How to Reliably Measure Software Performance

Augusto de Oliveira, Kemal Akkoyun

FOSDEM 2026

Performance matters.

Performance matters.

Low latency.

Performance matters.

Low latency. High throughput.

Performance matters.

Low latency. High throughput. **Better user experience.**

Performance has real business impact

- **Google**: 500ms delay → 20% traffic drop
- **Yahoo**: 400ms faster → 5-9% more traffic
- **Cloud costs**: $675B+ market by 2024 (Gartner)

[1]

> "Not all fast software is world-class,
> but all world-class software is fast."
>
> — Tobi Lutke, CEO of Shopify

Users feel the difference

Response Time	User Perception
100-200ms	Minimally noticeable
300-500ms	Quick but slightly slow
1-3s	Amount of work noticeable
5-10s+	User switches away

Write benchmarks.

Run them continuously.

Quick poll

Who here has written a benchmark? 🙋

Who here has written a benchmark? 🙋

Who has been surprised by the results? 🤔

But first… why is software slow?

Optimizers can't save us

- **CPUs** don't recognize bad algorithms
  - Won't swap bubble sort for quicksort
- **Compilers** rely on heuristics
  - Can't restructure your data layout
- **Big O** hides real-world costs
  - Cache misses, branch mispredictions invisible

[1]

Matrix multiplication optimization study:

60,000x speedup

through systematic tuning

This is why we need to measure.

[2]

How to Design Benchmarks

`representative` and

`repeatable`

The Art of Writing Benchmarks

Macro vs. Micro Benchmarks

Macro vs. Micro Benchmarks

Microbenchmarks

- Test isolated functions/operations
- Nanosecond-level precision
- Prone to compiler tricks
- Risk: **not representative**

Macro vs. Micro Benchmarks

Microbenchmarks

- Test isolated functions/operations
- Nanosecond-level precision
- Prone to compiler tricks
- Risk: **not representative**

Macrobenchmarks

- Test end-to-end workflows
- Realistic workloads
- Higher variance
- Risk: **hard to isolate cause**

Choose the right tool

Use Case	Benchmark Type
Comparing algorithms	Micro
Validating optimizations	Micro
Regression detection	Both
Capacity planning	Macro
User experience	Macro

Choose the right tool

Use Case	Benchmark Type
Comparing algorithms	Micro
Validating optimizations	Micro
Regression detection	Both
Capacity planning	Macro
User experience	Macro

Best practice: *Use both in your pipeline*

Representative workloads

What does your application actually do?

Representative workloads

What does your application actually do?

- **CPU-bound**: Number crunching, compression, encryption

Representative workloads

What does your application actually do?

- **CPU-bound**: Number crunching, compression, encryption
- **I/O-bound**: Database queries, API calls, file operations

Representative workloads

What does your application actually do?

- **CPU-bound**: Number crunching, compression, encryption
- **I/O-bound**: Database queries, API calls, file operations
- **Mixed**: Most real-world applications

Representative workloads

What does your application actually do?

- **CPU-bound**: Number crunching, compression, encryption
- **I/O-bound**: Database queries, API calls, file operations
- **Mixed**: Most real-world applications

Your benchmark workload should match your production workload.

Workload archetypes

Archetype	Pattern	Characteristics
Idle	Background workers, minimal load	Low RPS, minimal CPU, few workers
Latency	Microservices, APIs	High RPS, low CPU per request
Throughput	Queue workers, batch processing	Moderate RPS, high CPU, many clients
Enterprise	Business apps with DB/API calls	Moderate RPS, mixed CPU / I/O

Workload archetypes

Archetype	Pattern	Characteristics
Idle	Background workers, minimal load	Low RPS, minimal CPU, few workers
Latency	Microservices, APIs	High RPS, low CPU per request
Throughput	Queue workers, batch processing	Moderate RPS, high CPU, many clients
Enterprise	Business apps with DB/API calls	Moderate RPS, mixed CPU / I/O

Choose the archetype that matches your application's behavior.

How to Design Benchmarks: Case Study


-First time?
imgflip.com

An non-repeatable benchmark

- Goal: Measuring dd-trace-java instrumentation overhead on a Spring app.

An non-repeatable benchmark

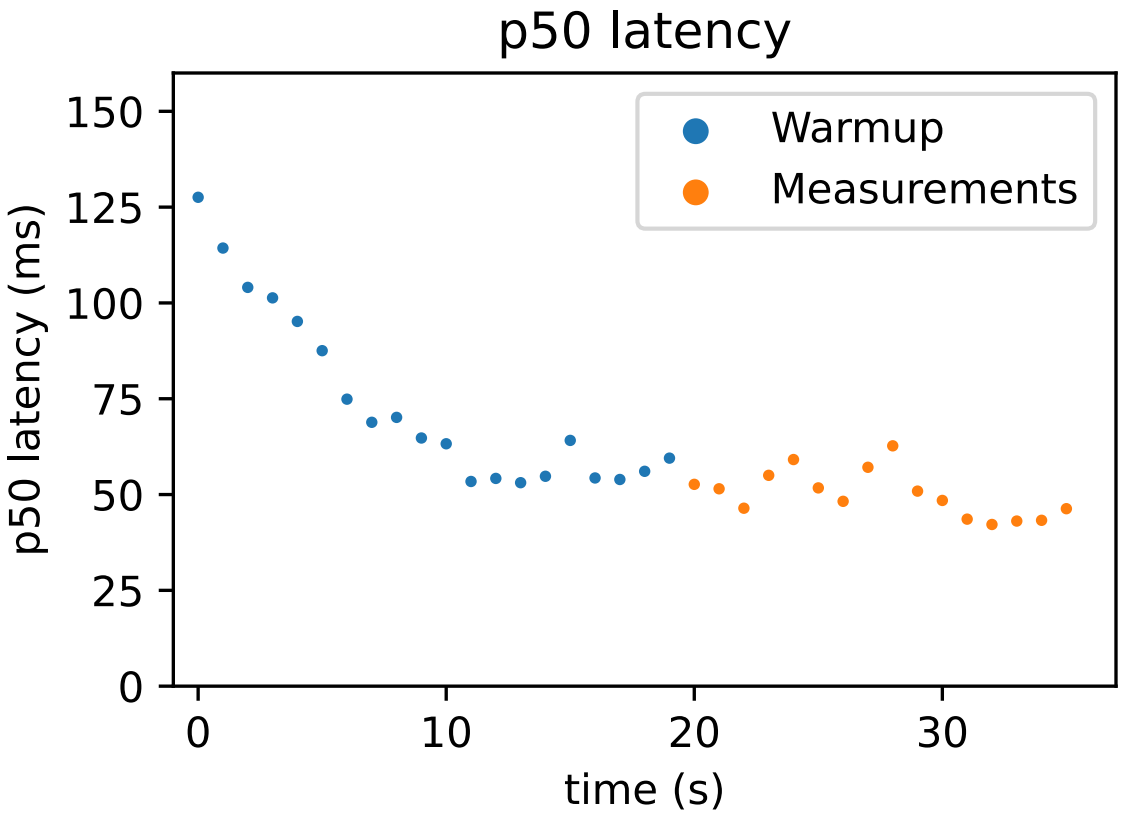
- Goal: Measuring dd-trace-java instrumentation overhead on a Spring app.
- **System under test: Spring app instrumented (or not) with dd-trace-java.**

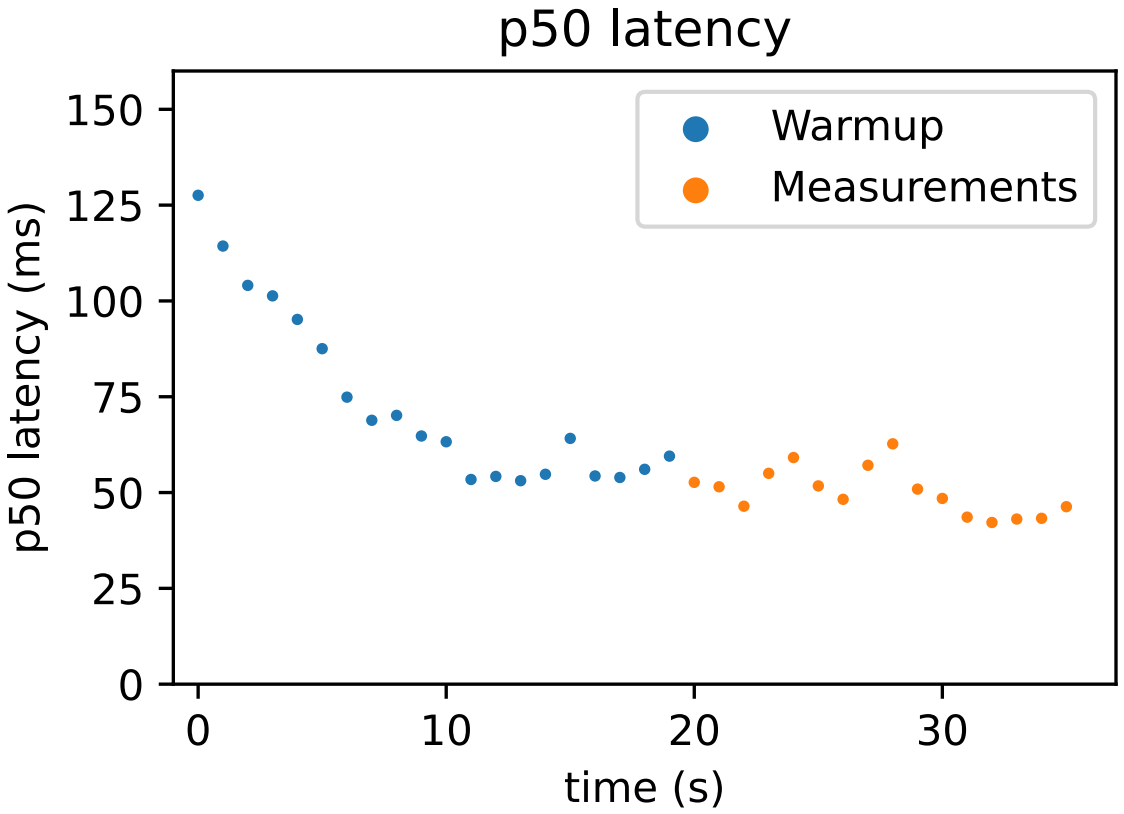
An non-repeatable benchmark

- Goal: Measuring dd-trace-java instrumentation overhead on a Spring app.
- System under test: Spring app instrumented (or not) with dd-trace-java.
- **Workload: As many requests as possible by 5 concurrent users.**

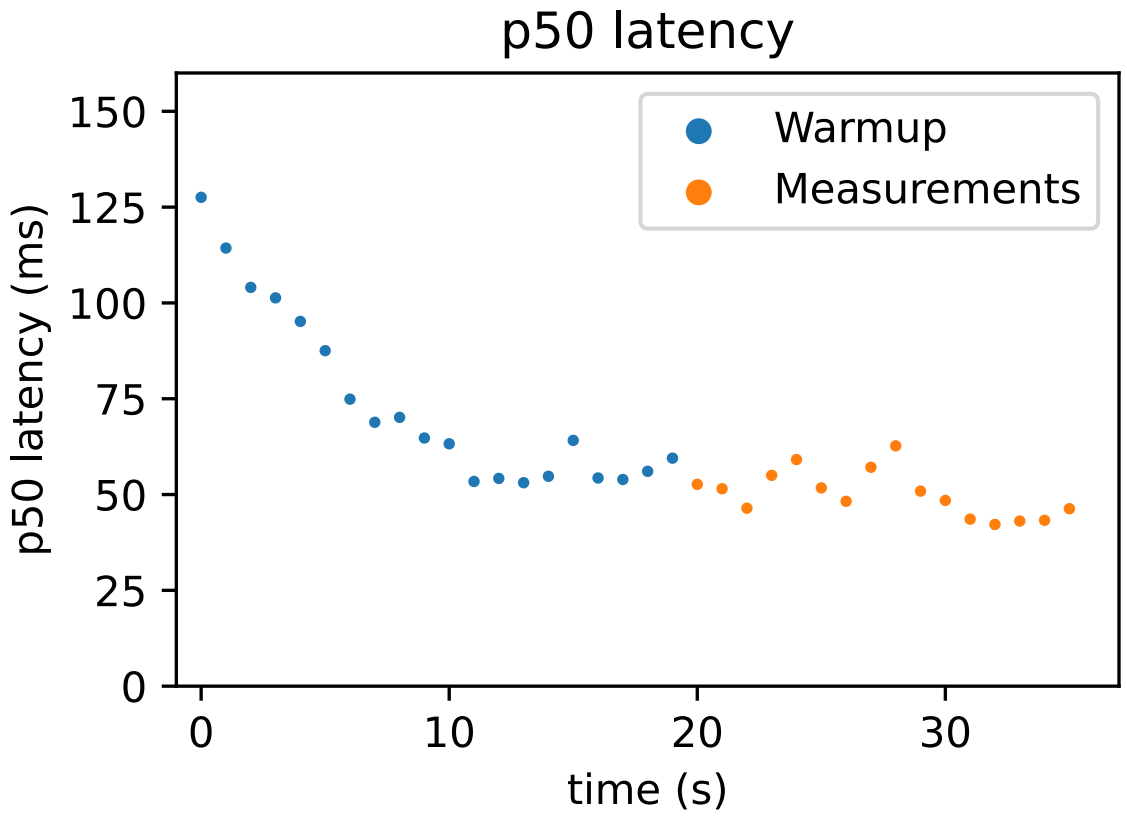
An non-repeatable benchmark

- Goal: Measuring dd-trace-java instrumentation overhead on a Spring app.
- System under test: Spring app instrumented (or not) with dd-trace-java.
- Workload: As many requests as possible by 5 concurrent users.
- **20 second warmup, 15 seconds of actual measurements.**


p50 latency
Warmup
Measurements
p50 latency (ms)
150
125
100
75
50
25
0
0
10
20
30
time (s)

