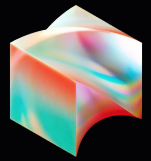
Improving CI Performance



Michael Yoon
Block Inc.

Why CI Performance?

We survey our developers quarterly

They said build are CI builds are slow

What part of CI would slow them down?

The CI Build running on their Pull Requests.

Therefore PR Build Duration / Walltime

Goal:

Reduce PR Build Walltime

The elapsed duration from start of the build to the end

Non-Goals:

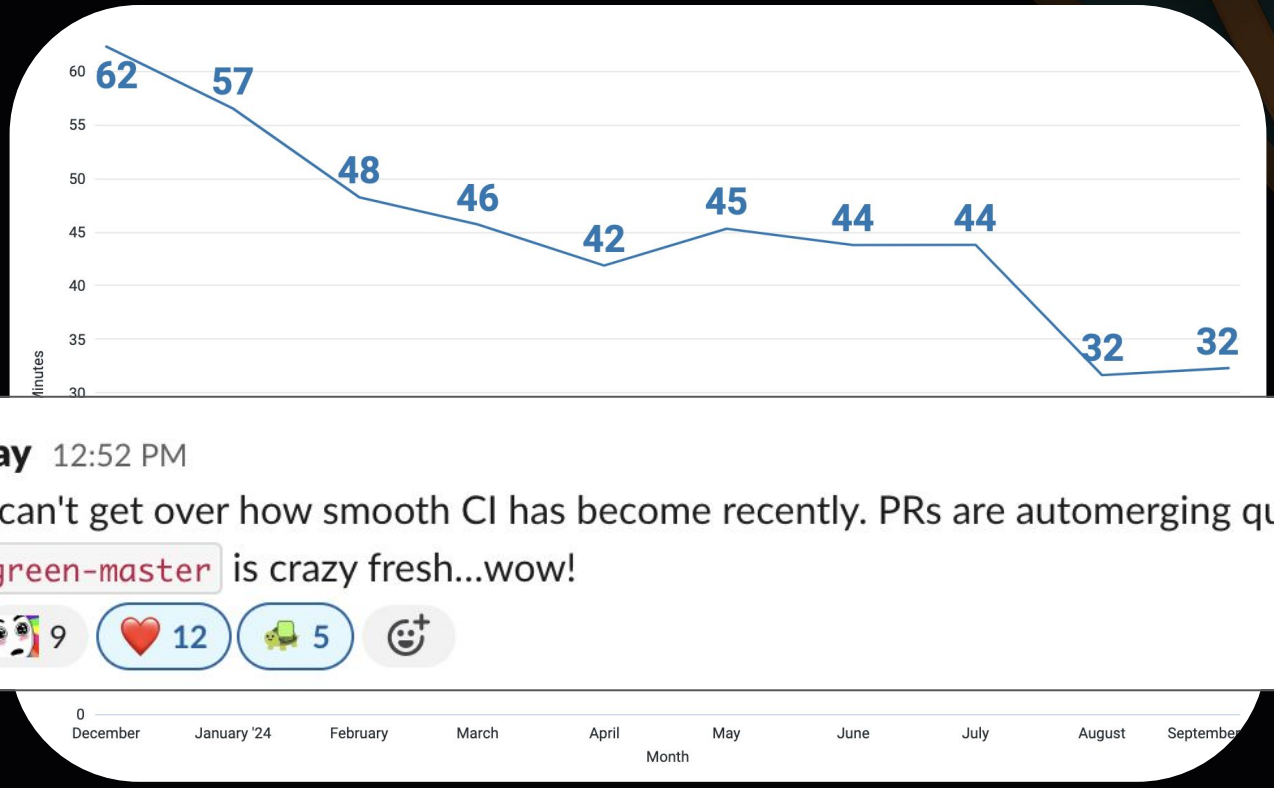
Optimizing CI Cost

Decreasing total jobs duration

Post-merge CI Builds

PR Build Wall-time 2024

>10,000 hrs saved per year



Context

200 Developers

Point-of-sale Android megarepo
> 6000 Gradle Modules
> 5 Million Lines of Code

Products:

- 7 apps
- Mobile, Tablet
- Register, Terminal

Our Average PR Build:

- 750 shards (Up to 1500)
- Types of jobs: checks, lints. App and Test compilation. unit tests, snapshot tests, UI tests.
- 60 worker hours (excluding UI Test Execution, which runs on emulator farms)
- Started with ~60 min wall-time duration
- 100 PRs per day

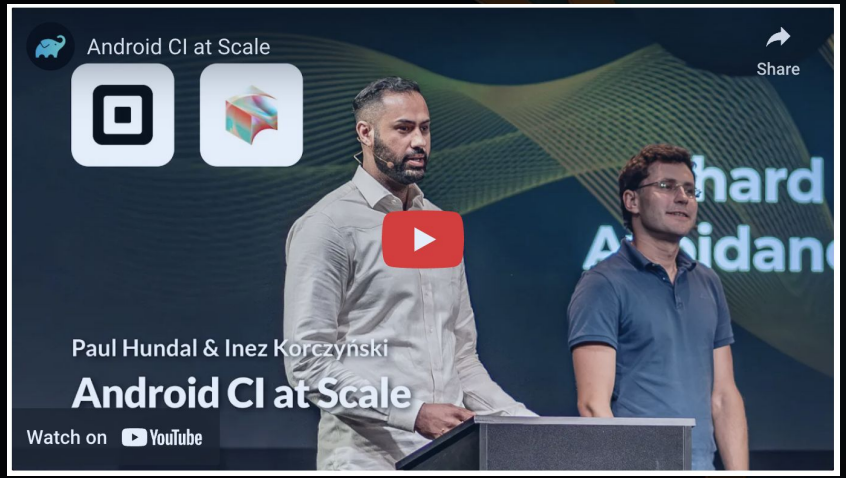
Our CI System: Kochiku

- Built in-house over a decade ago
- Runners on AWS EC2 Instances
- Ephemeral containers for each job

For More Context...

Watch “Android CI at Scale” from Inez and Paul last years DPE Summit on YouTube”

