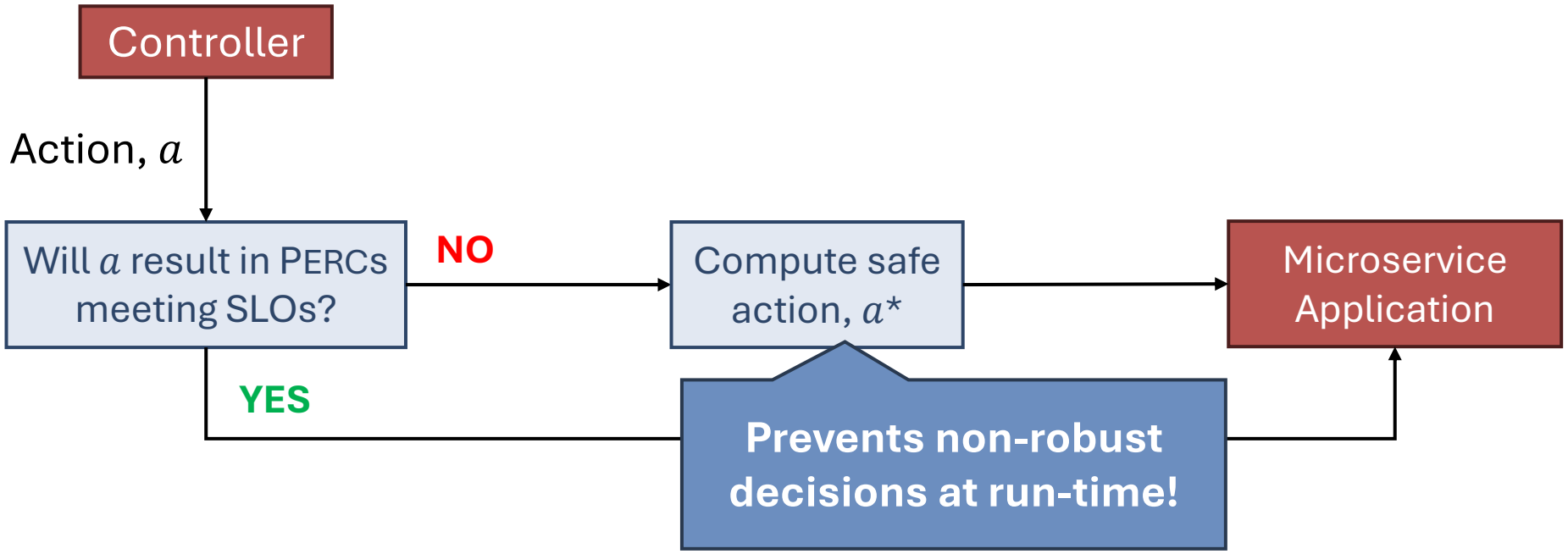
⚠ Stochastic nature of learning algorithms ⟶ May not result in robust actions always!



Shields to Train and Deploy Robust Controllers

⚠ Stochastic nature of learning algorithms ⟶ May not result in robust actions always!