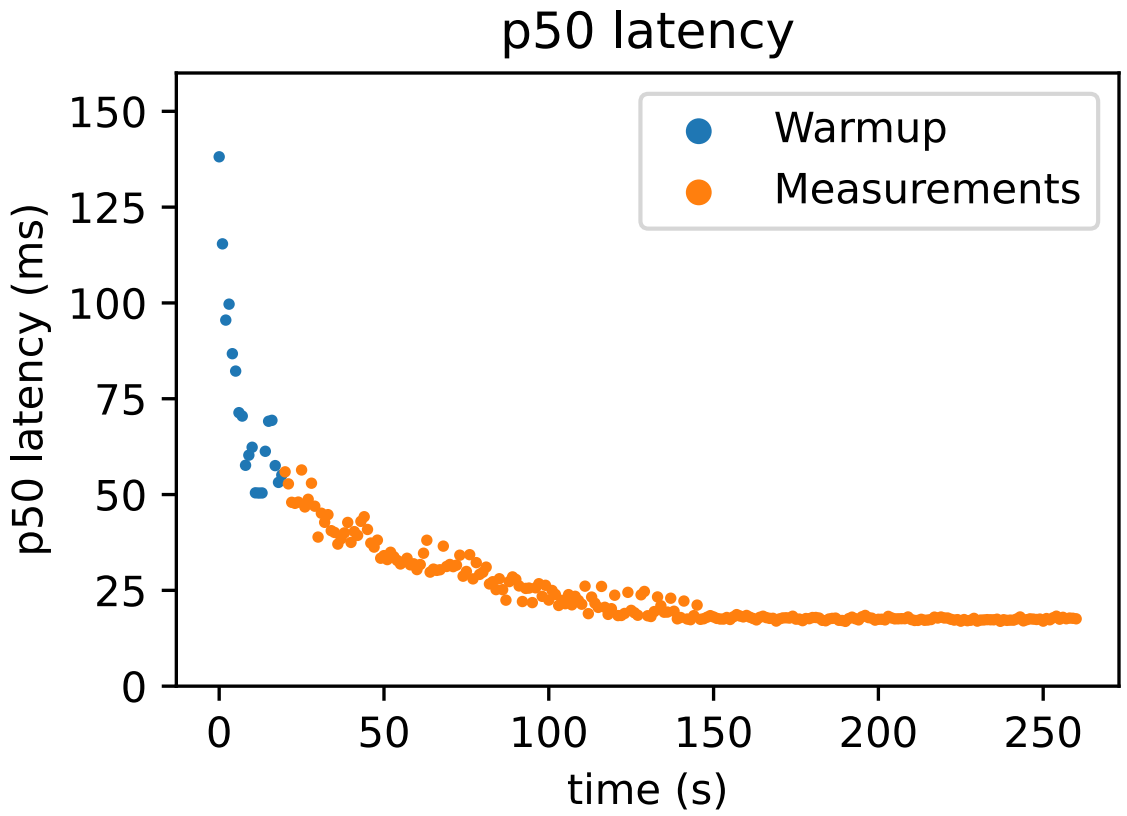


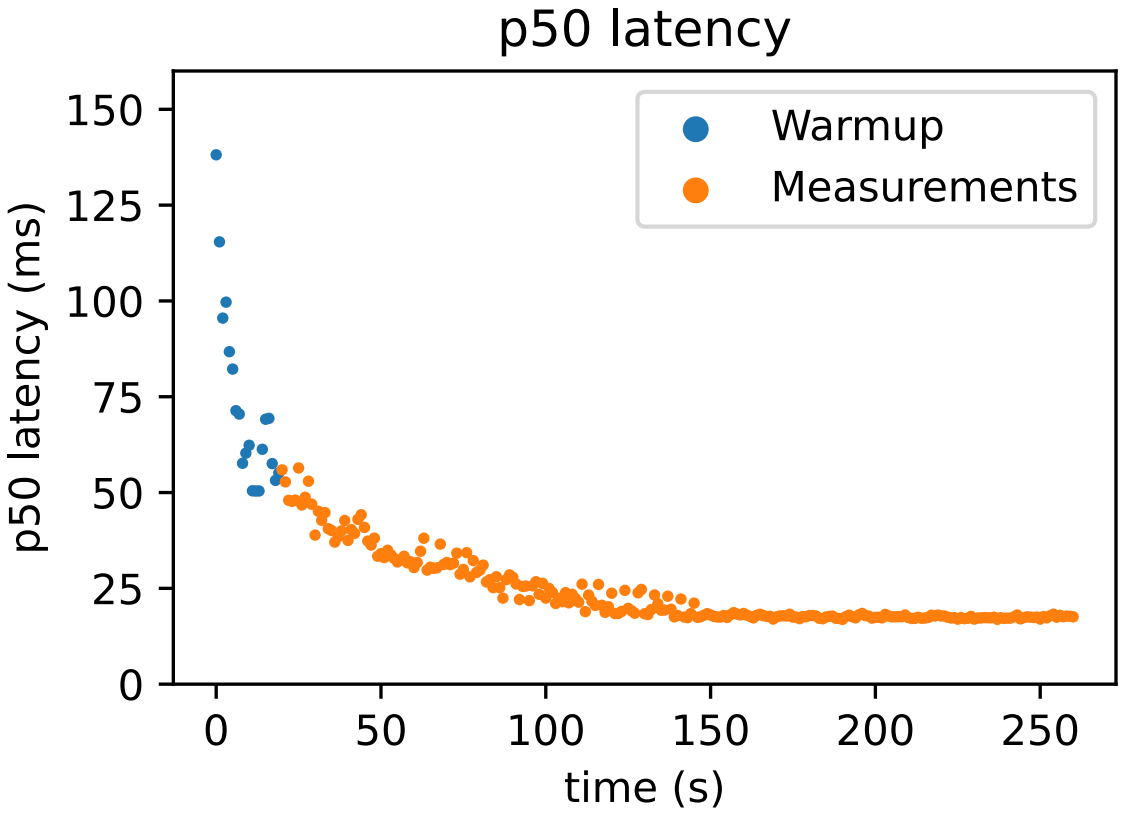
Many **false positives** and **high coeff. of variation** (= standard deviation / mean) of 11.80%.



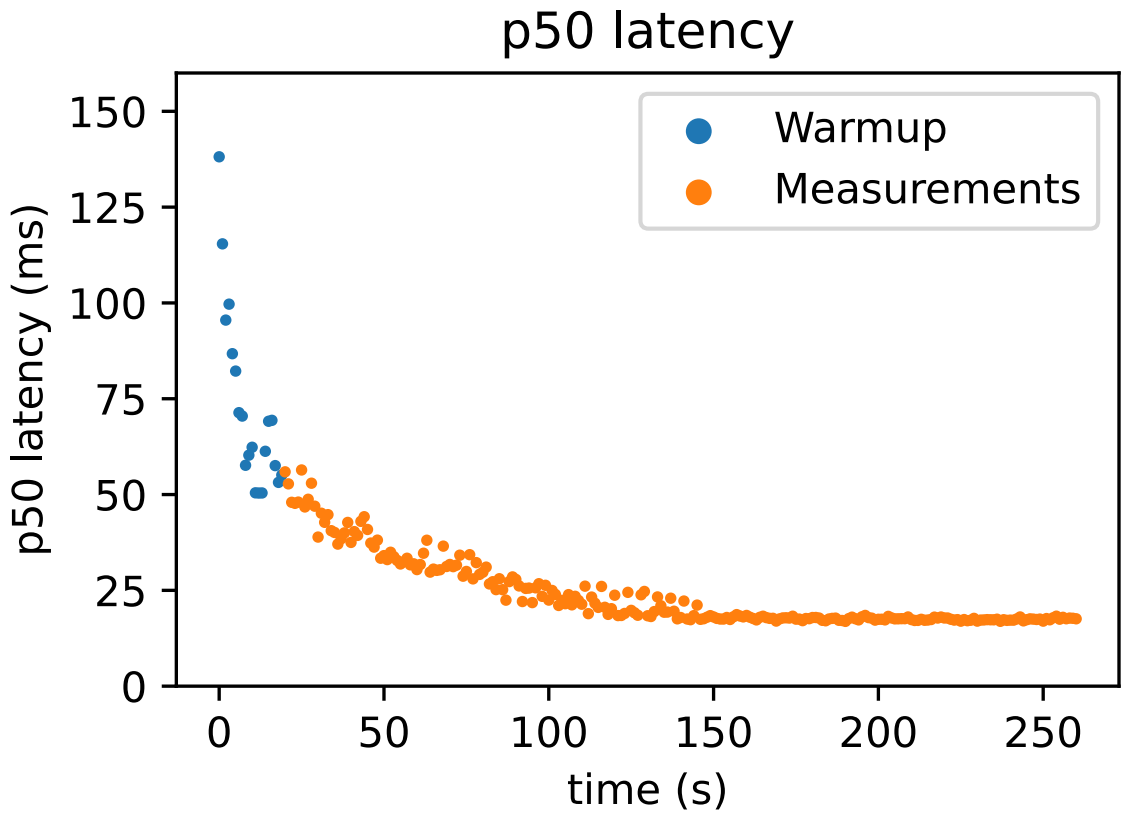
Many **false positives** and **high coeff. of variation** (= standard deviation / mean) of 11.80%.

Are we running the benchmark long enough?


p50 latency
Warmup
Measurements
p50 latency (ms)
150
125
100
75
50
25
0
0
50
100
150
200
250
time (s)

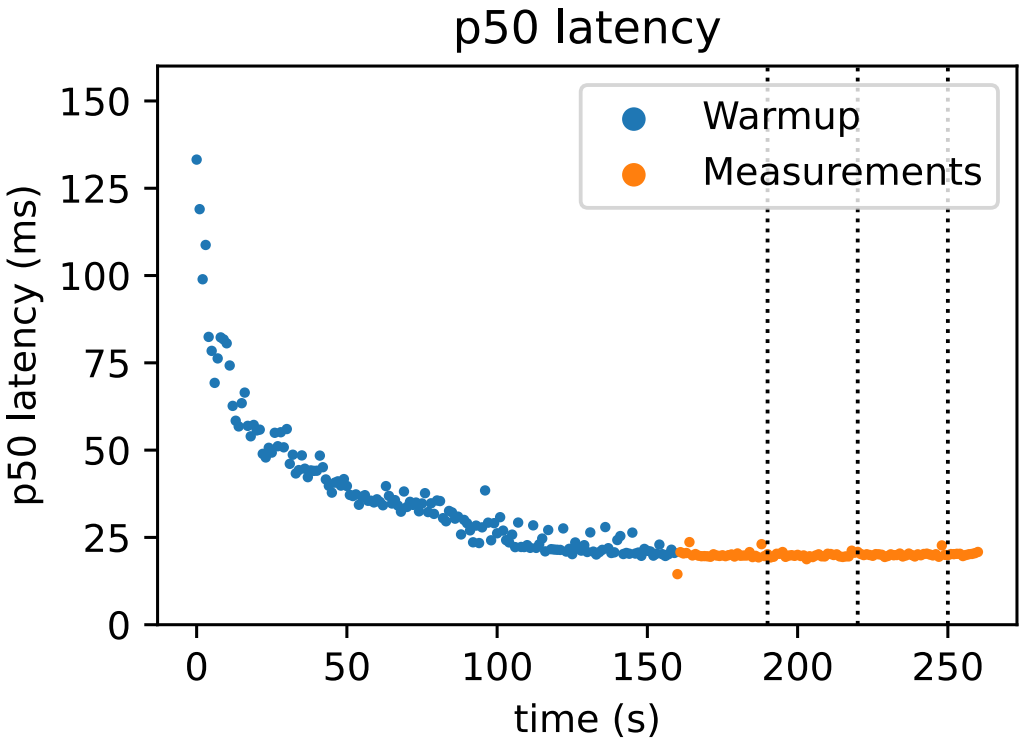


Tip #1: Run benchmarks for longer to uncover perturbations (e.g., warmup effects).

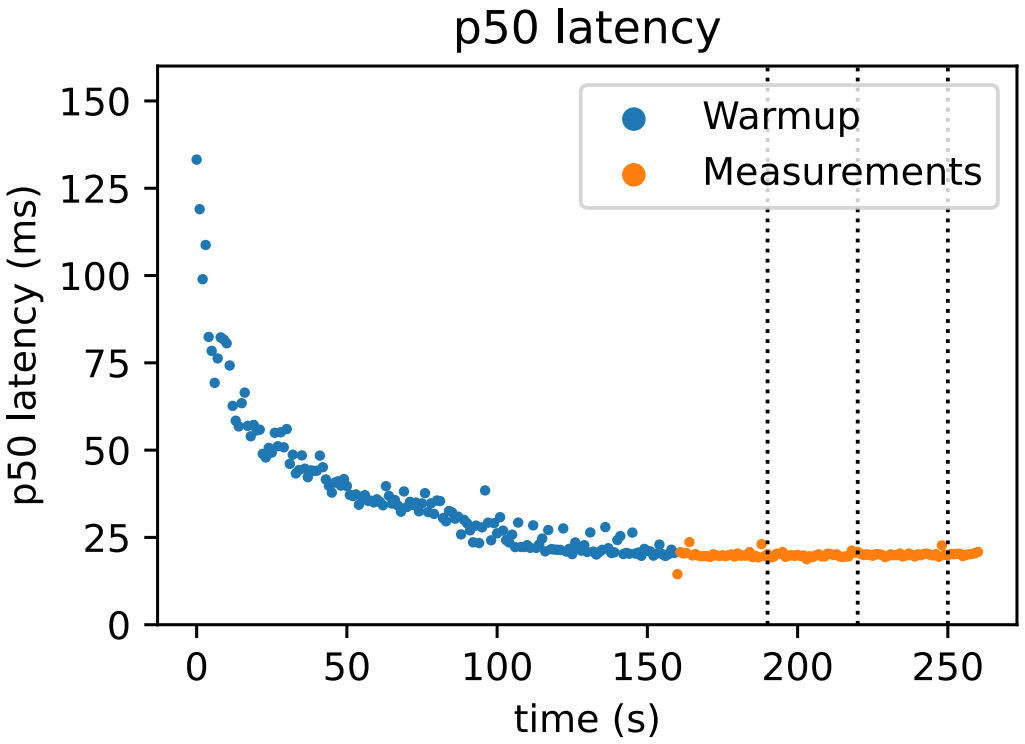


Tip #1: Run benchmarks for longer to uncover perturbations (e.g., warmup effects).

For how long should we run the benchmark?

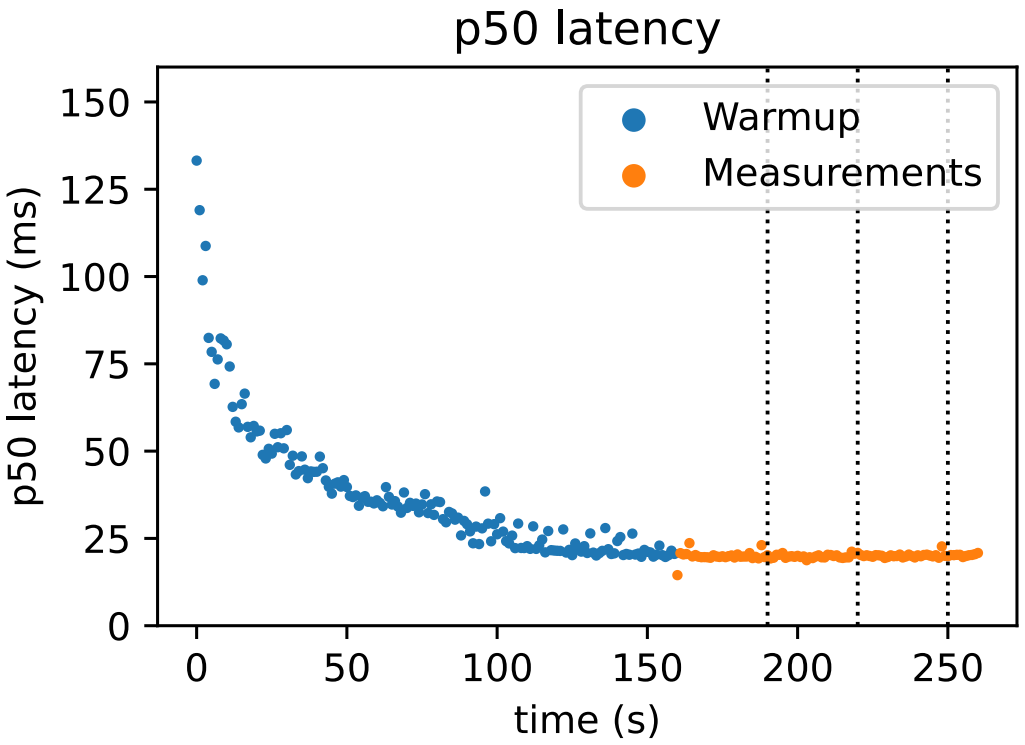


# measurements	coeff. of variation
30	6.95%
60	5.23%
90	4.59%



# measurements	coeff. of variation
30	6.95%
60	5.23%
90	4.59%

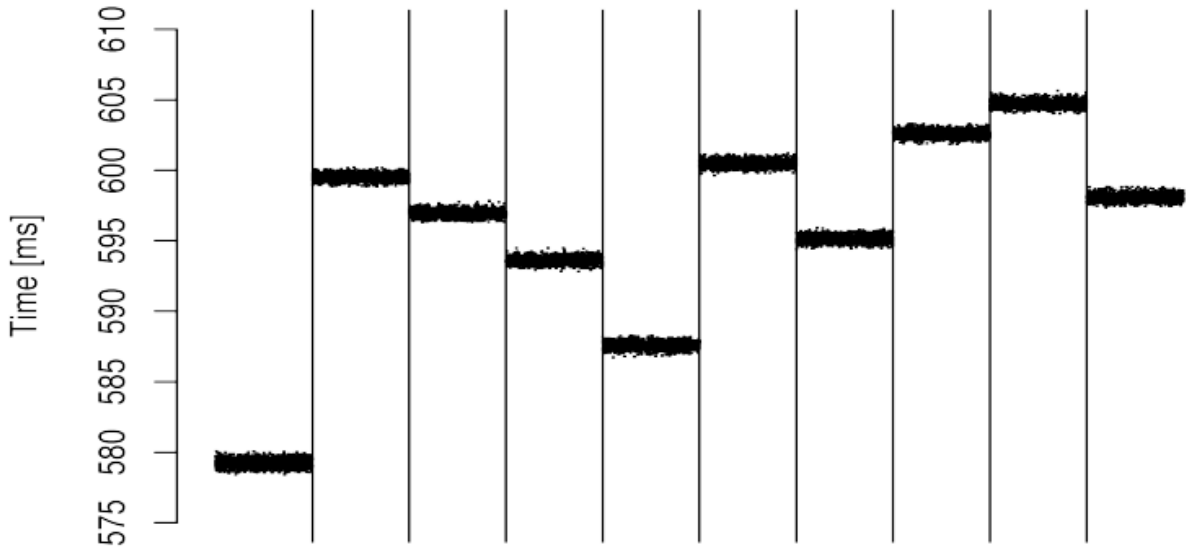
Tip #2: Collect enough samples to reduce intra-run variation (N ≥ 30).