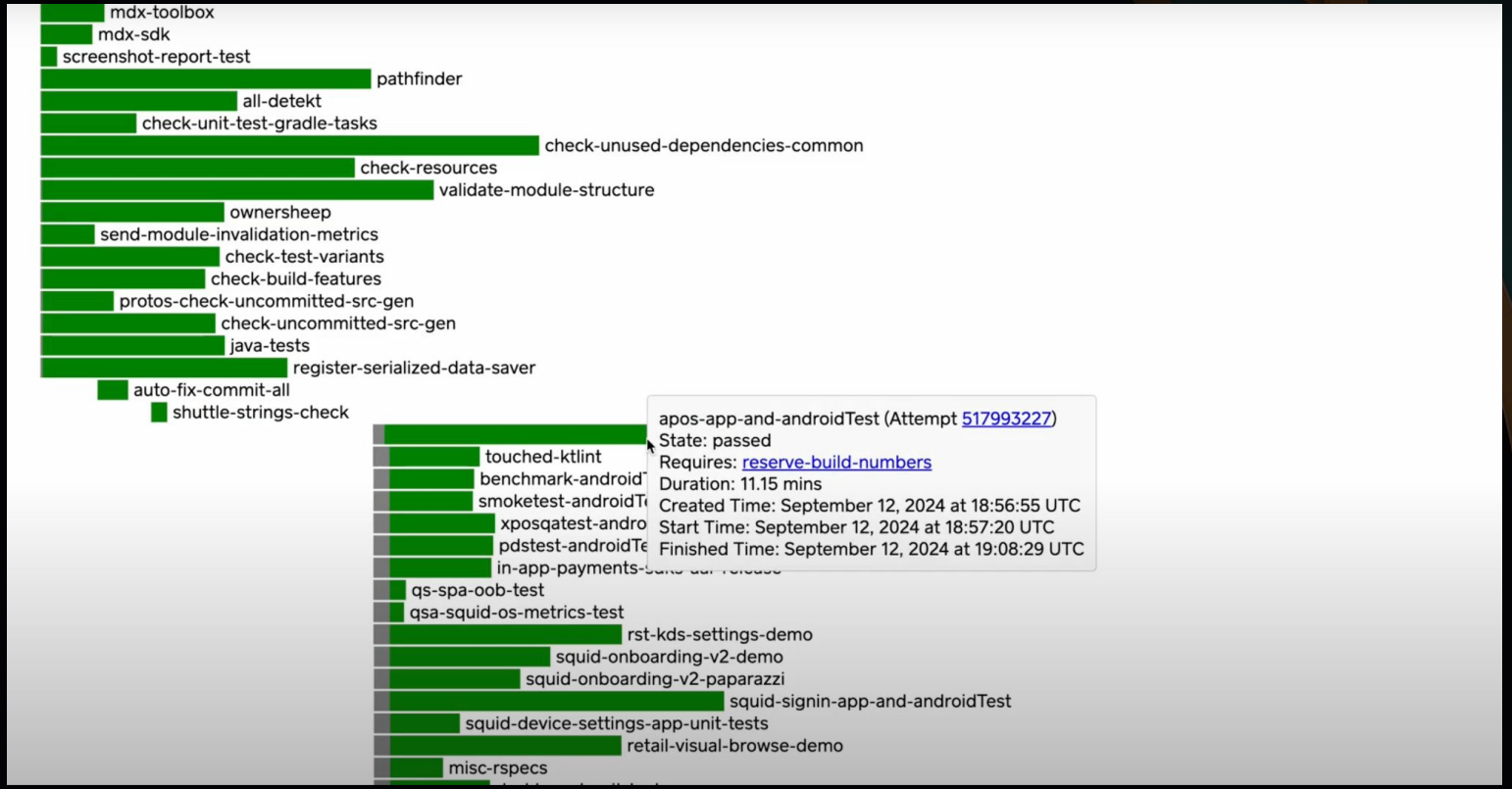
- Shard Avoidance
- UI Test Avoidance
- Strategy for Scaling Git



DPE SUMMIT

Visualize the CI Build

Build Waterfall View

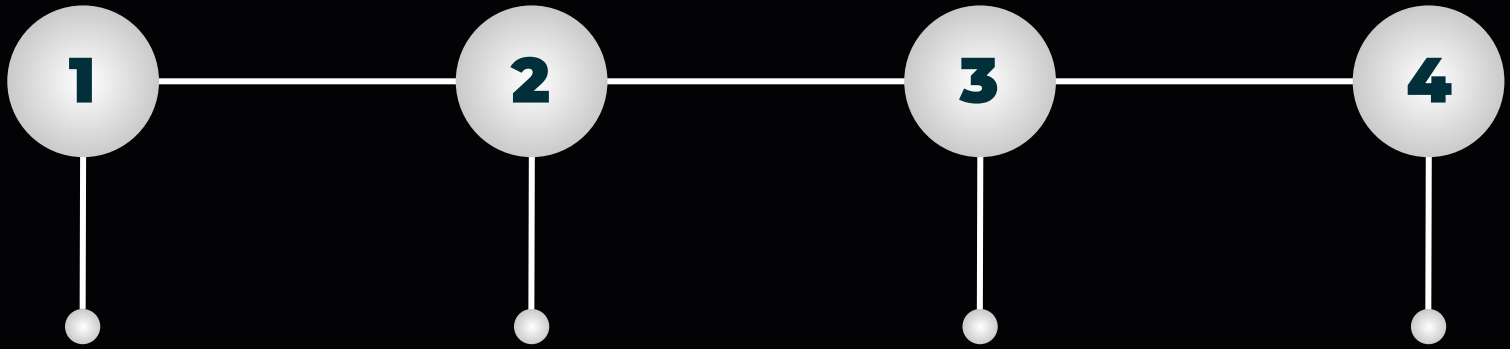


Waterfall: Load some “bad builds” and investigate

- Some problems become very obvious
- Majority of shards are inconsequential to walltime
- “Long poles”
  - Monitor the shards that finished last
- Great for finding clues. But quantify the problems and opportunities after