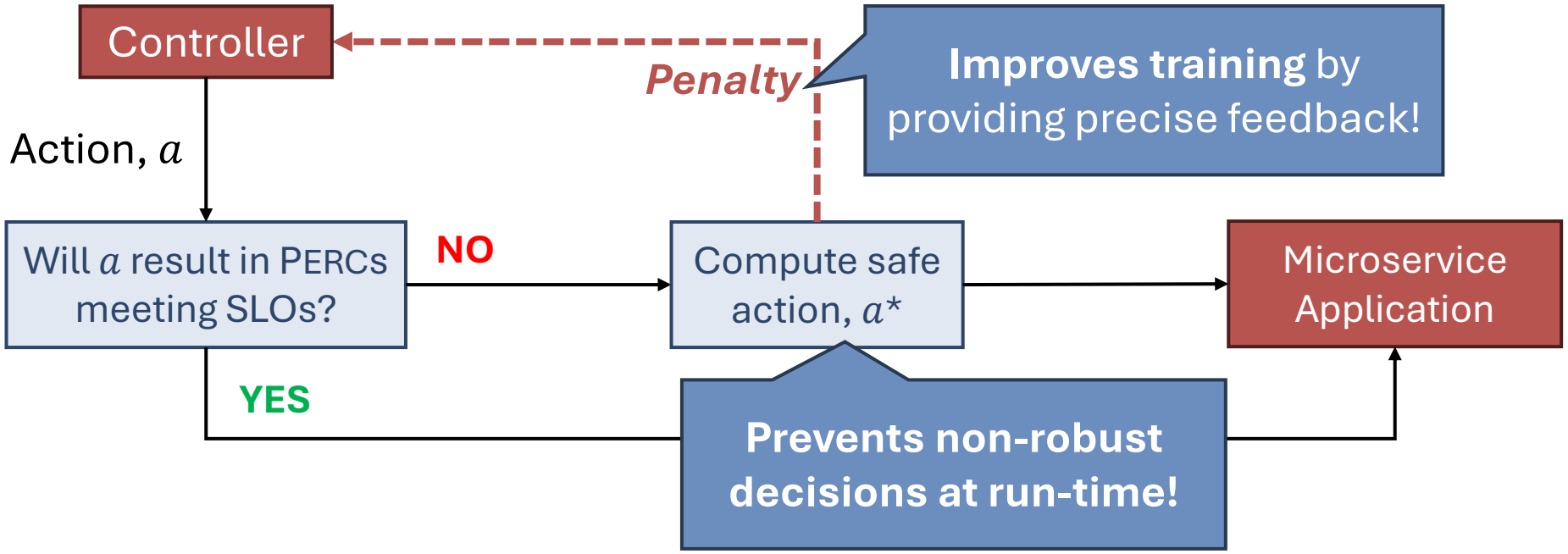
⚠ Cascading impact of controller actions ⟶ Hard to pin-point non-robust actions!



Shields to Train and Deploy Robust Controllers

⚠ Stochastic nature of learning algorithms ⟶ May not result in robust actions always!

⚠ Cascading impact of controller actions ⟶ Hard to pin-point non-robust actions!



Evaluation Highlights

- Implemented over two controllers:
  - Autothrottle*: bandit-based CPU autoscaler
  - TopFull^: RL-based admission controller
- Do PERCs provide tight bounds for latencies under perturbations?
- Are PERCs useful for run-time control?
- Overheads of computing PERCs?
- Impact of hyper-parameters?

*Wang et al. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. NSDI'24
^Park et al. 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. SIGCOMM'24

PERCs Provide Good Upper Bounds

Execution of open-source benchmarks for over 250 hours.

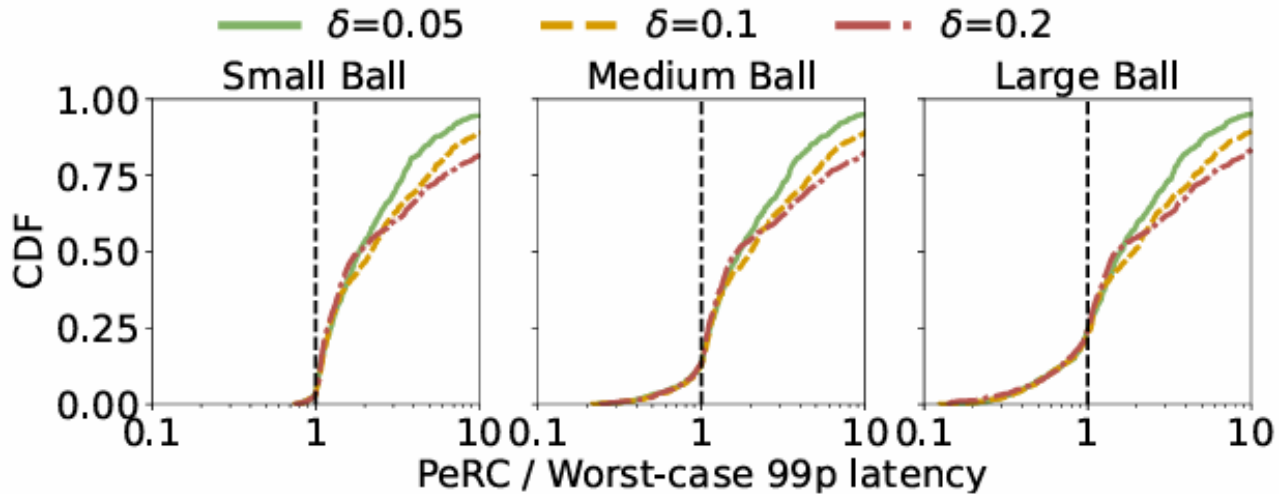
Subject to changing arrival rates, allocations and CPU contention.