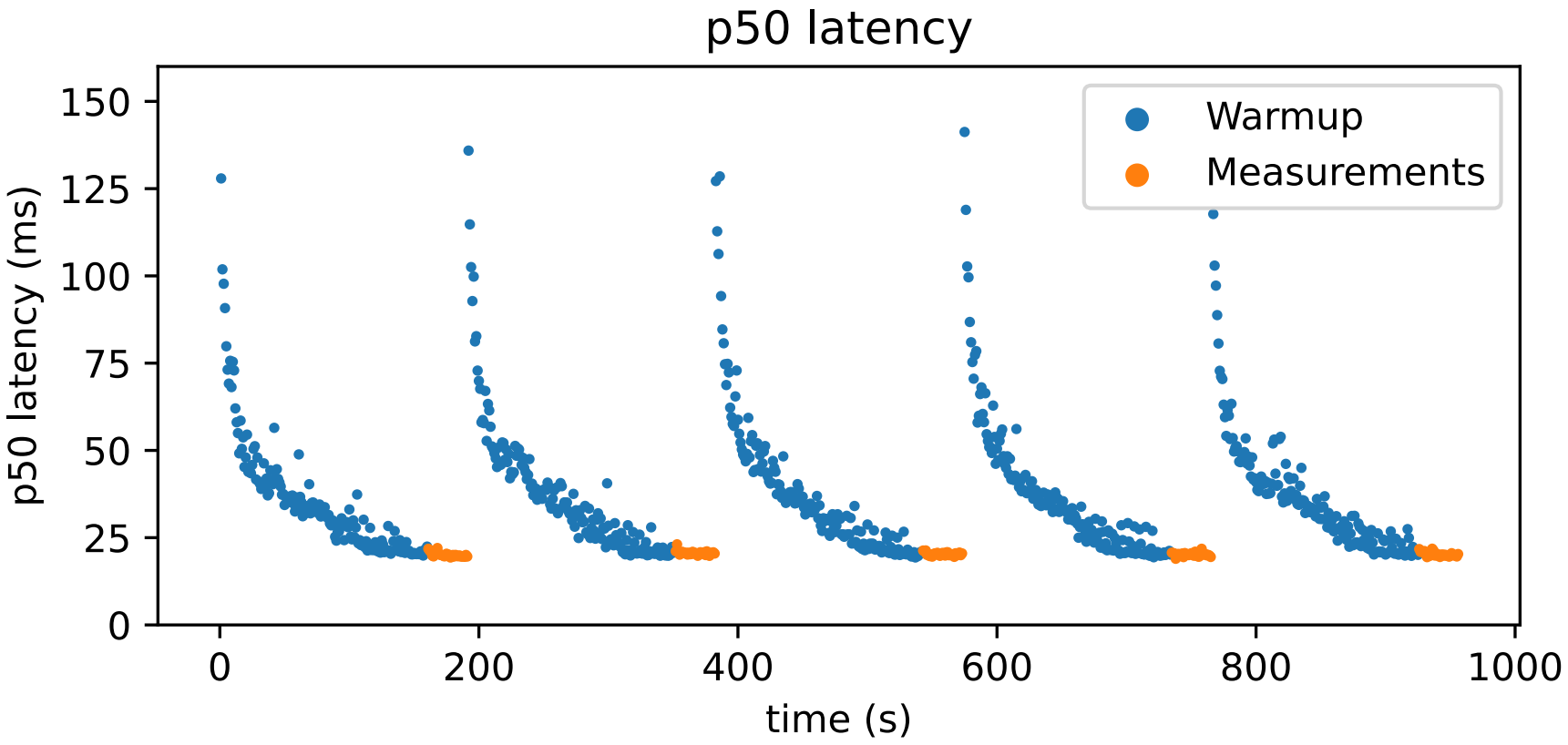
# measurements	coeff. of variation
30	6.95%
60	5.23%
90	4.59%

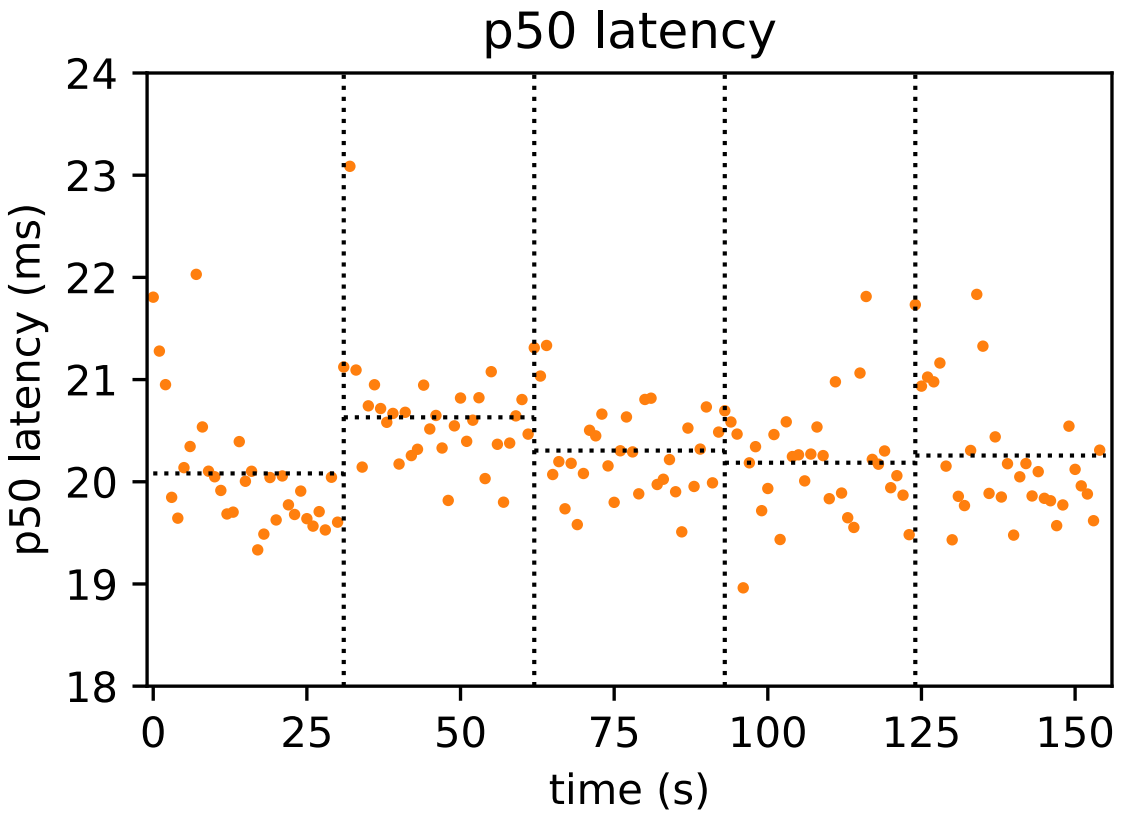
Tip #2: Collect enough samples to reduce intra-run variation (N ≥ 30).

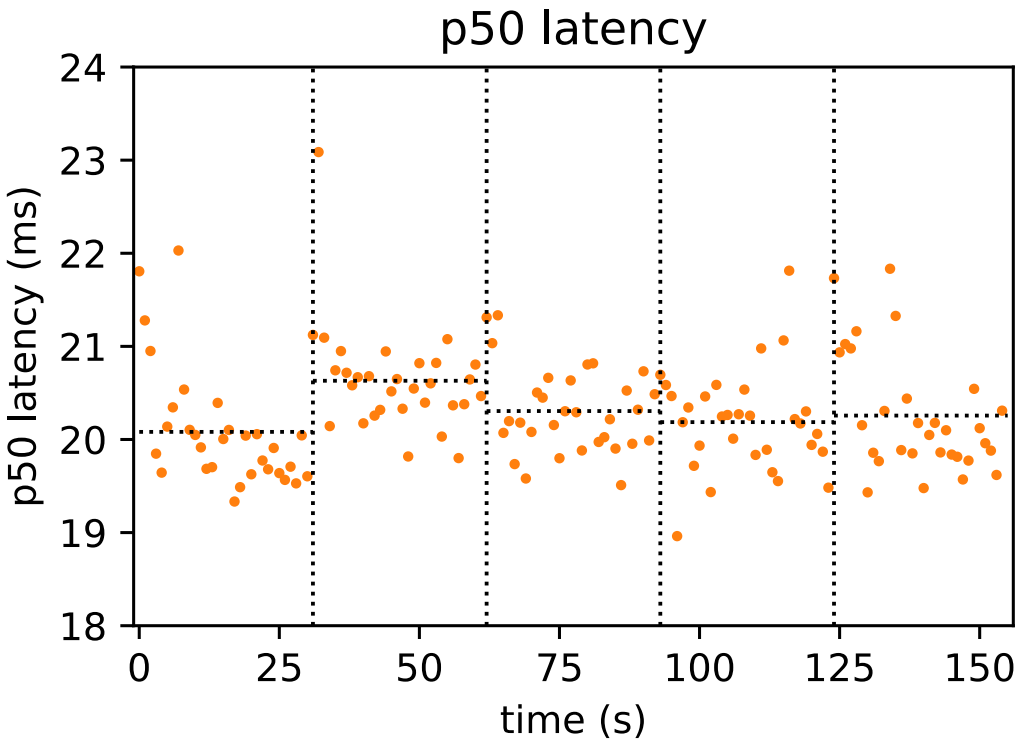
But what about inter-run variation?



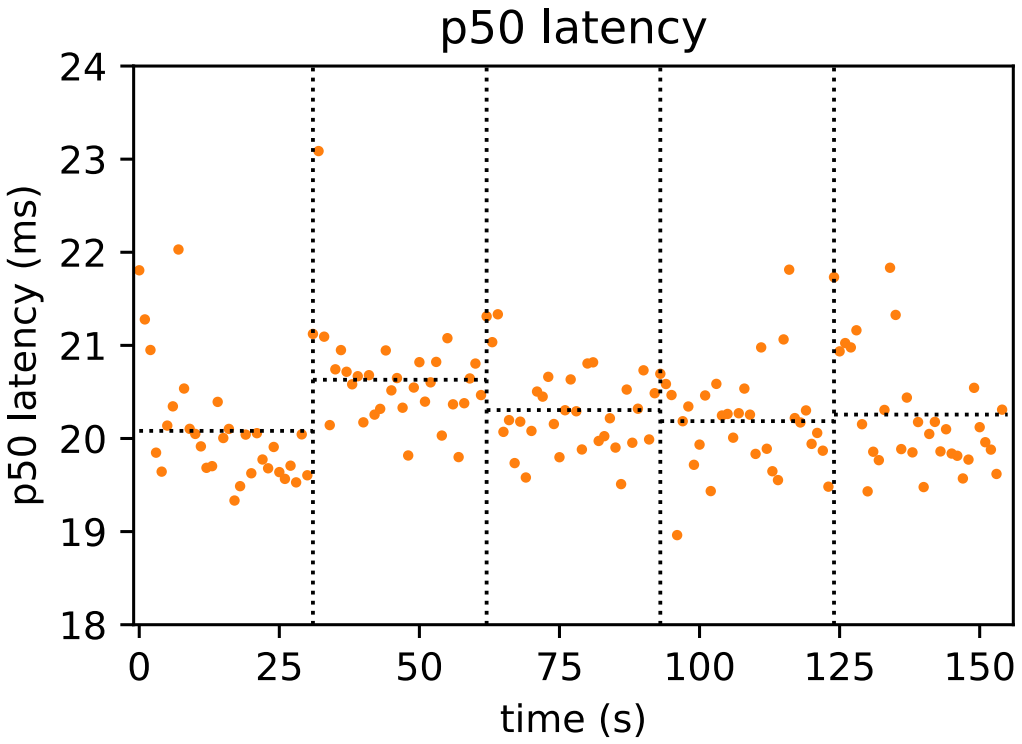
Impact of initial state on FFT benchmark results [3]


p50 latency
p50 latency (ms)
150
125
100
75
50
25
0
0
200
400
600
800
1000
time (s)
Warmup
Measurements


p50 latency
p50 latency (ms)
24
23
22
21
20
19
18
0
25
50
75
100
125
150
time (s)



Run #	mean ± stddev	coeff. of variation
1	20.08 ± 0.63 ms	3.16%
2	20.63 ± 0.56 ms	2.72%
3	20.31 ± 0.45 ms	2.23%
4	20.19 ± 0.54 ms	2.66%
5	20.26 ± 0.63 ms	3.11%
all	20.29 ± 0.60 ms	2.94%



Run #	mean ± stddev	coeff. of variation
1	20.08 ± 0.63 ms	3.16%
2	20.63 ± 0.56 ms	2.72%
3	20.31 ± 0.45 ms	2.23%
4	20.19 ± 0.54 ms	2.66%
5	20.26 ± 0.63 ms	3.11%
all	20.29 ± 0.60 ms	2.94%

Tip #3: Rerun benchmarks multiple times to reduce inter-run variation (M ≥ 5).

Tip #1: Run benchmarks for longer to uncover perturbations (e.g., warmup effects).

Tip #2: Collect enough samples to reduce intra-run variation (N ≥ 30).

Tip #3: Rerun benchmarks multiple times to reduce inter-run variation (M ≥ 5).