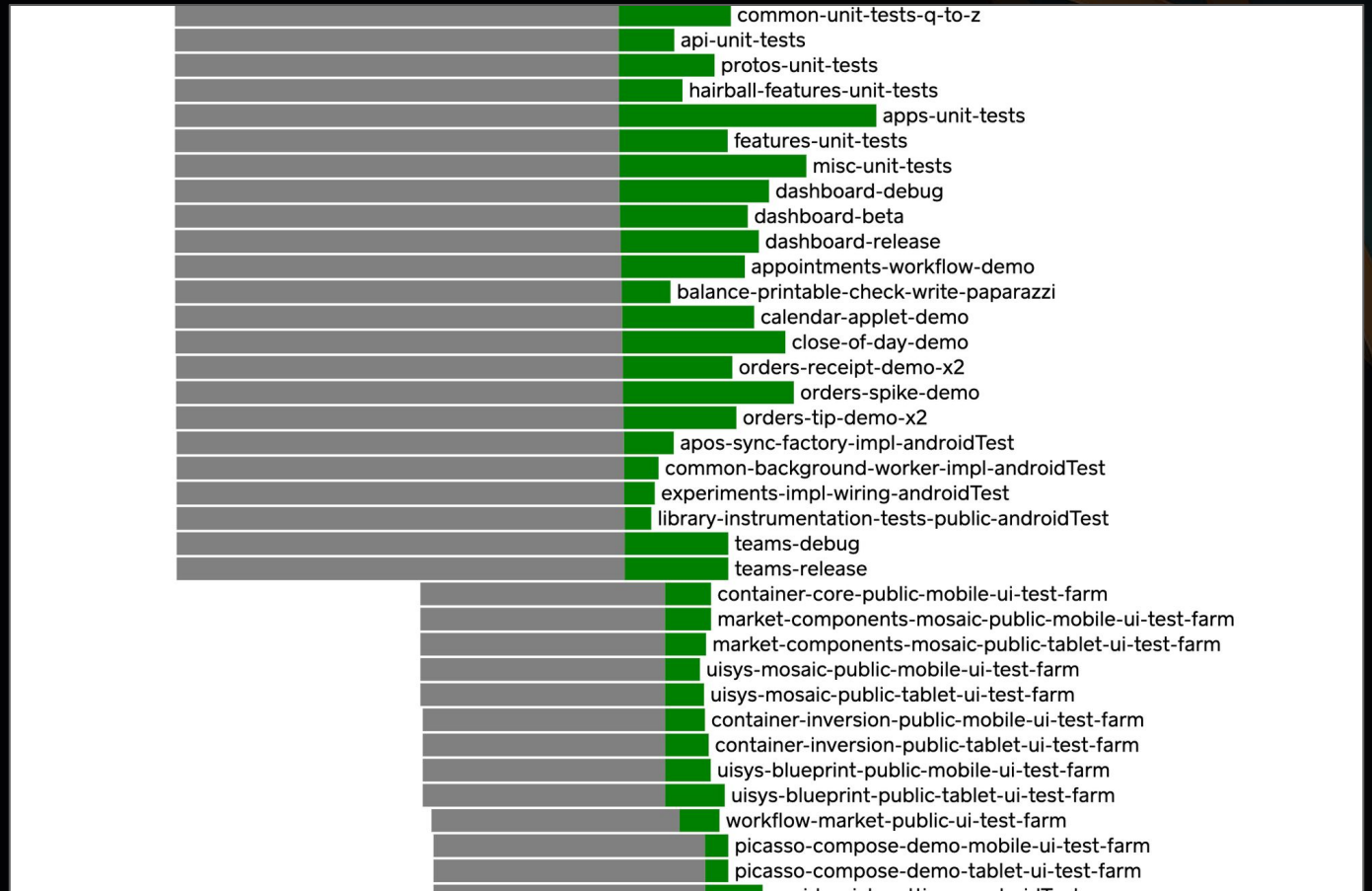
PR Build Wall-time Improvements

Improvements

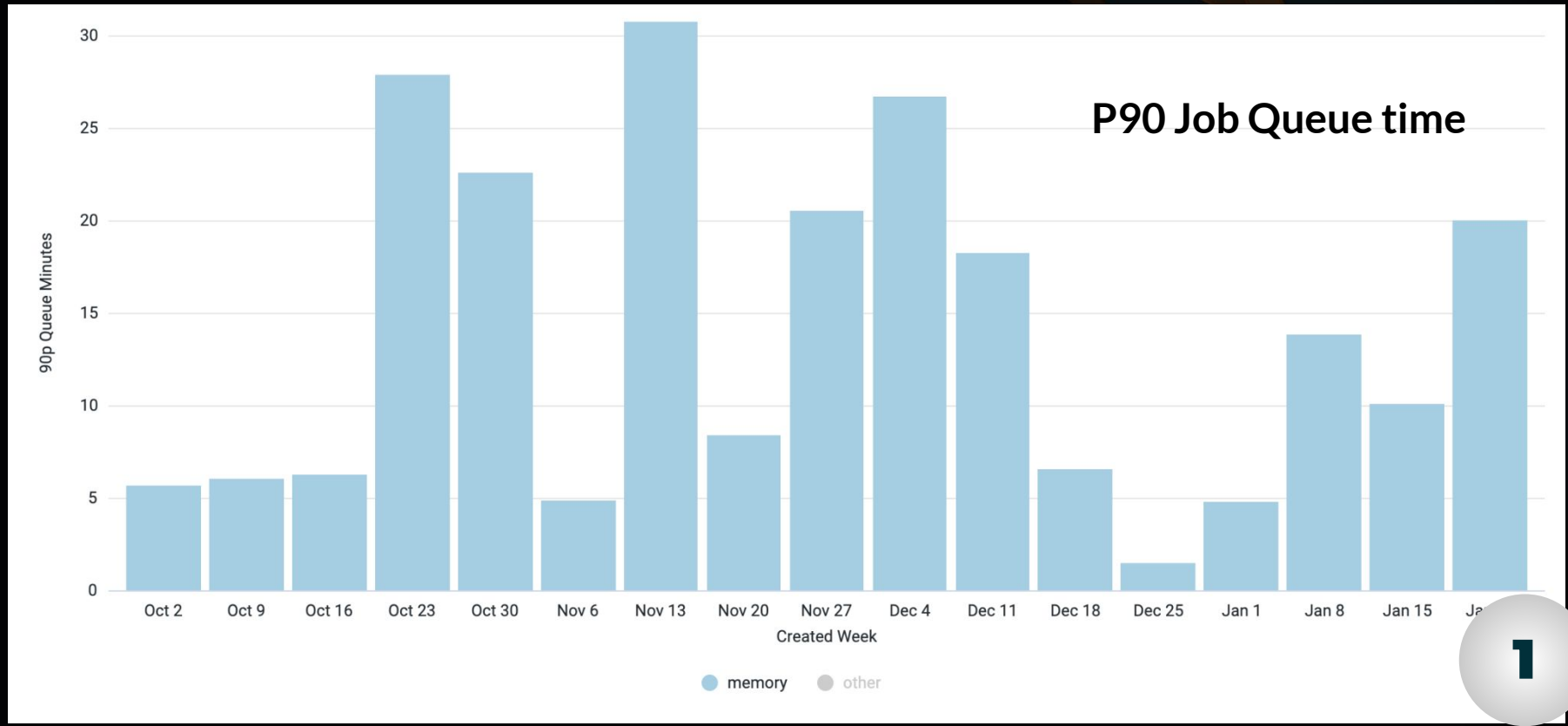


Reduce Queue Times

Build Waterfall



How bad is it?

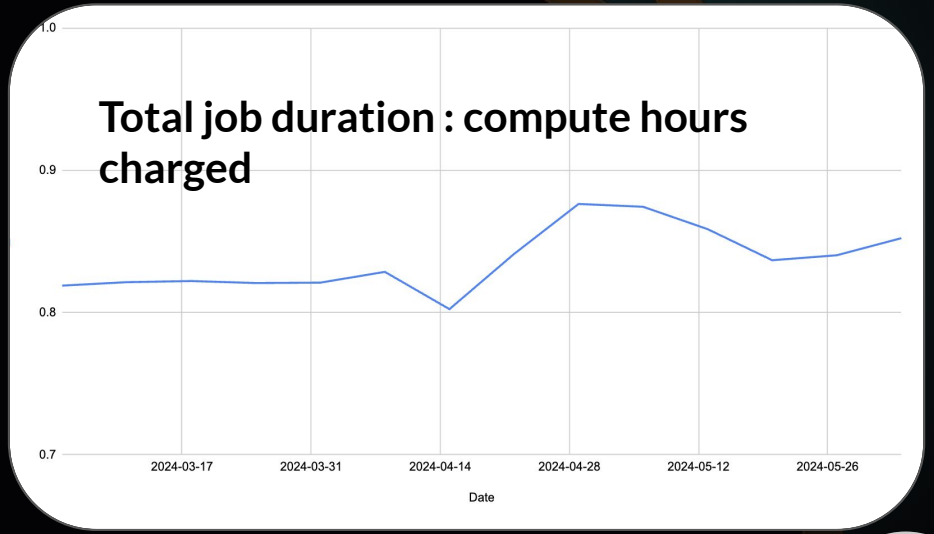

P90 Job Queue time
90p Queue Minutes
30
25
20
15
10
5
0
Oct 2
Oct 9
Oct 16
Oct 23
Oct 30
Nov 6
Nov 13
Nov 20
Nov 27
Dec 4
Dec 11
Dec 18
Dec 25
Jan 1
Jan 8
Jan 15
Created Week
memory
other

How much will it cost to improve queue times?