PERCs Provide Good Upper Bounds

Execution of open-source benchmarks for over 250 hours.

Subject to changing arrival rates, allocations and CPU contention.

Tightness = ratio of PERC to the worst latency observed empirically.

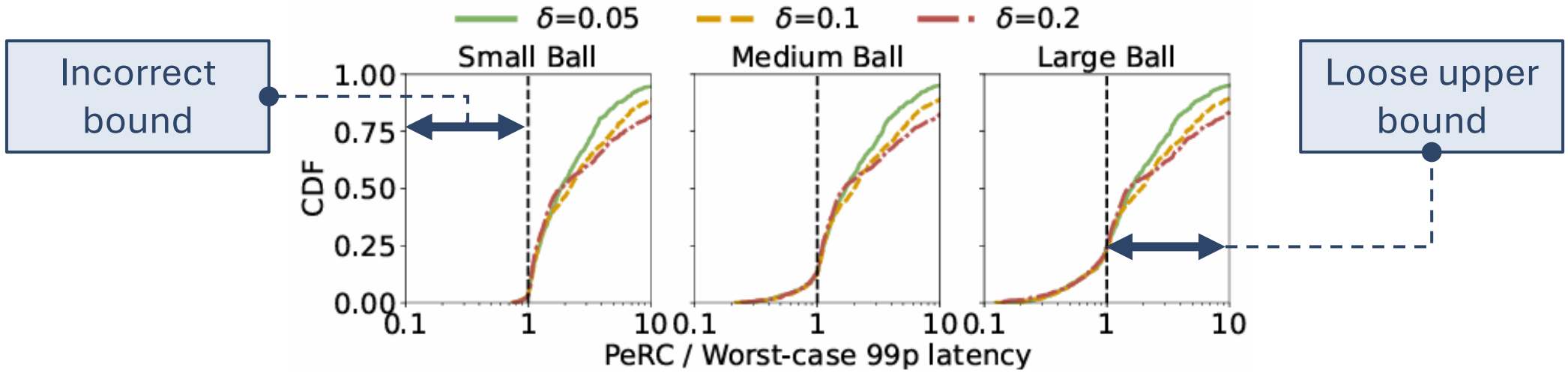


PERCs Provide Good Upper Bounds

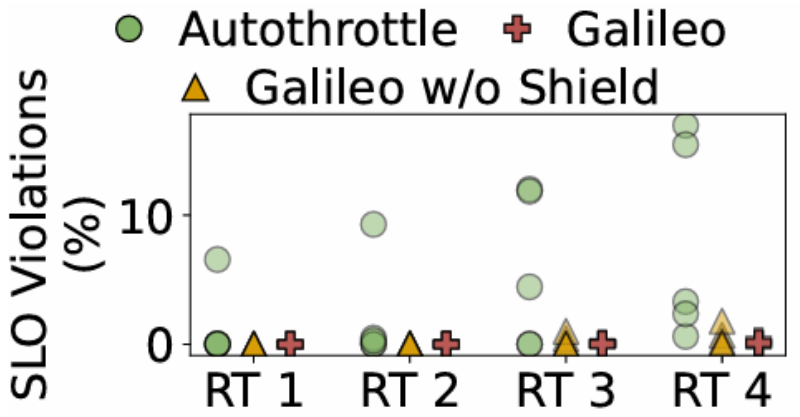
Execution of open-source benchmarks for over 250 hours.

Subject to changing arrival rates, allocations and CPU contention.

Tightness = ratio of PERC to the worst latency observed empirically.