Coefficient of variation: 11.80% → **2.94%**

Tip #1: Run benchmarks for longer to uncover perturbations (e.g., warmup effects).

Tip #2: Collect enough samples to reduce intra-run variation (N ≥ 30).

Tip #3: Rerun benchmarks multiple times to reduce inter-run variation (M ≥ 5).

Coefficient of variation: 11.80% → **2.94%**

Tip #4: Use deterministic inputs.

Tip #1: Run benchmarks for longer to uncover perturbations (e.g., warmup effects).

Tip #2: Collect enough samples to reduce intra-run variation (N ≥ 30).

Tip #3: Rerun benchmarks multiple times to reduce inter-run variation (M ≥ 5).

Coefficient of variation: 11.80% → **2.94%**

Tip #4: Use deterministic inputs.

Tip #5: Use load generators that avoid the coordinated omission problem (e.g., k6).

Tip #1: Run benchmarks for longer to uncover perturbations (e.g., warmup effects).

Tip #2: Collect enough samples to reduce intra-run variation (N ≥ 30).

Tip #3: Rerun benchmarks multiple times to reduce inter-run variation (M ≥ 5).

Coefficient of variation: 11.80% → **2.94%**

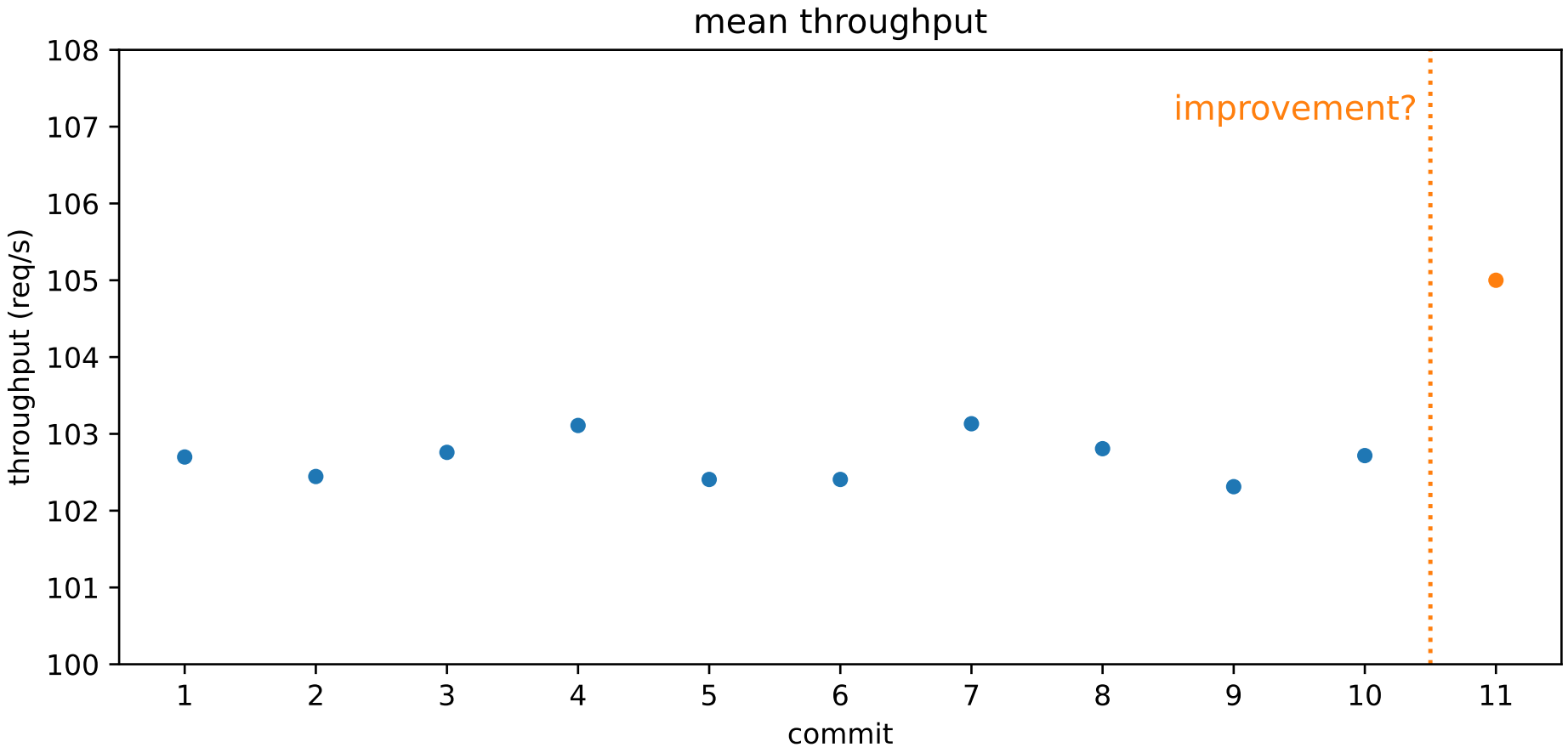
Tip #4: Use deterministic inputs.

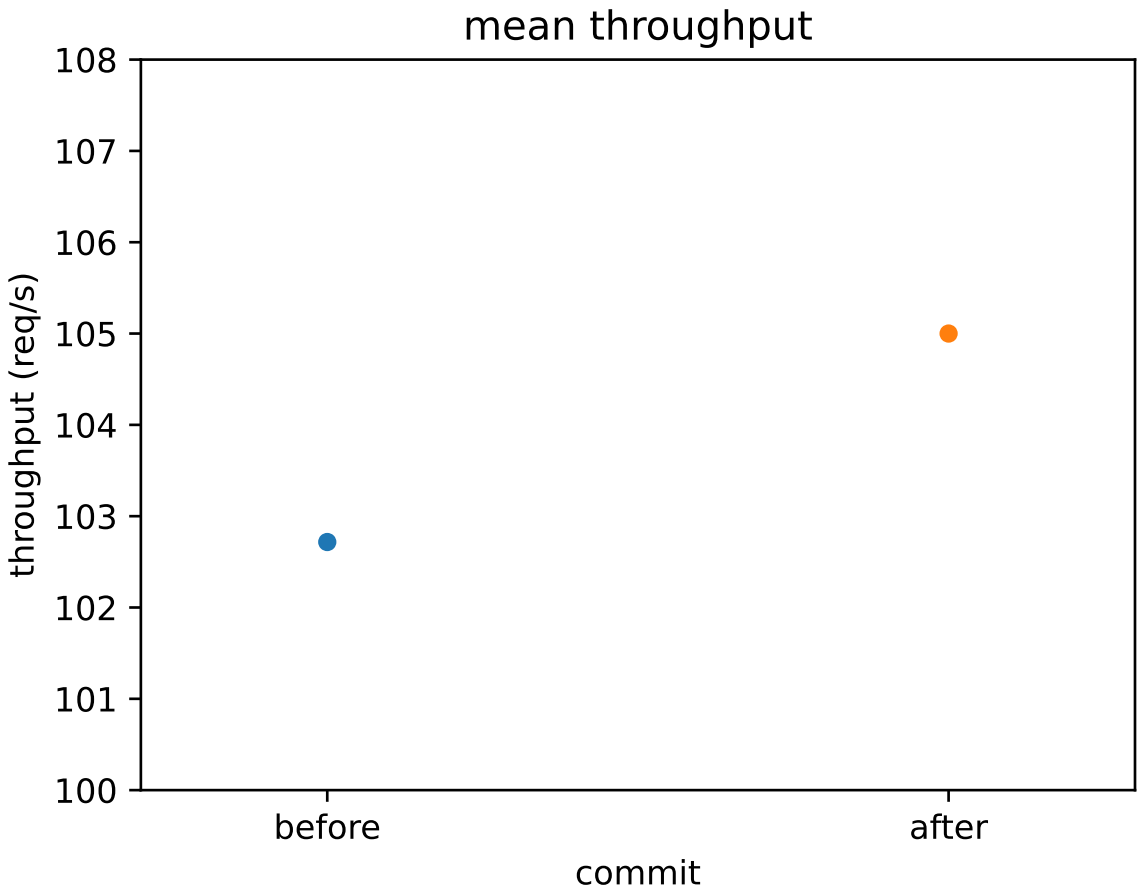
Tip #5: Use load generators that avoid the **coordinated omission** problem (e.g., k6).

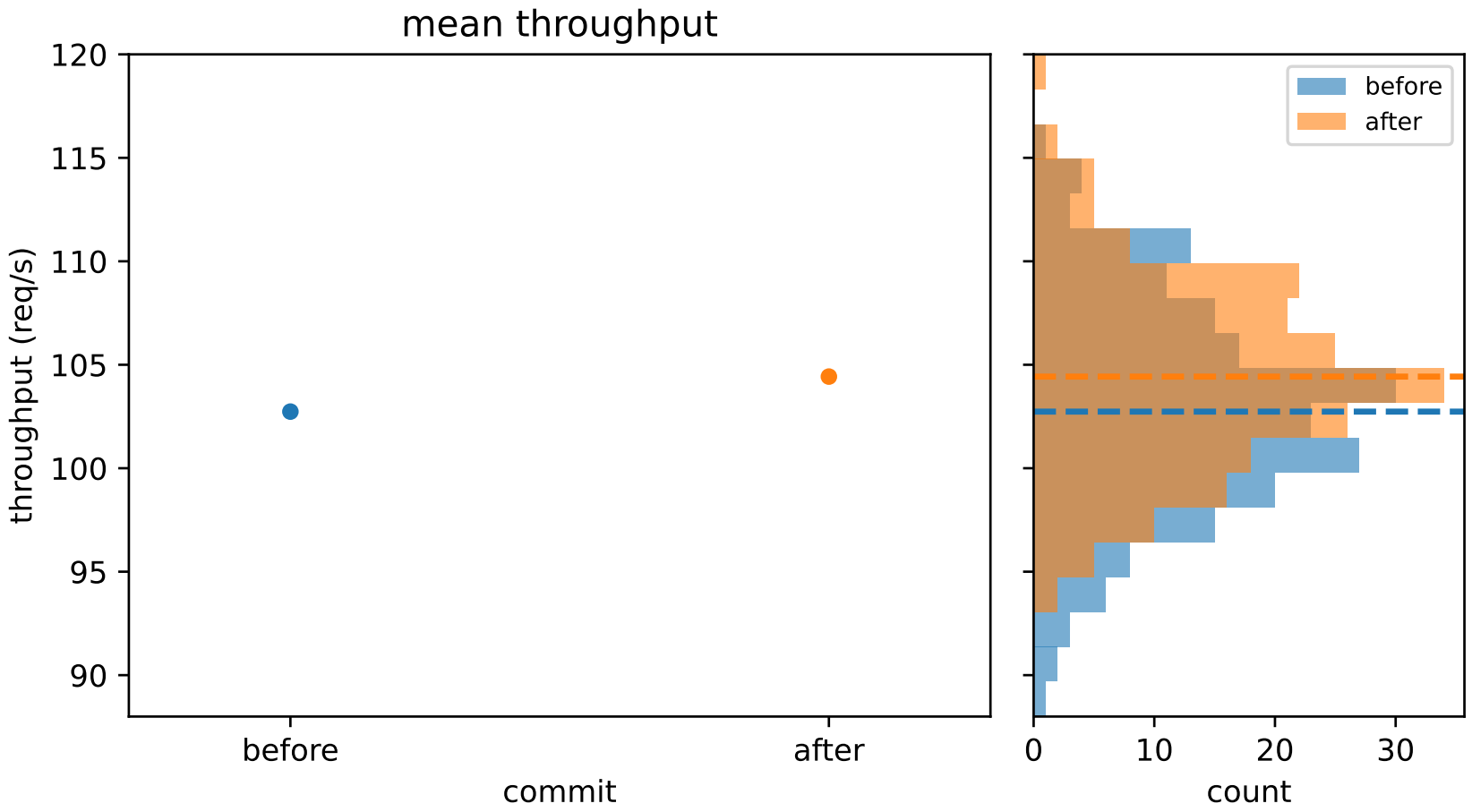
Slow system → load generator slows down → artificially better latencies.

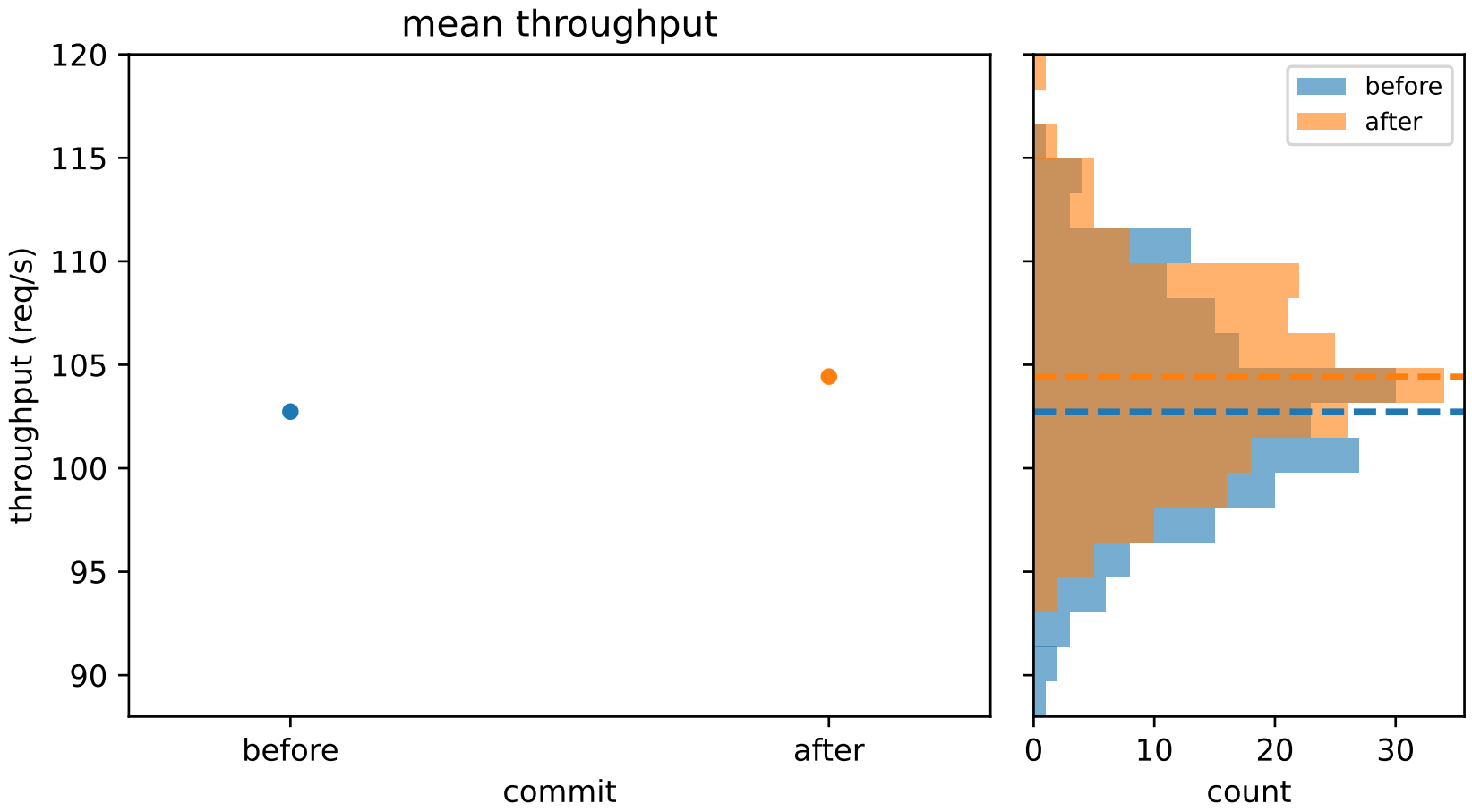
Gil Tene, "How NOT to Measure Latency" [4]

Interpreting Benchmark Results


mean throughput
improvement?
throughput (req/s)
108
107
106
105
104
103
102
101
100
1
2
3
4
5
6
7
8
9
10
11
commit


mean throughput
108
107
106
105
104
103
102
101
100
throughput (req/s)
before
after
commit


mean throughput
throughput (req/s)
120
115
110
105
100
95
90
before
after
commit
before
after
0
10
20
30
count



How can we tell if the difference is big enough?