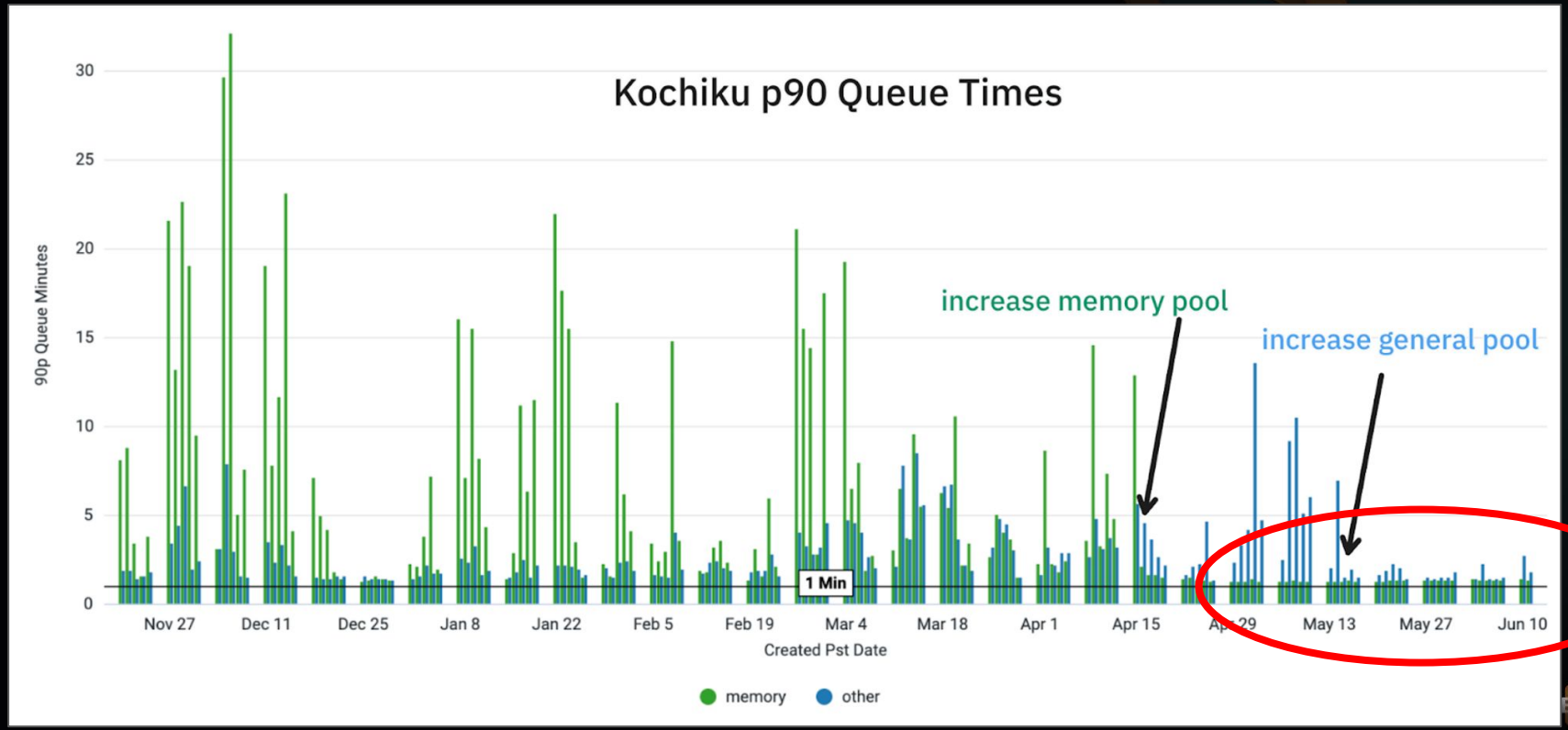
- Need to add more workers
- In our case:
  - Self-managed EC2 Instances
  - AWS Charges for vCPU hour
- But… in theory: our total job duration *should* be ~1:1 with compute hour charged.
- So… $0?

What to watch out for while increasing worker capacity

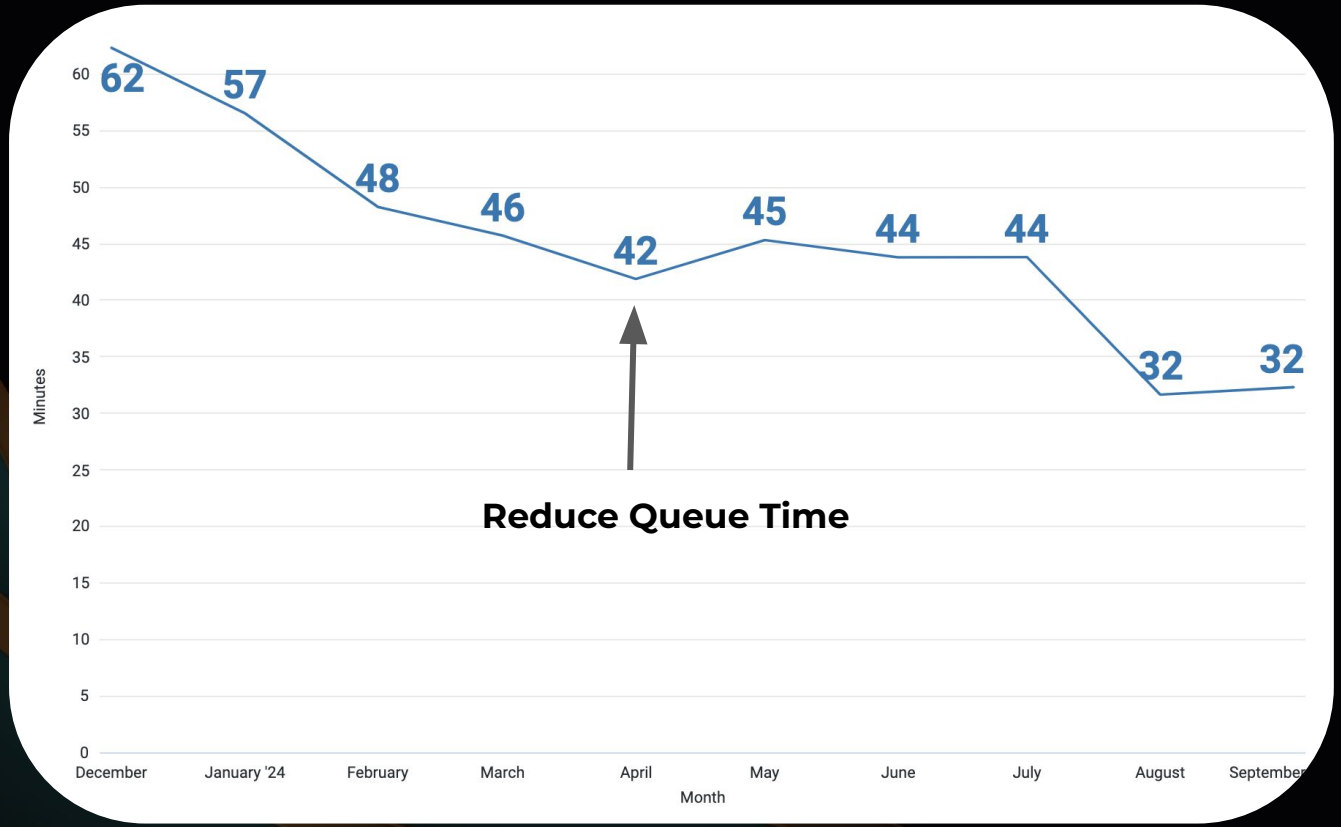
- Reliability of downstream services: Artifactory, Build Cache Infra, CI Secrets store, etc
- AWS service quotas, IP addresses available in subnet, etc
- Cost efficiency