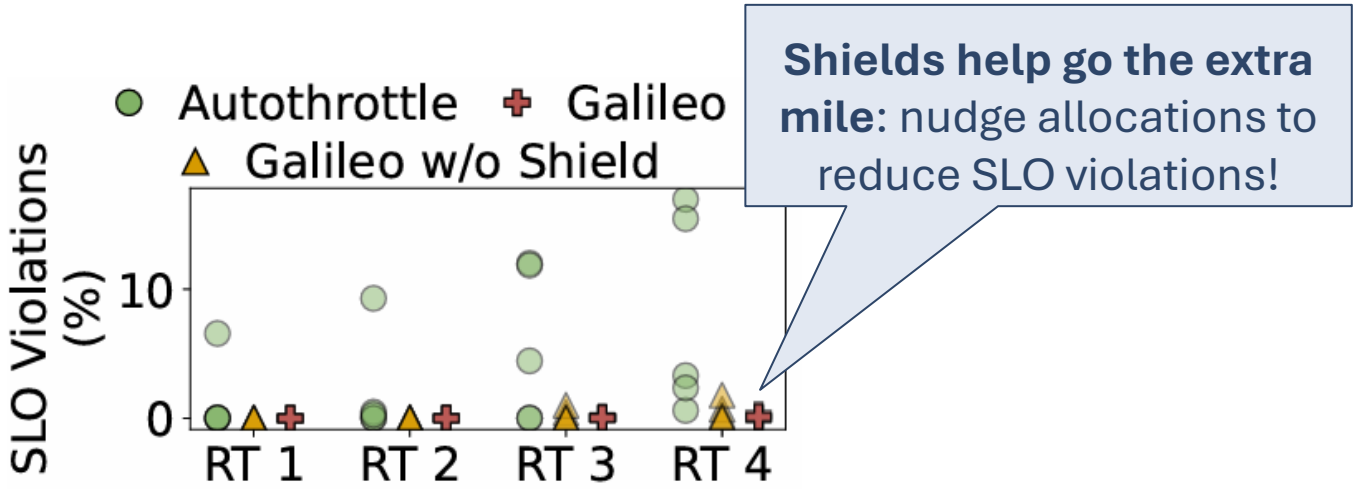


Galileo Controllers Are More Robust



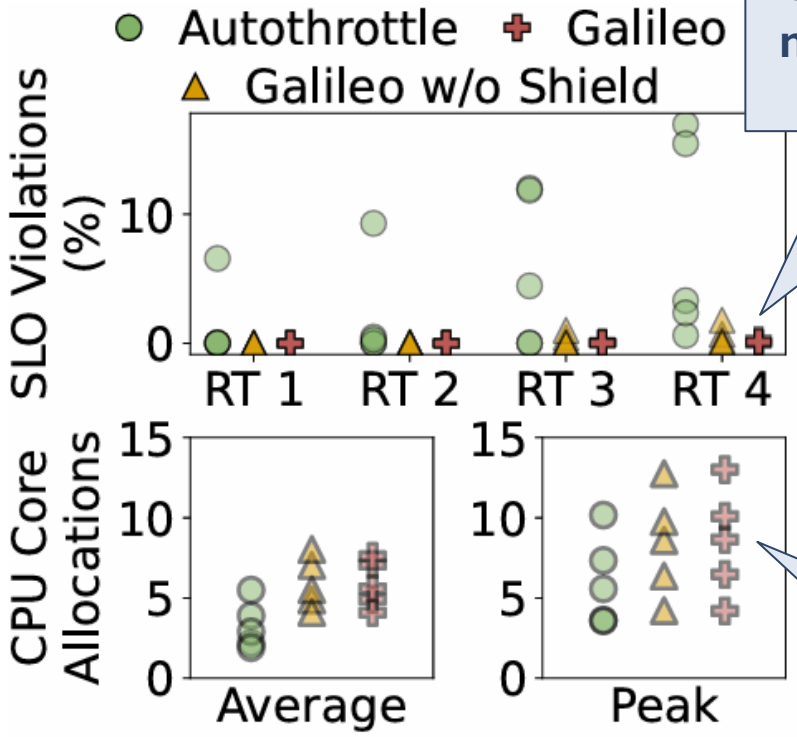
Galileo autoscaler achieves **0.05% SLO violations on average** for Hotel Reservation.

Galileo Controllers Are More Robust



Galileo autoscaler achieves **0.05% SLO violations on average** for Hotel Reservation.

Galileo Controllers Are More Robust



Shields help go the extra mile: nudge allocations to reduce SLO violations!

The cost of robustness: allocate more CPU cores

Galileo autoscaler achieves **0.05% SLO violations on average** for Hotel Reservation.

Summary and Conclusion

- **PERCs:** Bounds on tail latency in a range of perturbations.
- **Galileo:** Compute PERCs using a queueing-based model, combined with gradient-estimation techniques.
- Galileo controllers show significant improvements in SLO violation rates.

Performance robustness is **important**, but **generally hard** and **understudied**.

Galileo is a first attempt at making performance reasoning practical.

PERCs are a general abstraction applicable beyond microservices!

SCAN ME

Open-source project link

LDOS

Contact: Divyanshu Saxena (dsaxena@cs.utexas.edu)