INTUITION
MATHS
imgflip.com

$$\frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t > \text{critical value}$$

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$t >$ critical value

false positive rate

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t > \text{critical value}$$

$$\alpha = \text{false positive rate}$$

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t > \text{critical value}(\alpha)$$

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

t > critical value(α)

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

t > critical value(α)

$$t > t_{\alpha,\mathrm{df}}$$

$$t = \frac{\text{how big the difference is}}{\text{how big the noise is}}$$

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

t > critical value(α)

$$t > t_{\alpha,\mathrm{df}}$$

Hypothesis test (t-test).

Another approach: changepoint detection

Event	Speakers	Start	End
Sunday			
Accessible software performance	Alexander Zaitsev	09:00	09:50
Beyond nvidia-smi: Tools for Real GPU Performance Metrics	YASH PANCHAL	09:50	10:30
Keeping the P in HPC: the EESSI Way	Kenneth Hoste	10:30	11:10
Towards unified full-stack performance analysis and automated computer system design with Adaptyst	Maks Graczyk	11:10	11:50
How to Reliably Measure Software Performance	Kemal Akkoyun, Augusto de Oliveira	11:50	12:30
Pulling 100k revisions 100× faster	Raphaël Gomès, Pierre-Yves David	12:30	13:10
Database benchmarks: Lessons learned from running a benchmark standard organization	Gábor Szárnyas	13:10	13:50
Continuous Performance Engineering HowTo	Henrik Ingo	13:50	14:30
Writing an ultrafast Lua/JSON encoder+decoder as a LuaJIT module	Adam Ivora	14:30	15:10
How To Move Bytes Around	Alexey Milovidov	15:10	15:50
A Performance Comparison of Kubernetes Multi-Cluster Networking	josecastillolema, Raul	15:50	16:30
Load Testing Real React Applications for Production Performance	Mohammed Zubair Ahmed	16:30	17:00

Tip #1: Long enough benchmarks.

Tip #2: Enough samples (N ≥ 30).

Tip #3: Enough runs (M ≥ 5).

Tip #4: Deterministic inputs.

Tip #5: Avoid coordinated omission.

Tip #6: Use hypothesis testing to determine if improvements/regressions are statistically significant.

Tip #1: Long enough benchmarks.

Tip #2: Enough samples (N ≥ 30).

Tip #3: Enough runs (M ≥ 5).

Tip #4: Deterministic inputs.

Tip #5: Avoid coordinated omission.

Tip #6: Use hypothesis testing.

But what about inter-experiment variation?

Tip #1: Long enough benchmarks.

Tip #2: Enough samples (N ≥ 30).

Tip #3: Enough runs (M ≥ 5).

Tip #4: Deterministic inputs.

Tip #5: Avoid coordinated omission.

Tip #6: Use hypothesis testing.

But what about inter-experiment variation?

Tip #7: Control your benchmarking environment.

Tip #1: Long enough benchmarks.

Tip #2: Enough samples (N ≥ 30).

Tip #3: Enough runs (M ≥ 5).

Tip #4: Deterministic inputs.

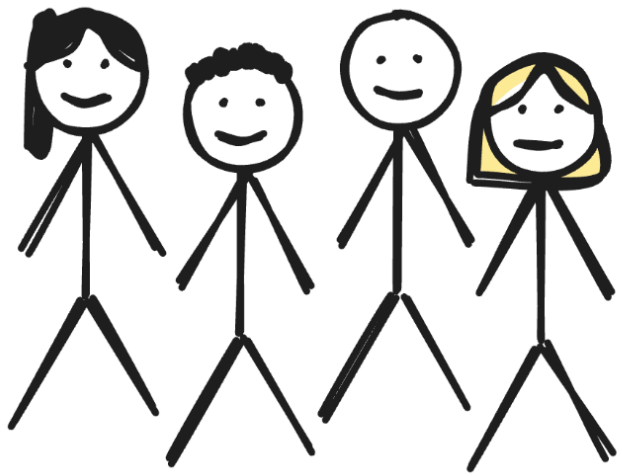
Tip #5: Avoid coordinated omission.

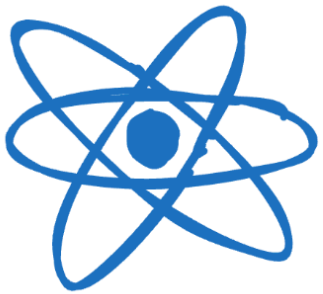
Tip #6: Use hypothesis testing.

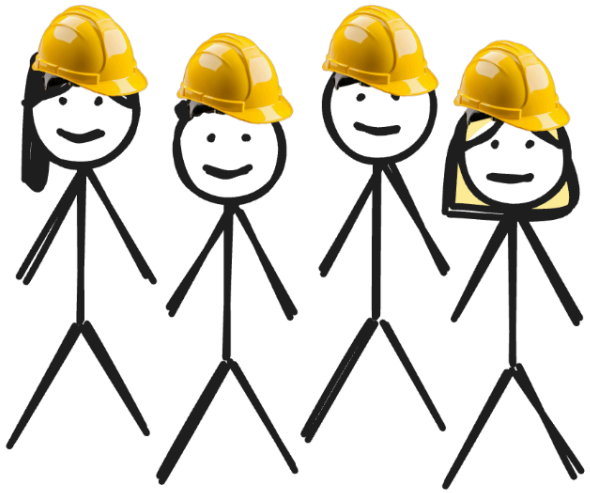
But what about inter-experiment variation?

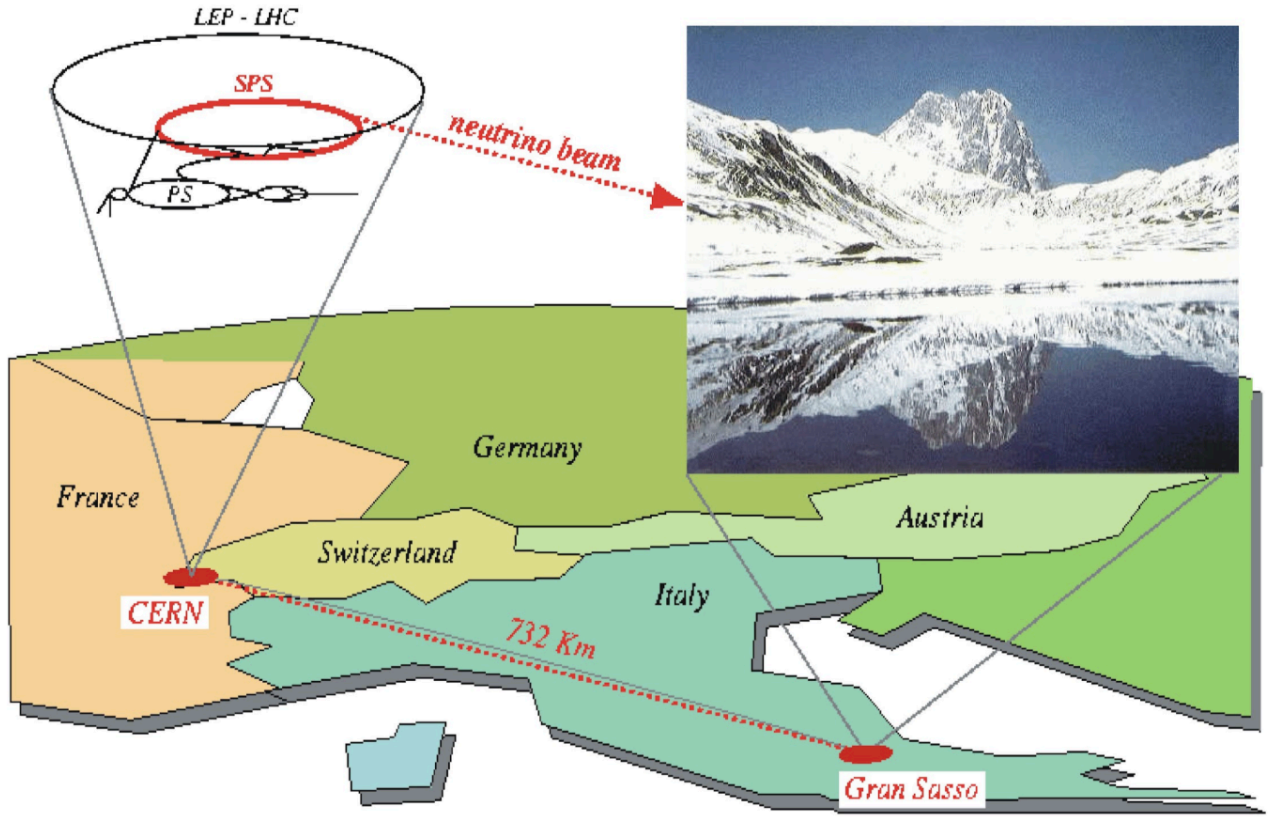
~~Tip #7~~**: Control your benchmarking environment.**
Tip #0

How to Control Your Benchmarking Environment








CERN to Gran Sasso Neutrino Beam
LEP - LHC
SPS
PS
neutrino beam
France
Germany
Switzerland
Austria
Italy
CERN
732 Km
Gran Sasso

[5]

5 years

~€100M 

[6, 7]

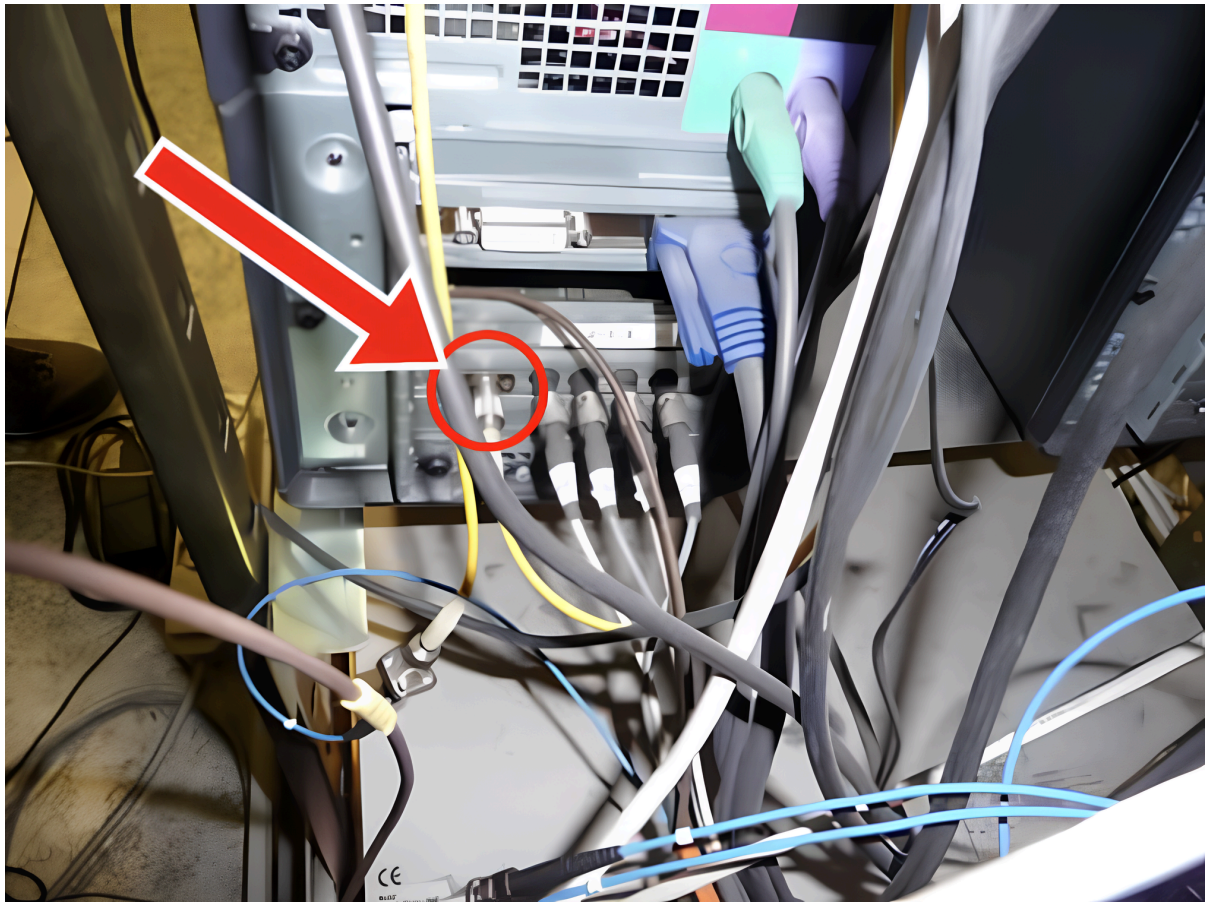
nature

News | Published: 22 September 2011

Particles break light-speed limit

Geoff Brumfiel

Nature (2011) | Cite this article



Loose fiber optic cable that caused the measurement error [8]

Most of us aren't building 730km tunnels.

But we deal with "loose cables" every day when measuring software performance.

Layer	Sources of Noise	Mitigations
External	Network Temperature Vibration Virtualization	Use dedicated on-prem hardware Use bare metal cloud instances
Application	Memory layout Compilation/linking	Set up fixed builds (e.g., disable ASLR)
Kernel	Scheduling Caching	Set CPU affinity Set process priority Warm up or drop caches
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Layer	Sources of Noise	Mitigations
External	Network Temperature Vibration Virtualization	Use dedicated on-prem hardware Use bare metal cloud instances
Application	Memory layout Compilation/linking	Set up fixed builds (e.g., disable ASLR)
Kernel	Scheduling Caching	Set CPU affinity Set process priority Warm up or drop caches
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Layer	Sources of Noise	Mitigations
External	Virtualization	Use bare metal cloud instances

Layer	Sources of Noise	Mitigations
External	Virtualization	Use bare metal cloud instances

Noisy neighbor problem.

Layer	Sources of Noise	Mitigations
External	Virtualization	Use bare metal cloud instances

Noisy neighbor problem.

Kernel- and CPU-layer mitigations require bare metal access.

Layer	Sources of Noise	Mitigations
Kernel	Scheduling Caching	Set CPU affinity Set process priority Warm up or drop caches

```
# Set CPU affinity
taskset -c 0 ./benchmark

# Set process priority
nice -n -5 ./benchmark

# Drop filesystem cache
echo 3 > /proc/sys/vm/drop_caches && sync
```

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Multiple *hardware threads* share the same core.

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Multiple *hardware threads* share the same core.

```
# Disable SMT
echo off > /sys/devices/system/cpu/smt/control
```

What's the impact of disabling SMT?

bare metal, dynamic frequency scaling (DFS) disabled

What's the impact of disabling SMT?