P90 Queue Times

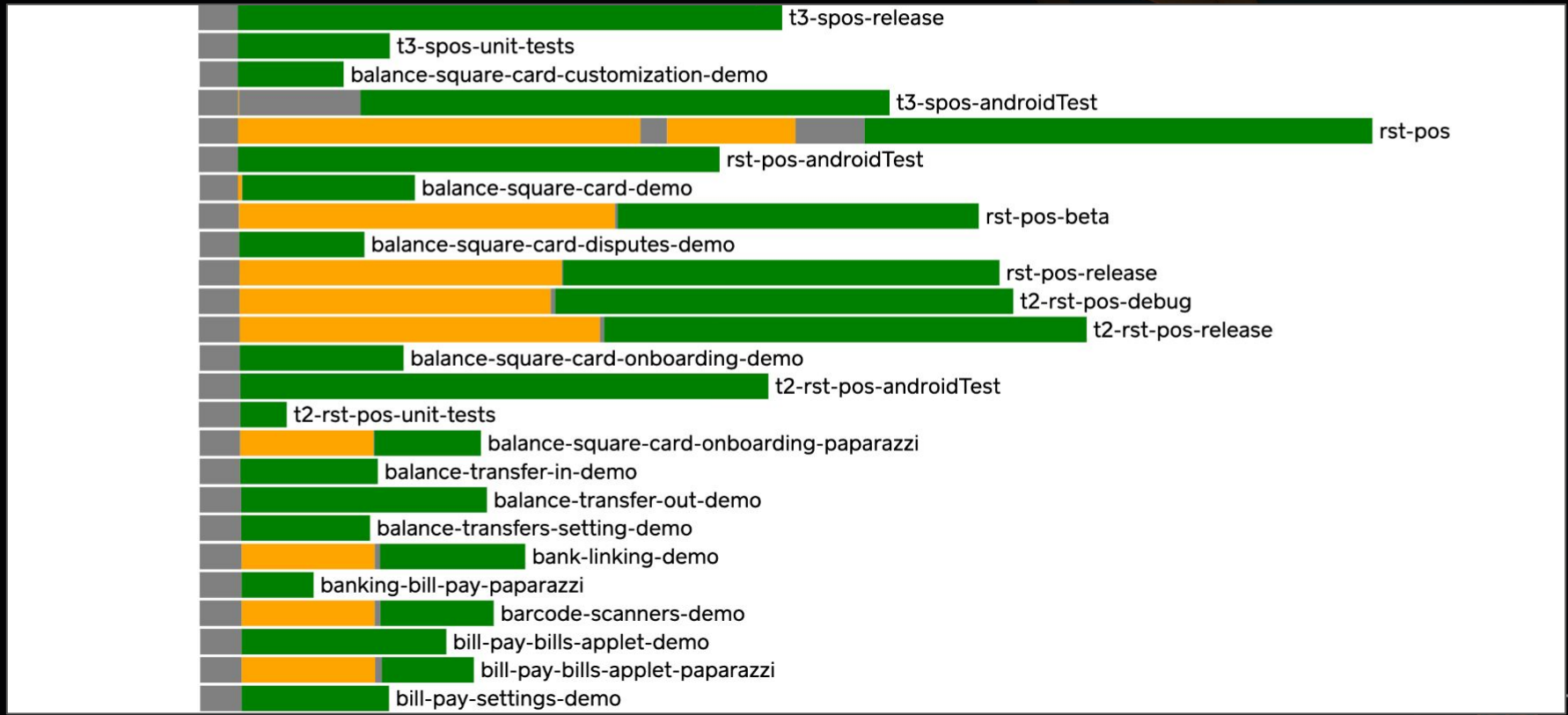


Reduce Queue Time =>
Improve PR build walltime by ~5 minutes

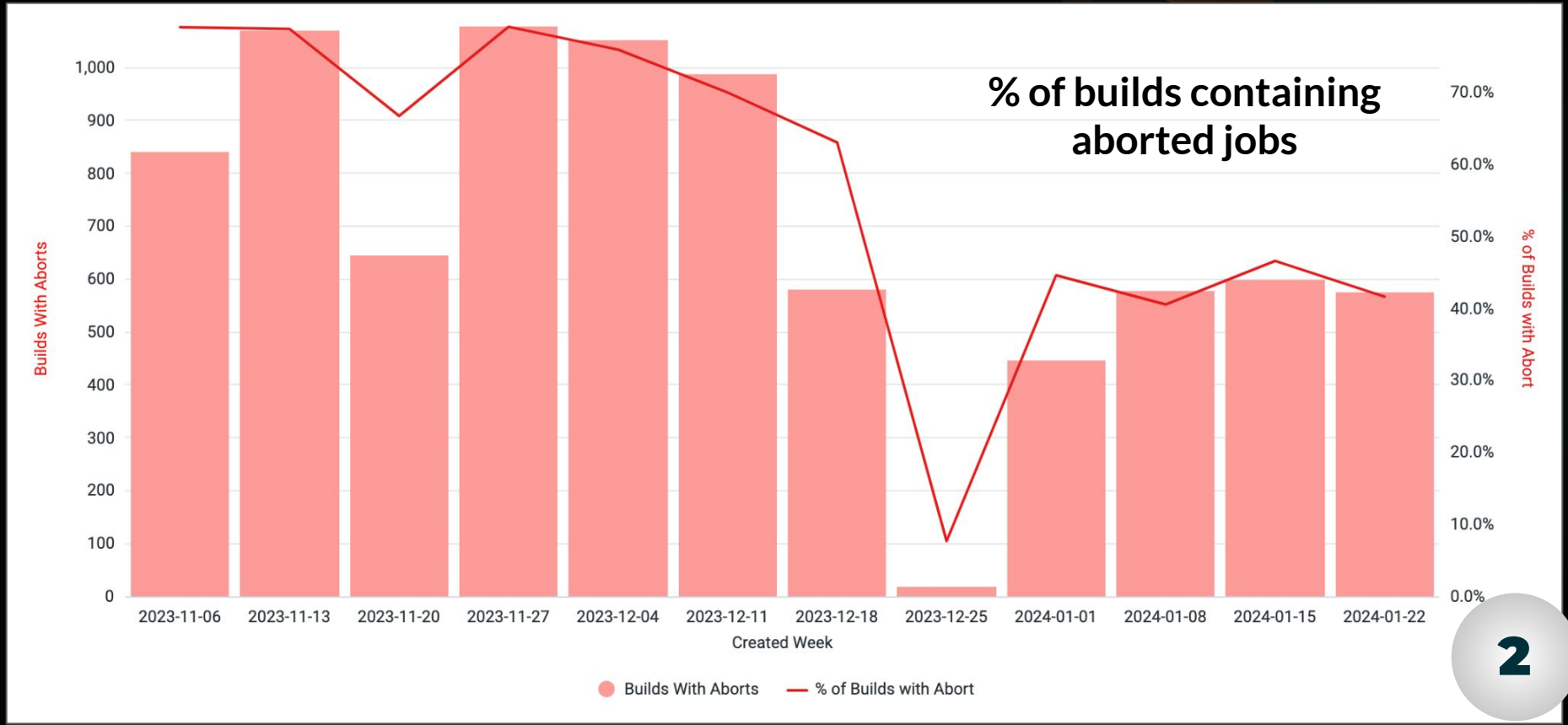


Eliminating Spot Interruptions

Build Waterfall



How bad is it? 😬


% of builds containing aborted jobs
Builds With Aborts
% of Builds with Abort
1,000
900
800
700
600
500
400
300
200
100
0
70.0%
60.0%
50.0%
40.0%
30.0%
20.0%
10.0%
0.0%
2023-11-06
2023-11-13
2023-11-20
2023-11-27
2023-12-04
2023-12-11
2023-12-18
2023-12-25
2024-01-01
2024-01-08
2024-01-15
2024-01-22
Created Week
Builds With Aborts
% of Builds with Abort

So what causes these aborts?