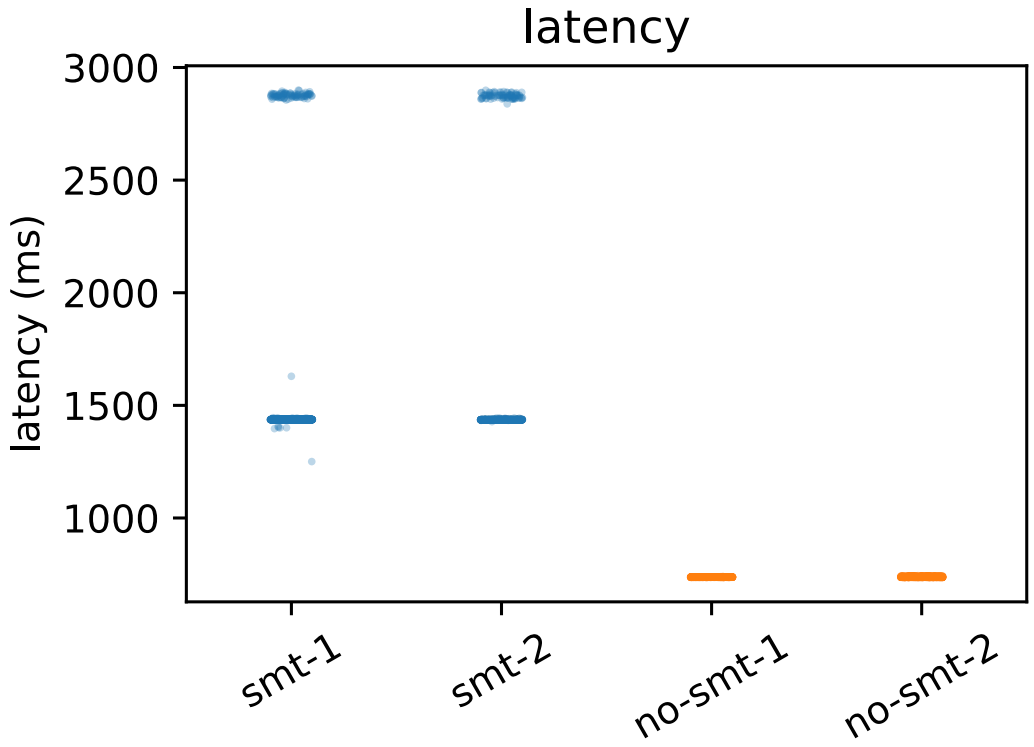
bare metal, dynamic frequency scaling (DFS) disabled

2 CPU-bound tasks, same core vs. separate cores

What's the impact of disabling SMT?

bare metal, dynamic frequency scaling (DFS) disabled

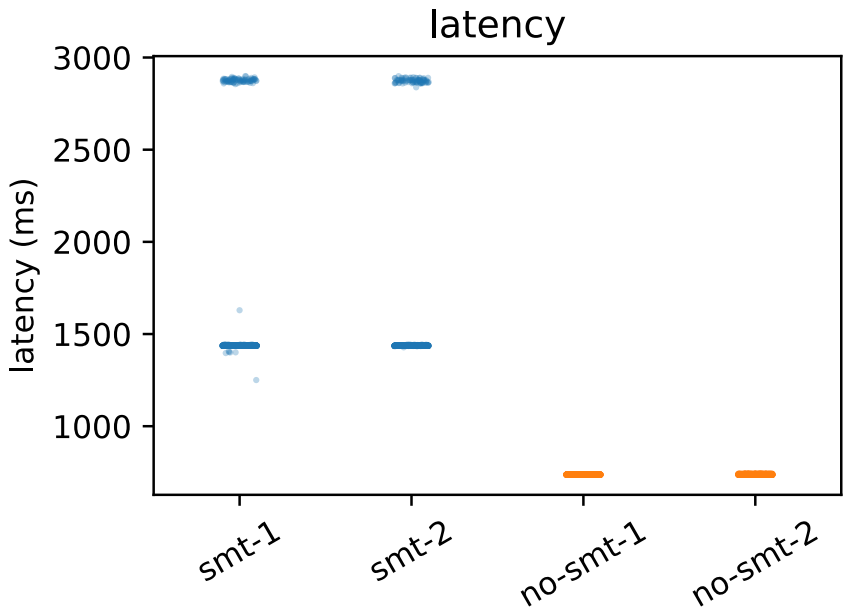
2 CPU-bound tasks, same core vs. separate cores



What's the impact of disabling SMT?

bare metal, dynamic frequency scaling (DFS) disabled

2 CPU-bound tasks, same core vs. separate cores

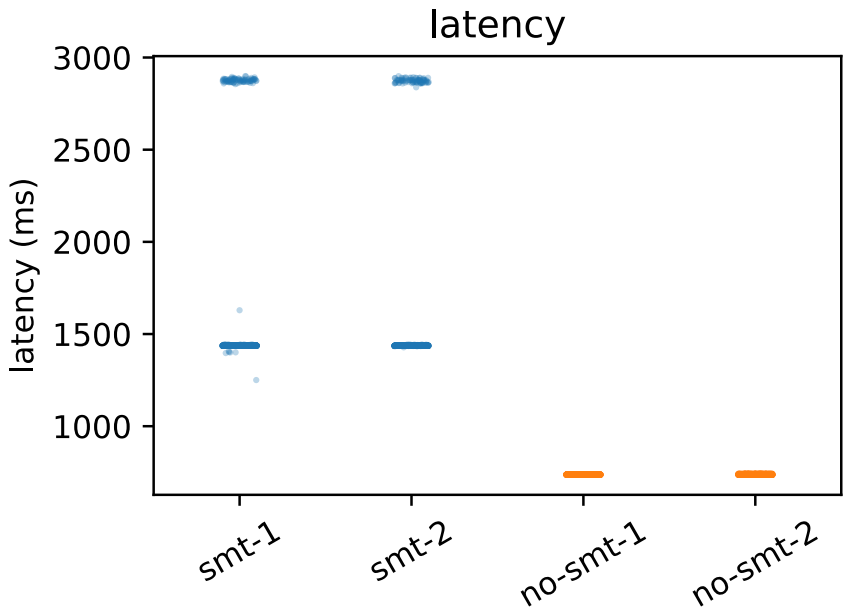


Task	mean ± stddev	coeff. of variation
smt-1	1537.64 ± 367.29 ms	23.887 %
smt-2	1536.88 ± 366.84 ms	23.869 %
no-smt-1	737.37 ± 0.32 ms	0.044 %
no-smt-2	737.93 ± 1.74 ms	0.235 %

What's the impact of disabling SMT?

bare metal, dynamic frequency scaling (DFS) disabled

2 CPU-bound tasks, same core vs. separate cores



Task	mean ± stddev	coeff. of variation
smt-1	1537.64 ± 367.29 ms	23.887 %
smt-2	1536.88 ± 366.84 ms	23.869 %
no-smt-1	737.37 ± 0.32 ms	0.044 %
no-smt-2	737.93 ± 1.74 ms	0.235 %

100x less variation

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Dynamic frequency scaling (DFS) adjusts the CPU frequency to match the workload.

Layer	Sources of Noise	Mitigations
CPU	Simultaneous multithreading (SMT) contention Dynamic frequency scaling (DFS)	Disable SMT Disable DFS

Dynamic frequency scaling (DFS) adjusts the CPU frequency to match the workload.

```
# Pin clock rate
echo 2500000 > /sys/devices/system/cpu/cpu*/cpufreq/scaling_max_freq

# Set scaling governor to "performance"
echo performance > /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

# Disable frequency boosting (Turbo-Boost, Intel CPUs only)
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
```

What's the impact of disabling DFS?

bare metal, simultaneous multithreading (SMT) disabled

What's the impact of disabling DFS?

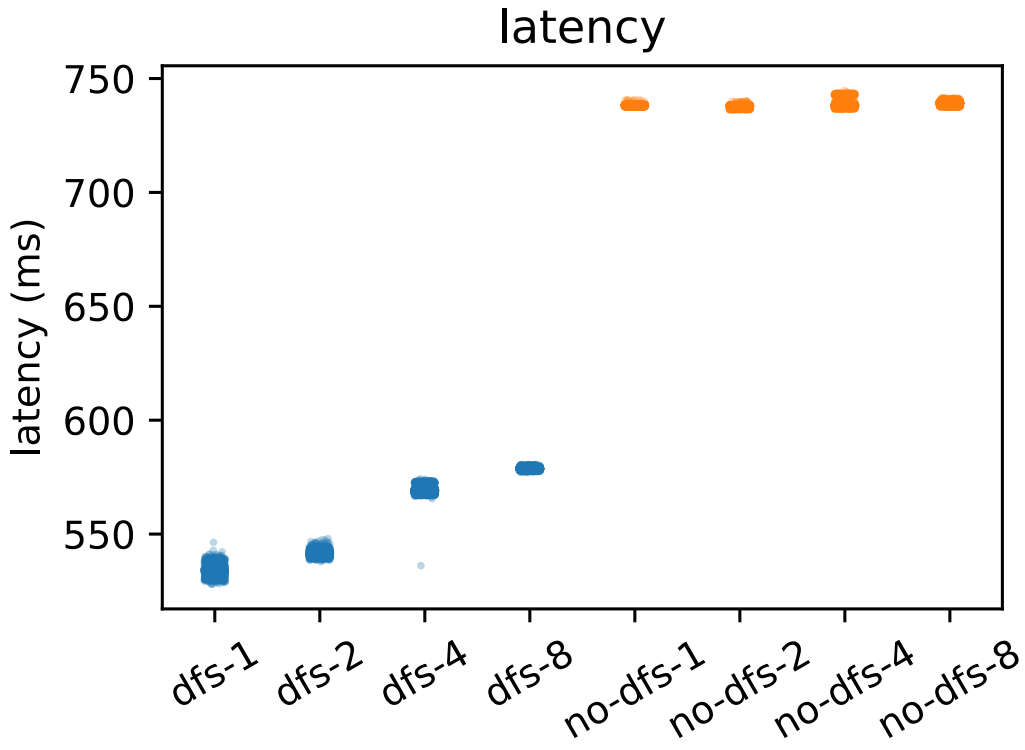
bare metal, simultaneous multithreading (SMT) disabled

Varying number of CPU-bound tasks, same core, DFS on vs. DFS off

What's the impact of disabling DFS?

bare metal, simultaneous multithreading (SMT) disabled

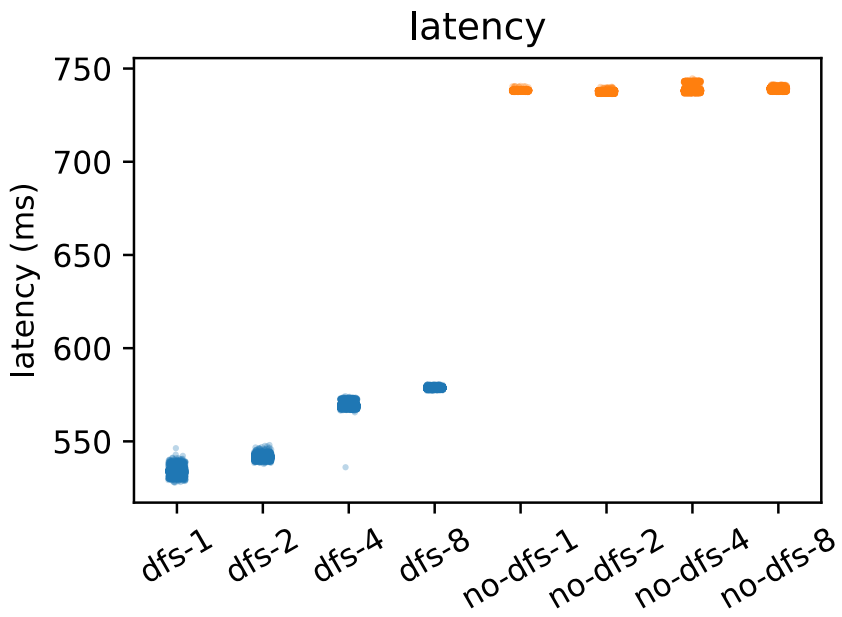
Varying number of CPU-bound tasks, same core, DFS on vs. DFS off



What's the impact of disabling DFS?

bare metal, simultaneous multithreading (SMT) disabled

Varying number of CPU-bound tasks, same core, DFS on vs. DFS off

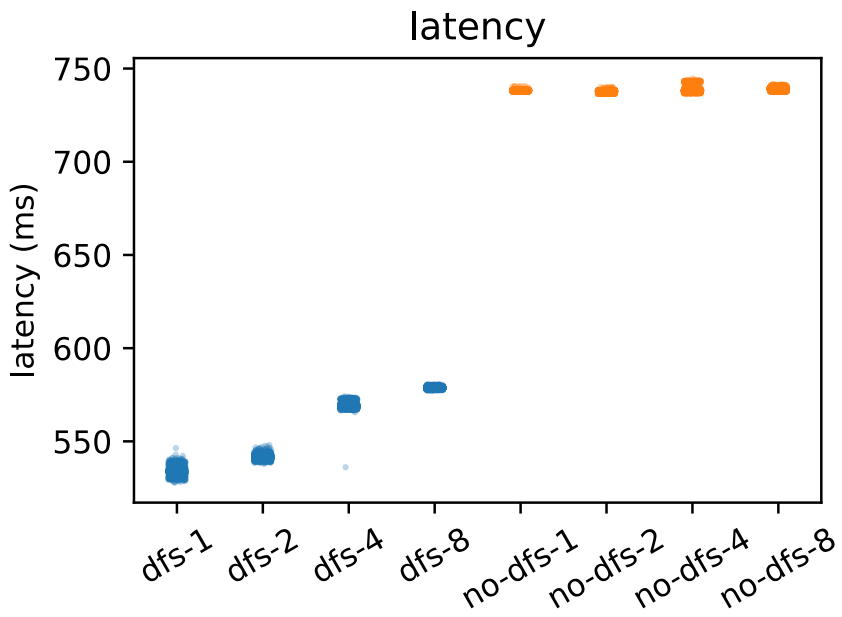


Task	mean ± stddev	coeff. of variation
dfs-1	533.97 ± 2.046 ms	0.383 %
dfs-8	578.67 ± 0.287 ms	0.050 %
no-dfs-1	738.18 ± 0.306 ms	0.041 %
no-dfs-8	739.18 ± 0.351 ms	0.047 %

What's the impact of disabling DFS?

bare metal, simultaneous multithreading (SMT) disabled

Varying number of CPU-bound tasks, same core, DFS on vs. DFS off



Task	mean ± stddev	coeff. of variation
dfs-1	533.97 ± 2.046 ms	0.383 %
dfs-8	578.67 ± 0.287 ms	0.050 %
no-dfs-1	738.18 ± 0.306 ms	0.041 %
no-dfs-8	739.18 ± 0.351 ms	0.047 %

10x less variation

SMT and DFS experiments by Dmytro Yurchenko

CPU-level tweaks at Denis Bakhvalov's
"Performance Analysis and Tuning on Modern CPUs" [1]

Layer	Sources of Noise	Mitigations
External	Vibration	Don't shout in the datacenter

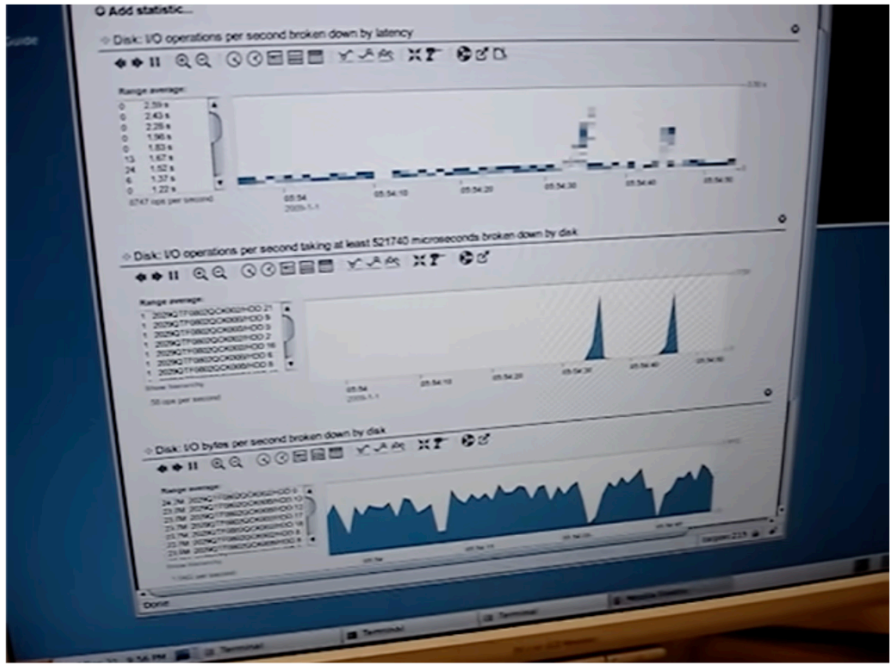
🔗 Shouting in the Datacenter

Shouting in the Datacenter

Bryan Cantrill
8.08K subscribers

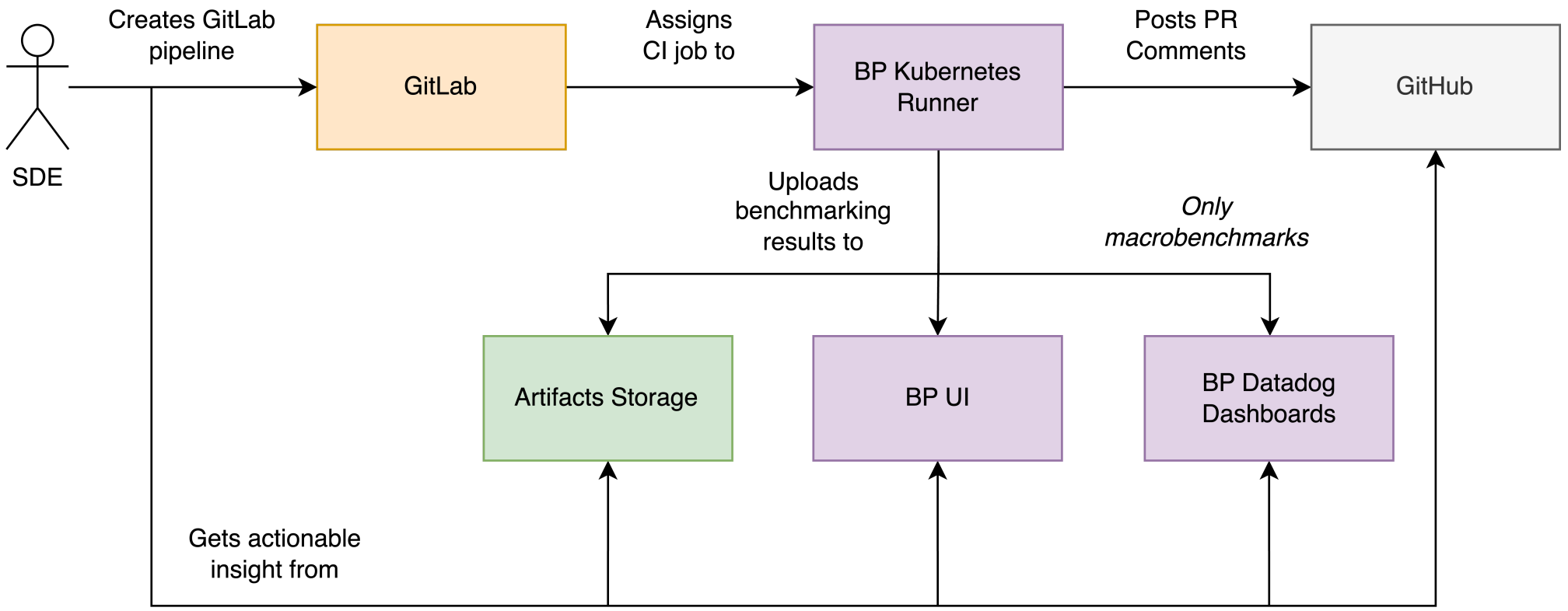
Subscribe

4.8M views 17 years ago
Brendan Gregg from Sun's Fishworks team makes an interesting discovery about inducing disk latency. For a ca. 2020 retrospective on this 2008 video: • Bryan Cantrill talks Sun Microsystems, DTr ...more