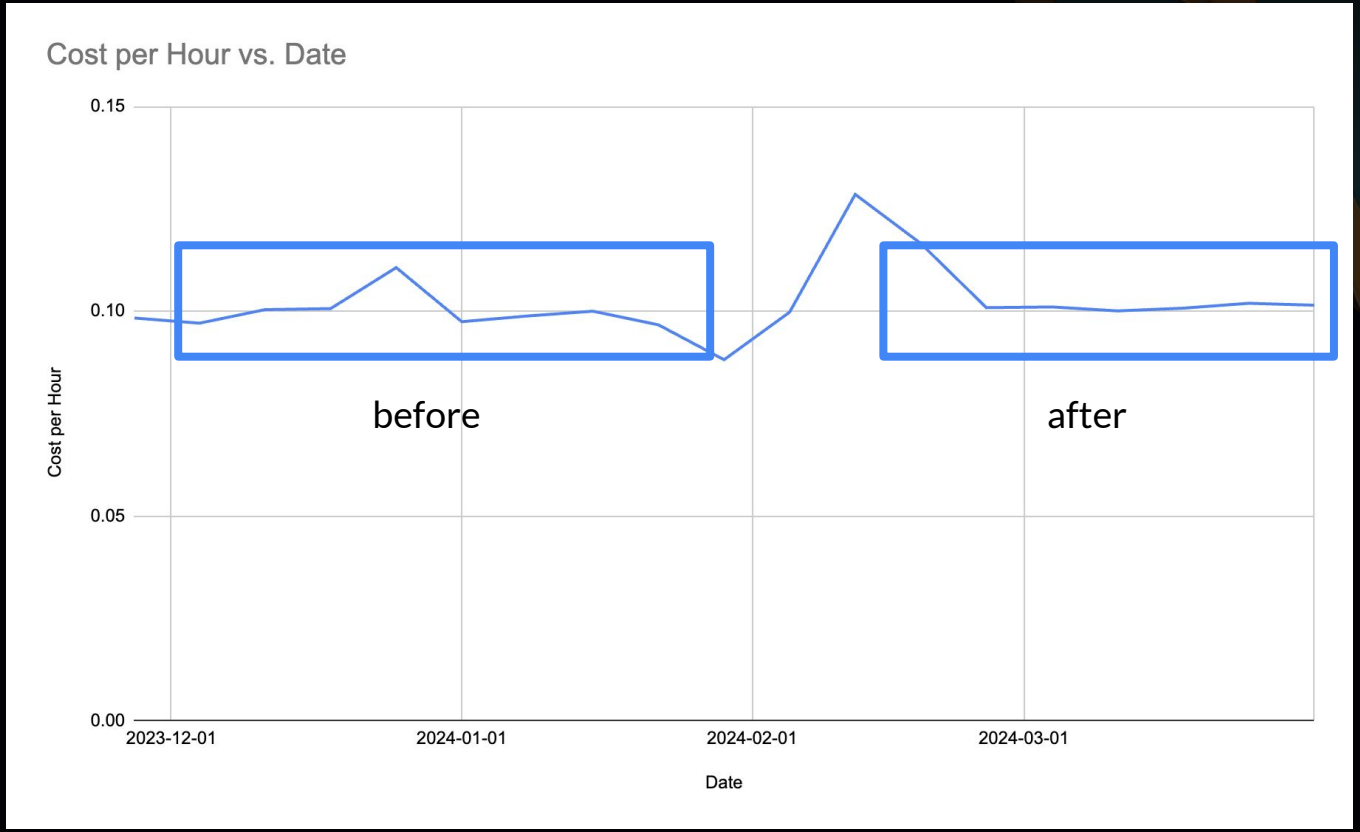
"You can launch Spot Instances on spare EC2 capacity for steep discounts in exchange for returning them when Amazon EC2 needs the capacity back. When Amazon EC2 reclaims a Spot Instance, we call this event a ***Spot Instance interruption****"*

So we need On-Demand Instances.
I guess we have to pay up $$
Of course, worth considering

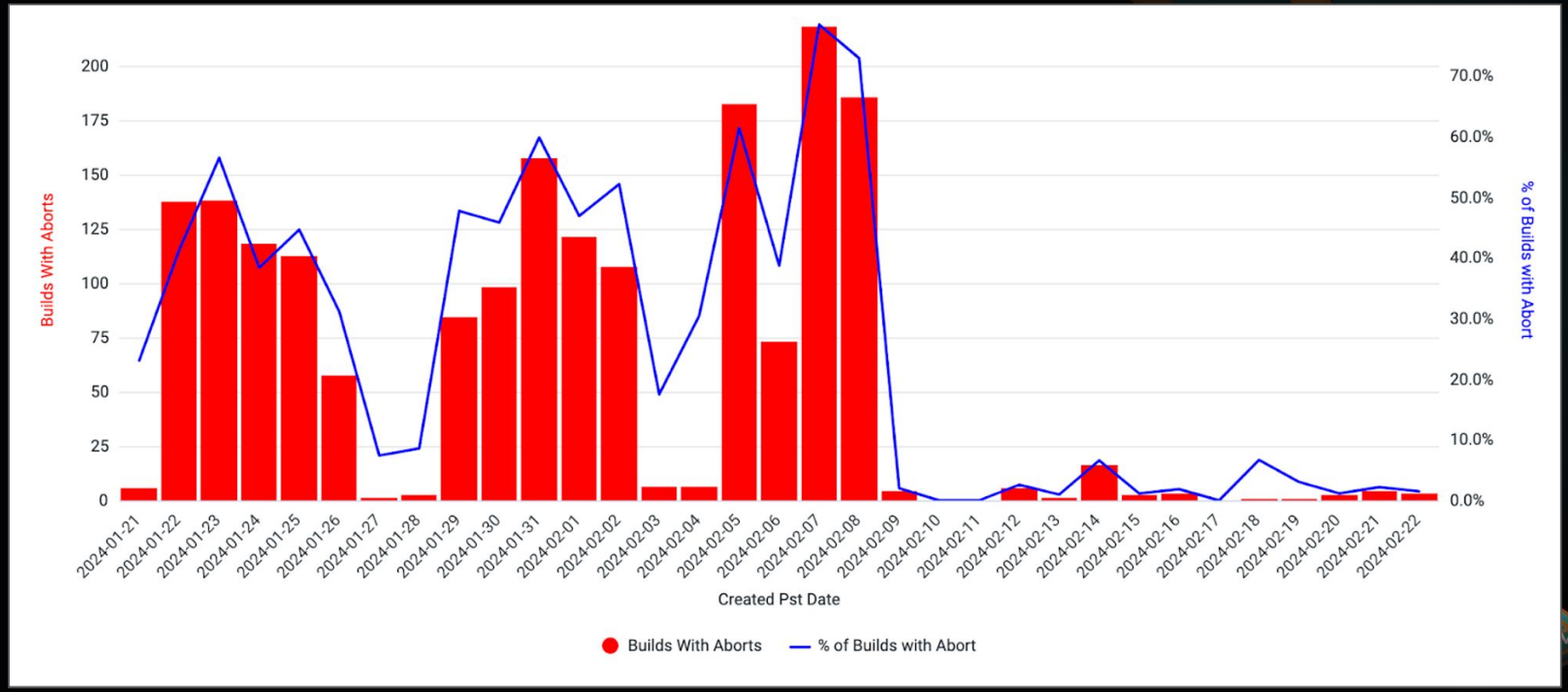
Let’s talk to our Cloud Economics team..

- They tell us about AWS Savings Plans
- It’s possible to get On Demand Instances highly discounted!
- Pick instance types that work well with Savings Plan and our work load
- Run an experiment!

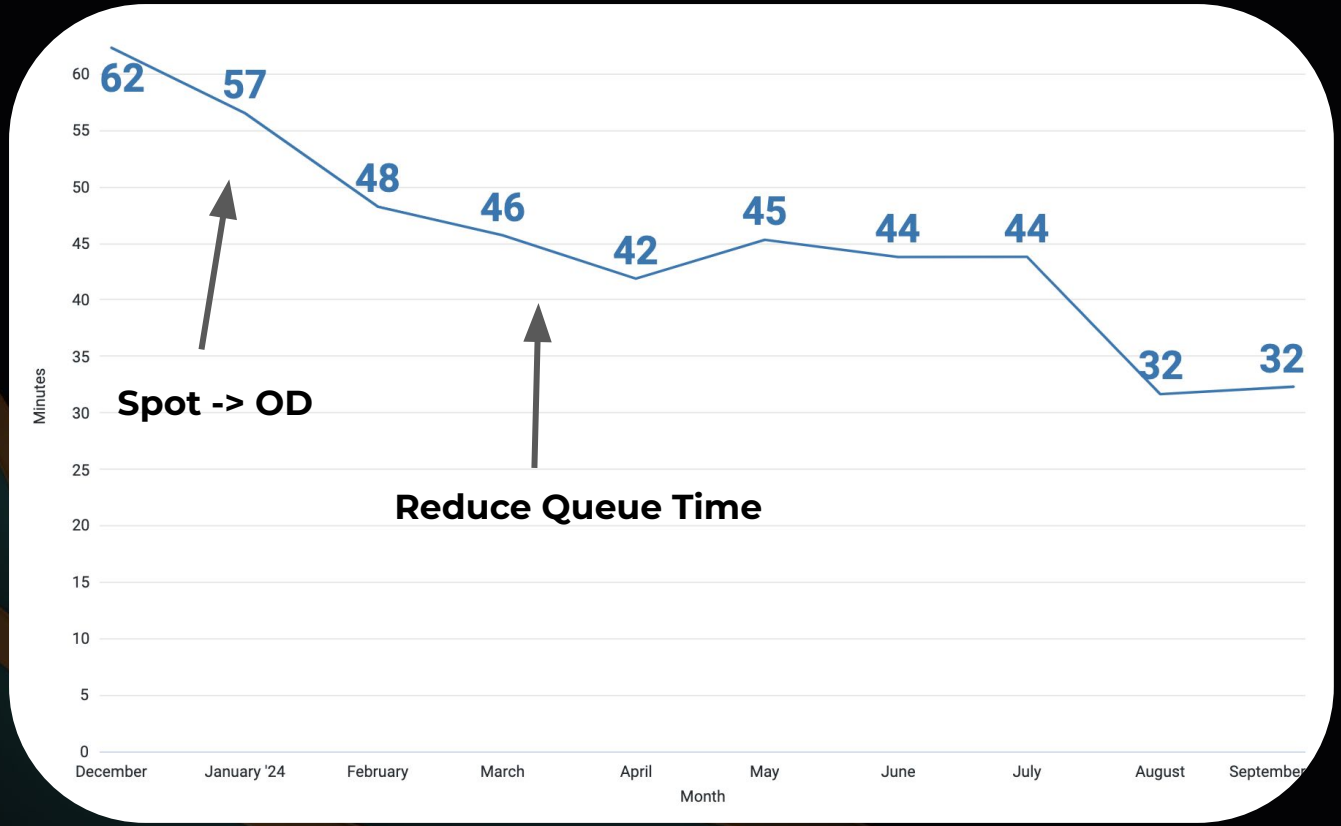
Spot -> On-Demand with SP
Cost per hour



No more Spot Interruptions!



Eliminate Spot Interruptions => Improve PR Build walltime by ~7.7 minutes



Bundled Gradle Cache

Bundled Gradle Cache

Prerequisite:

- using remote build cache with high hit rate
- ephemeral CI container

This is an optimization we make on top of that

Bundled Gradle Cache: Opportunity

Gradle Build Cache

- Each remote build cache fetch is a separate network request
- ~30,000 remote build cache requests per gradle invocation

Dependencies

- dependencies we fetch from our artifact store like Artifactory

Given that these are all fetched separately, you could see how this is accumulate for large builds

Bundled Gradle Cache: Idea

- Pre-populate the local gradle caches directory
- Stop fetching these individually
- This would eliminate the network overhead of fetching

The idea is to get closer to an “incremental” build on PRs

Bundled Gradle Cache: Implementation

Push from `main` CI builds:

- create tarball of the gradle caches dir:
  `tar -czf #{@bundled_cache} -C #{ENV['GRADLE_USER_HOME']} ./caches`
- upload to object store (S3)
  `s3://block-dx-bucket-us-w2/android-register/<git_sha>/<shard_name>`

Cost:

- Strictly worse performance on `main` CI builds
- Storage Cost

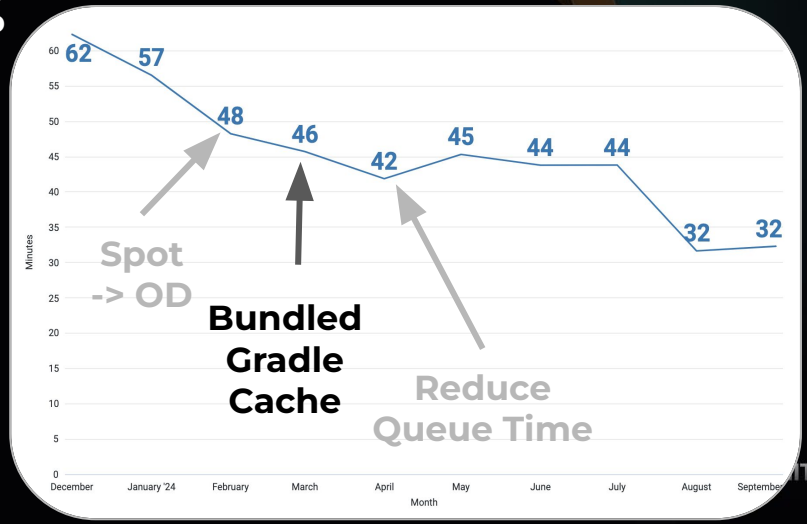
Bundled Gradle Cache: Implementation

Pull from PR builds

```
num_commits_to_check = 10
for each commit in git_revisions(merge_base_commit, num_commits_to_check):
  cache_path = commit + shard_name
  if cache_exists_in_s3(cache_path):
    log_cache_hit(commit)
    download_cache(cache_path)
    if local_gradle_cache_dir not exists:
      create local_gradle_cache_dir
    extract_cache_to(local_gradle_cache_dir)
    break
  else:
    log_cache_miss(commit)
```

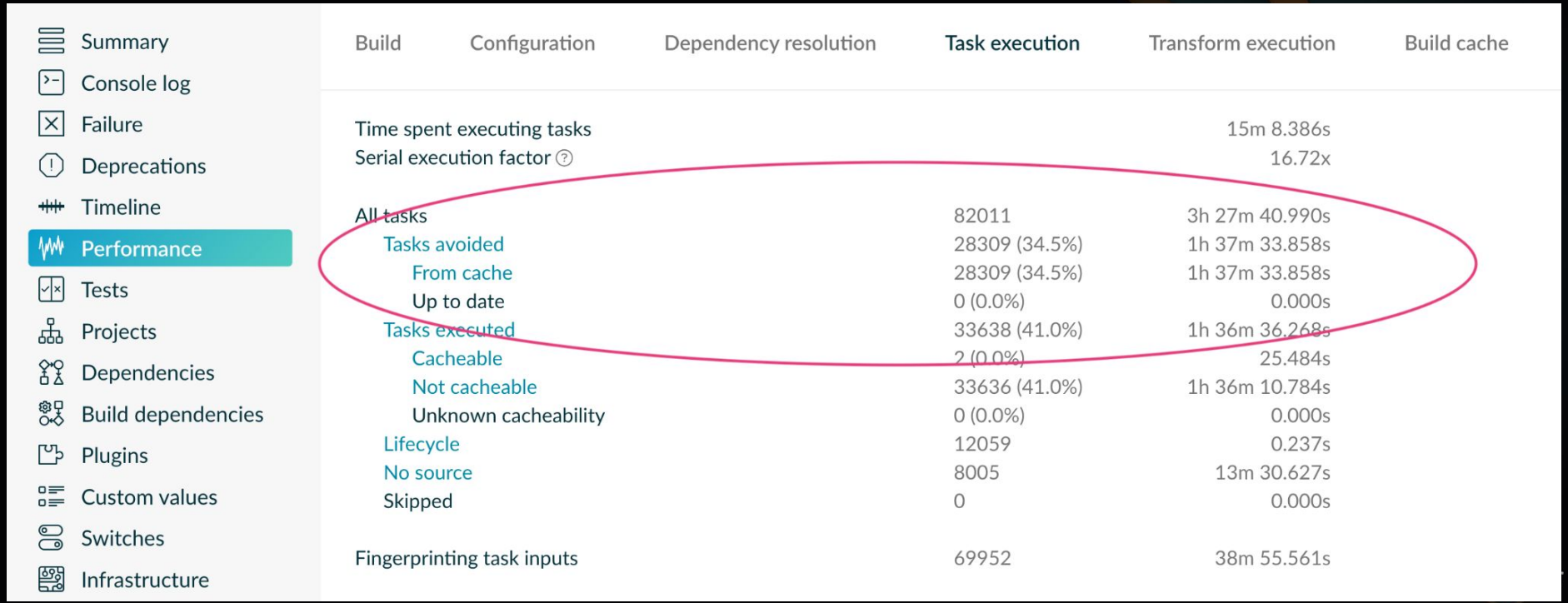
Bundled Gradle Cache: Outcome

- High local cache hit rates
- Reduced traffic on Artifactory
- Reduce remote cache requests by a factor of 10
  - 30,000 → 3,000
- gradle compilation shards improve by 17-28%
- Avg PR build walltime reduced by 3.8 min