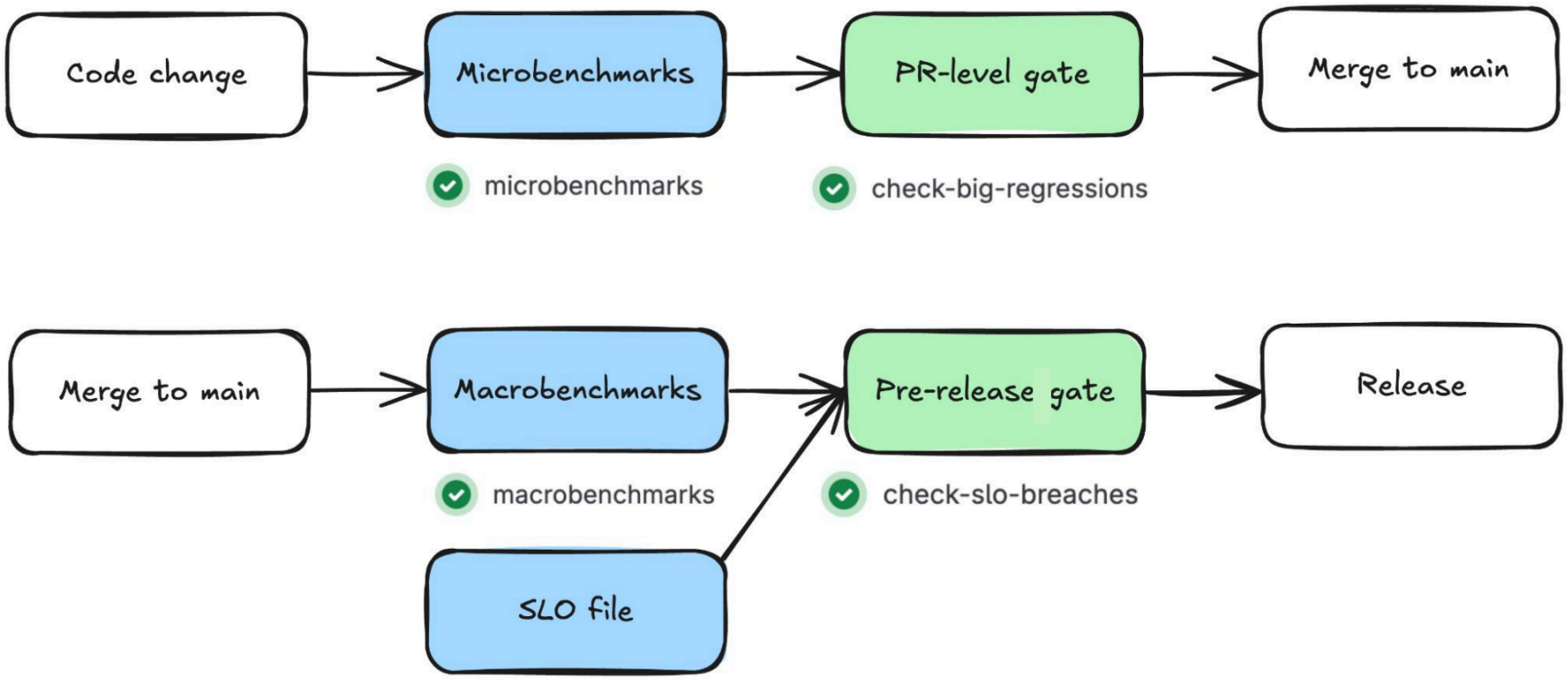


Integrating Benchmarks Into Your Workflows

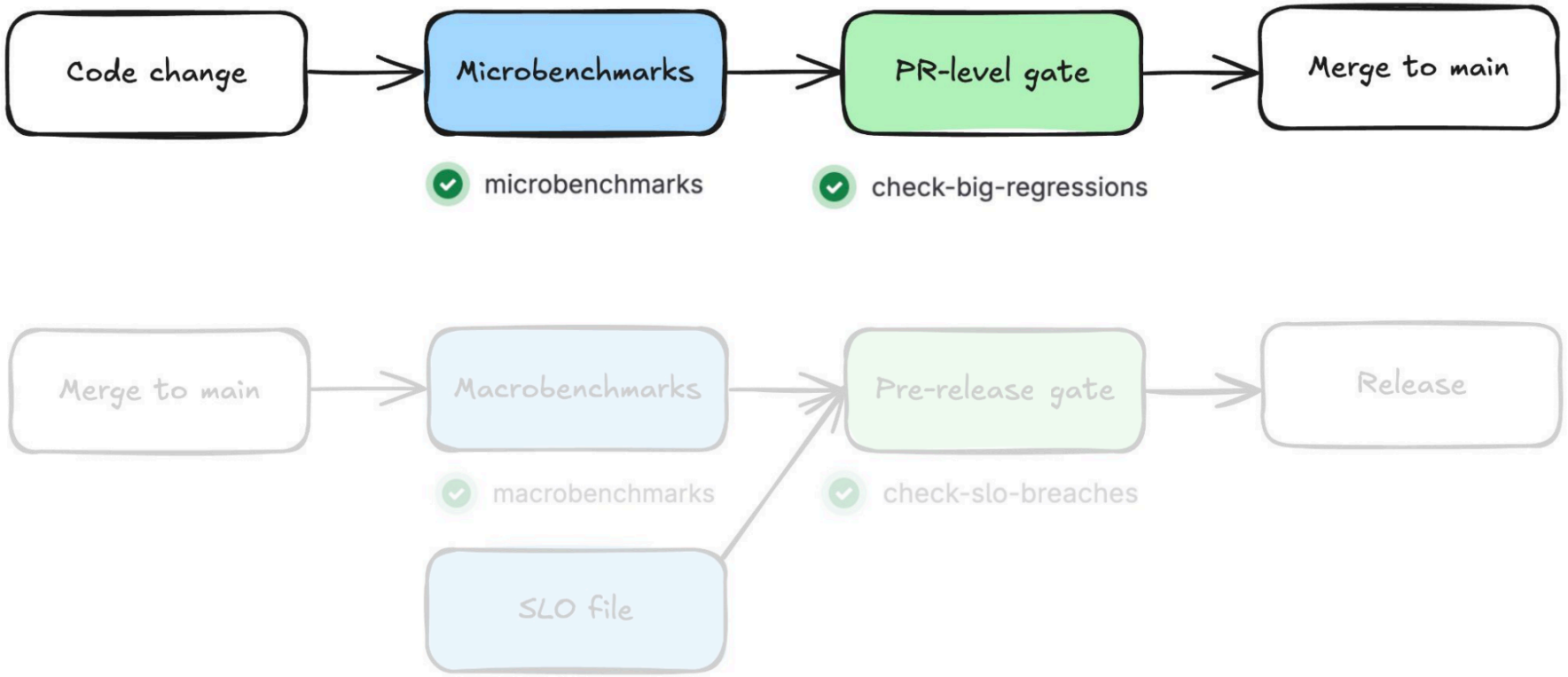
Architecture Overview



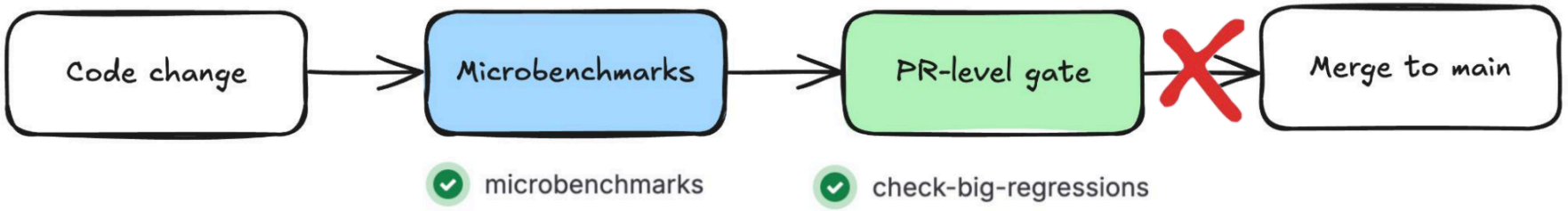
Feedback Loop



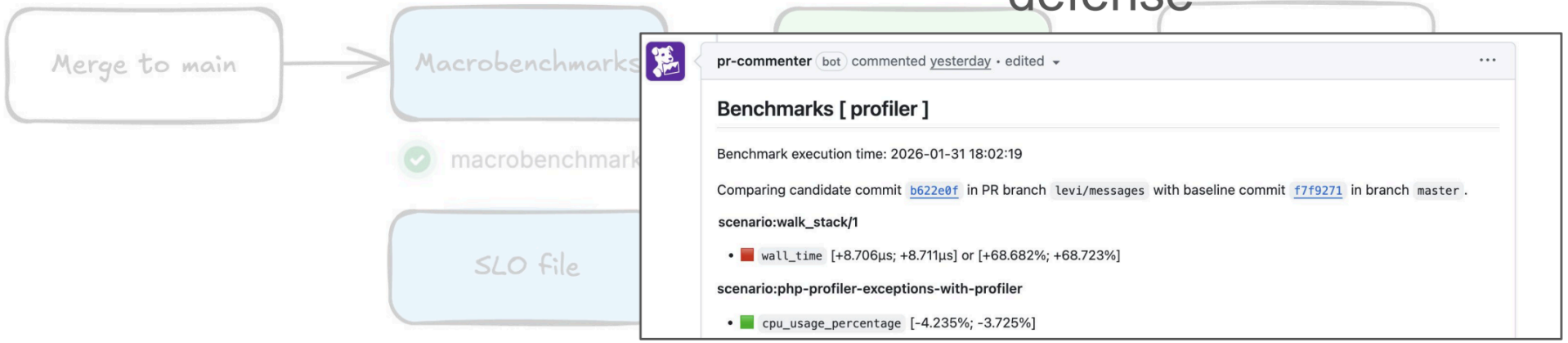
Feedback Loop



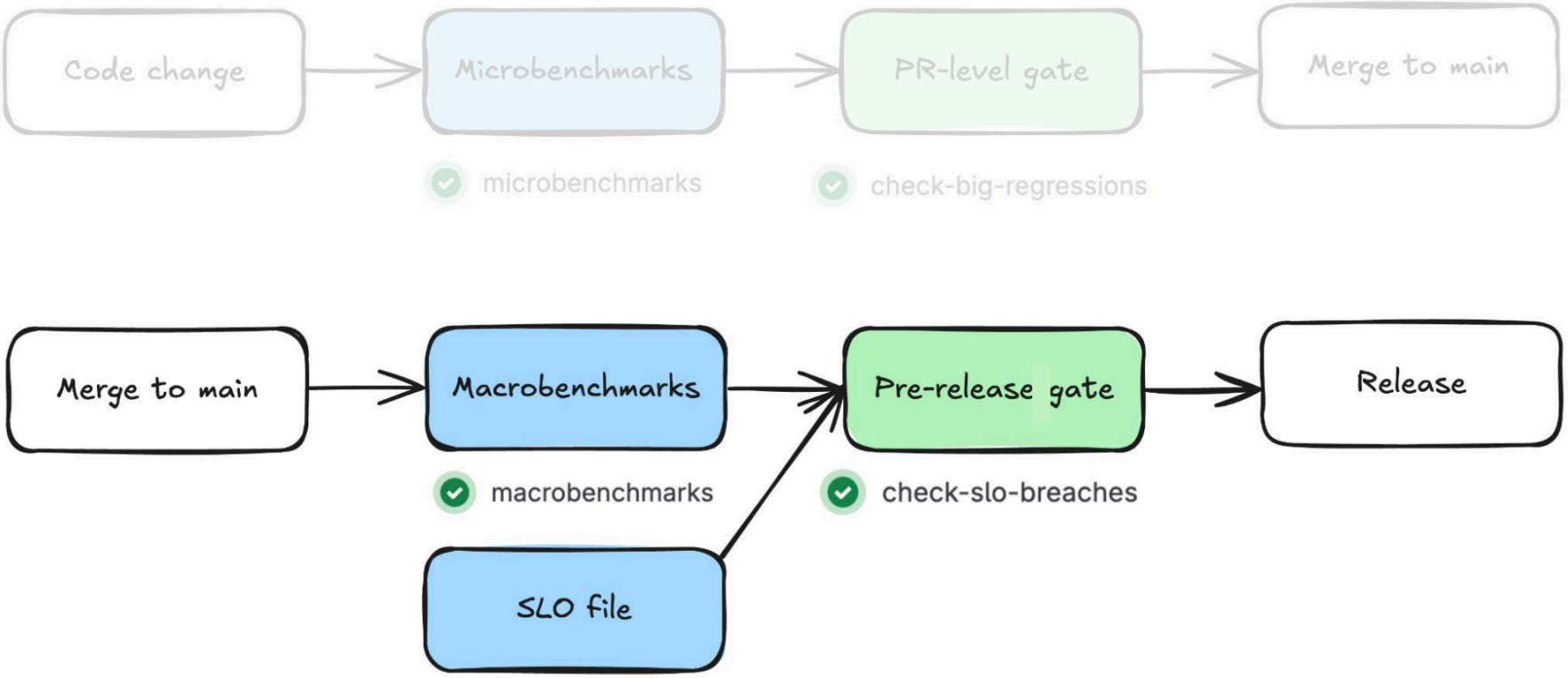
Feedback Loop



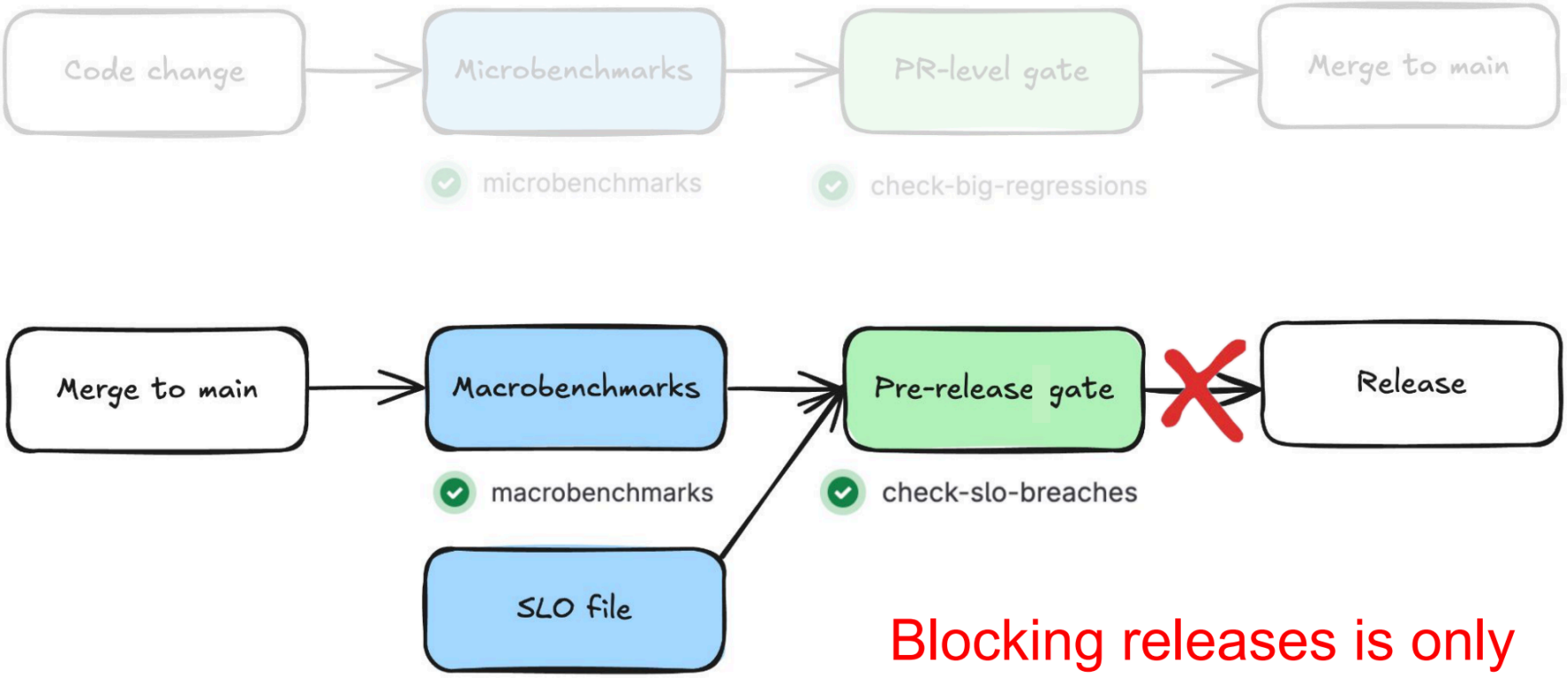
Blocking PRs is the main line of defense



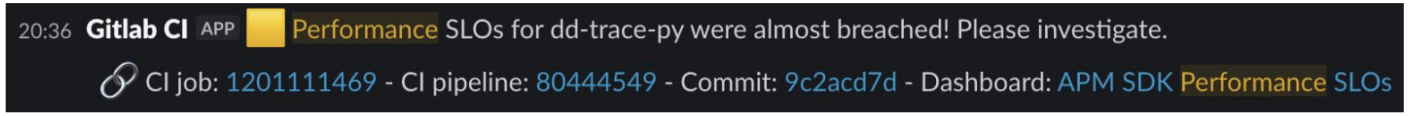
Feedback Loop



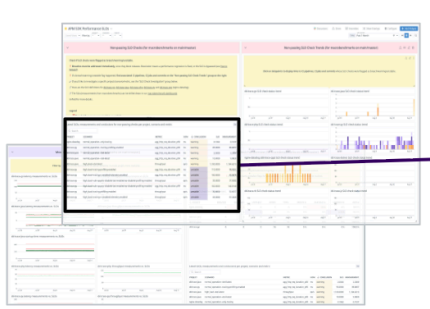
Feedback Loop



Blocking releases is only the last line of defense



Feedback Loop



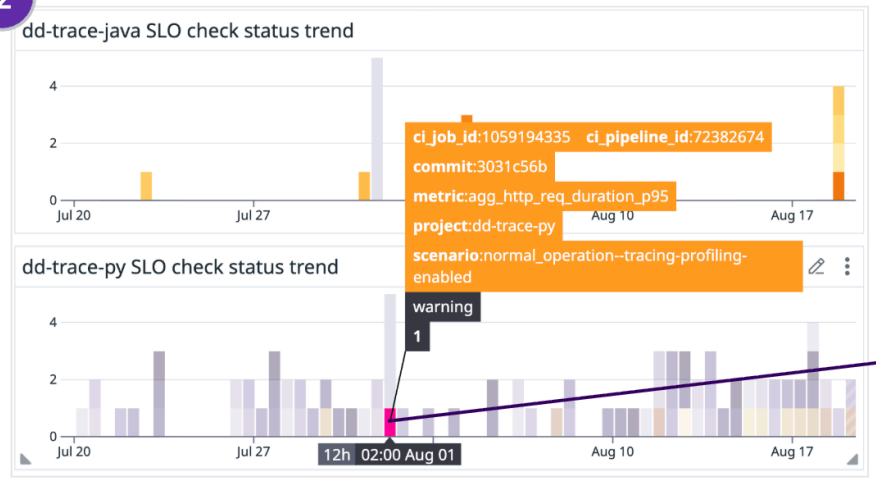
1

Latest SLOs, measurements and conclusions for non-passing checks per project, scenario and metric

Search

PROJECT	SCENARIO	METRIC	UOM	CONCLUSION	SLO	MEASUREMENT
nginx-datadog	normal_operation--only-tracing	agg_http_req_duration_p50	ms	warning	0.1062	0.1037
dd-trace-py	normal_operation--tracing-profiling-en...	agg_http_req_duration_p95	ms	warning	95.0000	89.8457

2



3

METRIC
exclude_null(sum:apm_sdk_slo.warning{project:dd-trace-py AND (branc... branch:master) AND (scenario:high* OR scenario:normal*)}.as_count())

View CI pipeline

View CI job

View commit

Open Source Tools

Start running benchmarks continuously today:

- bencher.dev - Continuous benchmarking platform
- hyperfine - CLI benchmark tool
- github-action-benchmark - GitHub Action
- chronologer - Benchmark tracking

Conclusion

Key Takeaways

1. **Control your benchmarking environment**

   Bare metal, isolation, disable SMT, disable DFS

Key Takeaways

1. **Control your benchmarking environment**

   Bare metal, isolation, disable SMT, disable DFS

2. **Design your benchmarks**

   Representative and repeatable

Key Takeaways

1. **Control your benchmarking environment**
   Bare metal, isolation, disable SMT, disable DFS
2. **Design your benchmarks**
   Representative and repeatable
3. **Interpret benchmark results**
   Statistics matter (hypothesis testing)

Key Takeaways

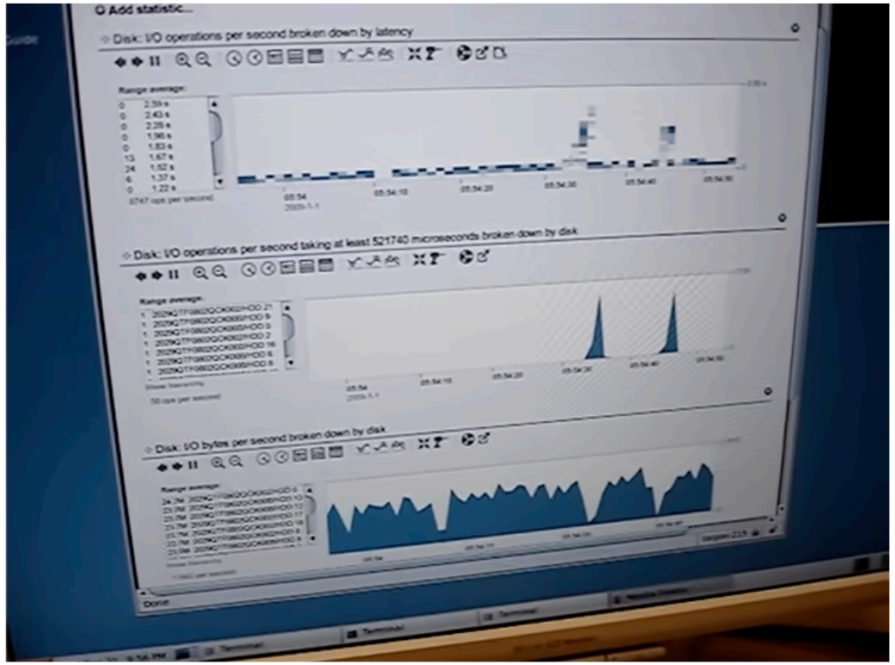
1. **Control your benchmarking environment**
   Bare metal, isolation, disable SMT, disable DFS
2. **Design your benchmarks**
   Representative and repeatable
3. **Interpret benchmark results**
   Statistics matter (hypothesis testing)
4. **Integrate benchmarks into your workflows**
   Run continuously, catch regressions early

Shouting in the Datacenter

Bryan Cantrill
8.08K subscribers

Subscribe

4.8M views 17 years ago
Brendan Gregg from Sun's Fishworks team makes an interesting discovery about inducing disk latency. For a ca. 2020 retrospective on this 2008 video: • Bryan Cantrill talks Sun Microsystems, DTr ...more



Don't shout in the datacenter

Thanks!

References

[1] Bakhvalov, D. (2020). *Performance Analysis and Tuning on Modern CPUs*. https://github.com/dendibakh/perf-book. Accessed Jan 2026.

[2] Leiserson, C. et al. (2020). "There's plenty of room at the Top: What will drive computer performance after Moore's law?" *Science*, 368(6495).

[3] Kalibera, T., Bulej, L., and Tuma, P. (2005). "Benchmark Precision and Random Initial State." In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 182-196. SCS.

[4] Tene, G. (2015). "How NOT to Measure Latency." https://www.youtube.com/watch?v=lJ8ydIuPFeU. Accessed Jan 2026.

[5] Universität Münster. "Neutrino oscillations in the neutrino beam from CERN to Gran Sasso." https://www.uni-muenster.de/Physik.KP/en/AGFrekers/forschung/opera.html. Accessed Jan 2026.

[6] CERN. (1999). "From Geneva to Gran Sasso in 2.5 milliseconds!". https://home.cern/news/press-release/cern/geneva-gran-sasso-25-milliseconds. Accessed Jan 2026.

[7] Wikipedia. "OPERA experiment." https://en.wikipedia.org/wiki/OPERA_experiment. Accessed Jan 2026.

[8] Strassler, M. (2012). "OPERA: What Went Wrong." https://profmattstrassler.com/articles-and-posts/particle-physics-basics/neutrinos/neutrinos-faster-than-light/opera-what-went-wrong/. Accessed Jan 2026.

[9] Gregg, B. (2020). *Systems Performance: Enterprise and the Cloud*, 2nd ed. Addison-Wesley. Chapter 2.8, "Visualizations."

[10] Valles, A. (2009). "Performance Insights to Intel Hyper-Threading Technology." https://web.archive.org/web/20150217050949/https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/. Accessed Jan 2026.

[11] Gregg, B. (2014). "Frequency Trails: Outliers." https://www.brendangregg.com/FrequencyTrails/outliers.html#Causes. Accessed Jan 2026.

[12] Gregg, B. (2020). "Systems Performance: Enterprise and the Cloud.", p. 233, "P-states and C-states."

[13] Humenay, E., Tarjan, D., and Skadron, K. (2007). "Impact of Process Variations on Multicore Performance Symmetry."

[14] Linux Kernel Documentation. "CPUFreq Governors." https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt. Accessed Jan 2026.

[15] ArchWiki. "CPU frequency scaling." https://wiki.archlinux.org/title/CPU_frequency_scaling. Accessed Jan 2026.

[16] Intel. "Intel Server Board and System Products Update on Intel Turbo Boost Technology Support with Low Power Intel Xeon Processor 3400/5500/5600 Series." https://cdrdv2-public.intel.com/840590/white_paper_turbo_boost_on_low_power_processor.pdf. Accessed Jan 2026.