Resolve Disk IOPS Bottleneck

What's going on here?



About EC2 Disk Storage

EBS: Elastic Block Store

- Network attached storage
- Default option
- 3000 IOPS free
- Pay to provision more

Attached Store:

- Directly Attached to Instance
- Millions of IOPS
- No additional cost
- Must be mounted

Instance Size	vCPU	Memory (GiB)	Instance Storage (GB)	Network Bandwidth (Gbps)	EBS Bandwidth (Gbps)
m6i.large	2	8	EBS-Only	Up to 12.5	Up to 10
m6i.xlarge	4	16	EBS-Only	Up to 12.5	Up to 10
m6i.2xlarge	8	32	EBS-Only	Up to 12.5	Up to 10
m6i.4xlarge	16	64	EBS-Only	Up to 12.5	Up to 10
m6i.8xlarge	32	128	EBS-Only	12.5	10
m6i.12xlarge	48	192	EBS-Only	18.75	15
m6i.16xlarge	64	256	EBS-Only	25	20
m6i.24xlarge	96	384	EBS-Only	37.5	30
m6i.32xlarge	128	512	EBS-Only	50	40
m6i.metal	128	512	EBS-Only	50	40
m6id.large	2	8	1x118 NVMe SSD	Up to 12.5	Up to 10
m6id.xlarge	4	16	1x237 NVMe SSD	Up to 12.5	Up to 10
m6id.2xlarge	8	32	1x474 NVMe SSD	Up to 12.5	Up to 10
m6id.4xlarge	16	64	1x950 NVMe SSD	Up to 12.5	Up to 10
m6id.8xlarge	32	128	1x1900 NVMe SSD	12.5	10
m6id.12xlarge	48	192	2x1425 NVMe SSD	18.75	15

Our situation

Using default IOPS on EBS (3000)

Provisioned instance types with disk

Not yet mounted

Instance Size	vCPU	Memory (GiB)	Instance Storage (GB)	Network Bandwidth (Gbps)	EBS Bandwidth (Gbps)
m6i.large	2	8	EBS-Only	Up to 12.5	Up to 10
m6i.xlarge	4	16	EBS-Only	Up to 12.5	Up to 10
m6i.2xlarge	8	32	EBS-Only	Up to 12.5	Up to 10
m6i.4xlarge	16	64	EBS-Only	Up to 12.5	Up to 10
m6i.8xlarge	32	128	EBS-Only	12.5	10
m6i.12xlarge	48	192	EBS-Only	18.75	15
m6i.16xlarge	64	256	EBS-Only	25	20
m6i.24xlarge	96	384	EBS-Only	37.5	30
m6i.32xlarge	128	512	EBS-Only	50	40
m6i.metal	128	512	EBS-Only	50	40
m6id.large	2	8	1x118 NVMe SSD	Up to 12.5	Up to 10
m6id.xlarge	4	16	1x237 NVMe SSD	Up to 12.5	Up to 10
m6id.2xlarge	8	32	1x474 NVMe SSD	Up to 12.5	Up to 10
m6id.4xlarge	16	64	1x950 NVMe SSD	Up to 12.5	Up to 10
m6id.8xlarge	32	128	1x1900 NVMe SSD	12.5	10
m6id.12xlarge	48	192	2x1425 NVMe SSD	18.75	15

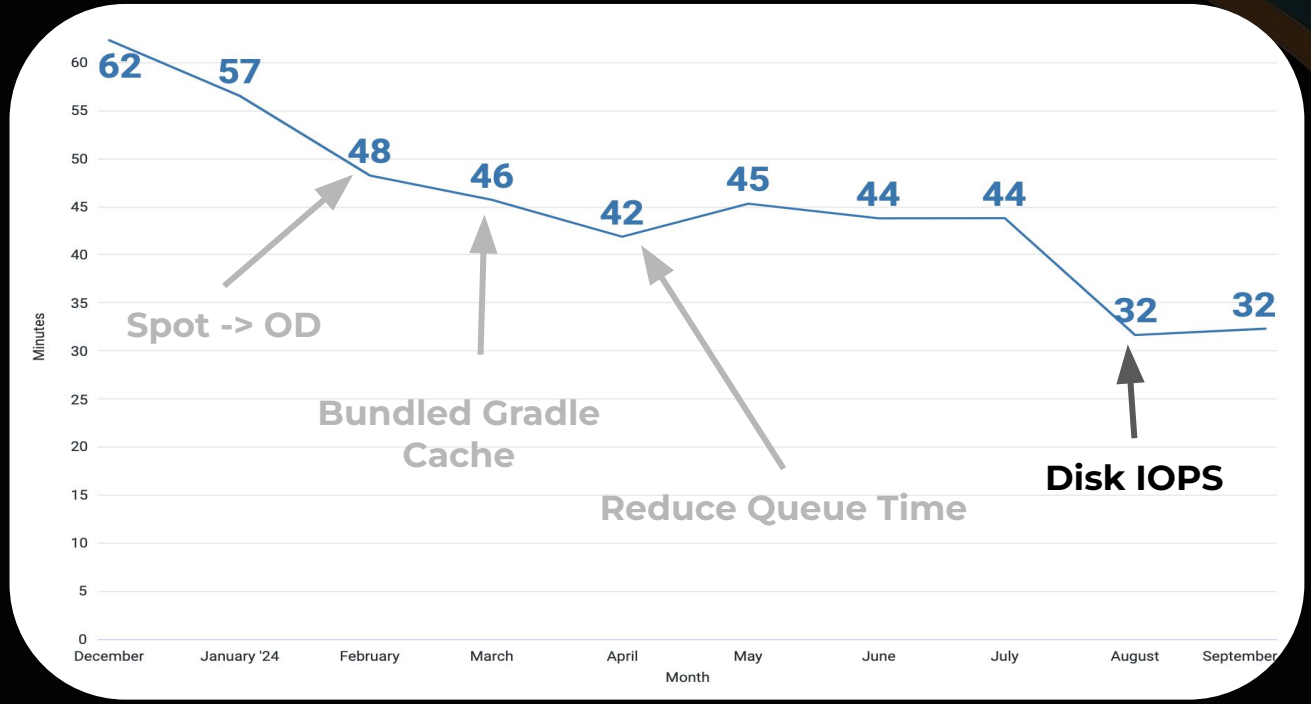
Mount the Attached SSD!

Tasks avoided	28309 (34.5%)	1h 37m 33.858s
From cache	28309 (34.5%)	1h 37m 33.858s
Up to date	0 (0.0%)	0.000s
Tasks executed	33638 (41.0%)	1h 36m 36.268s

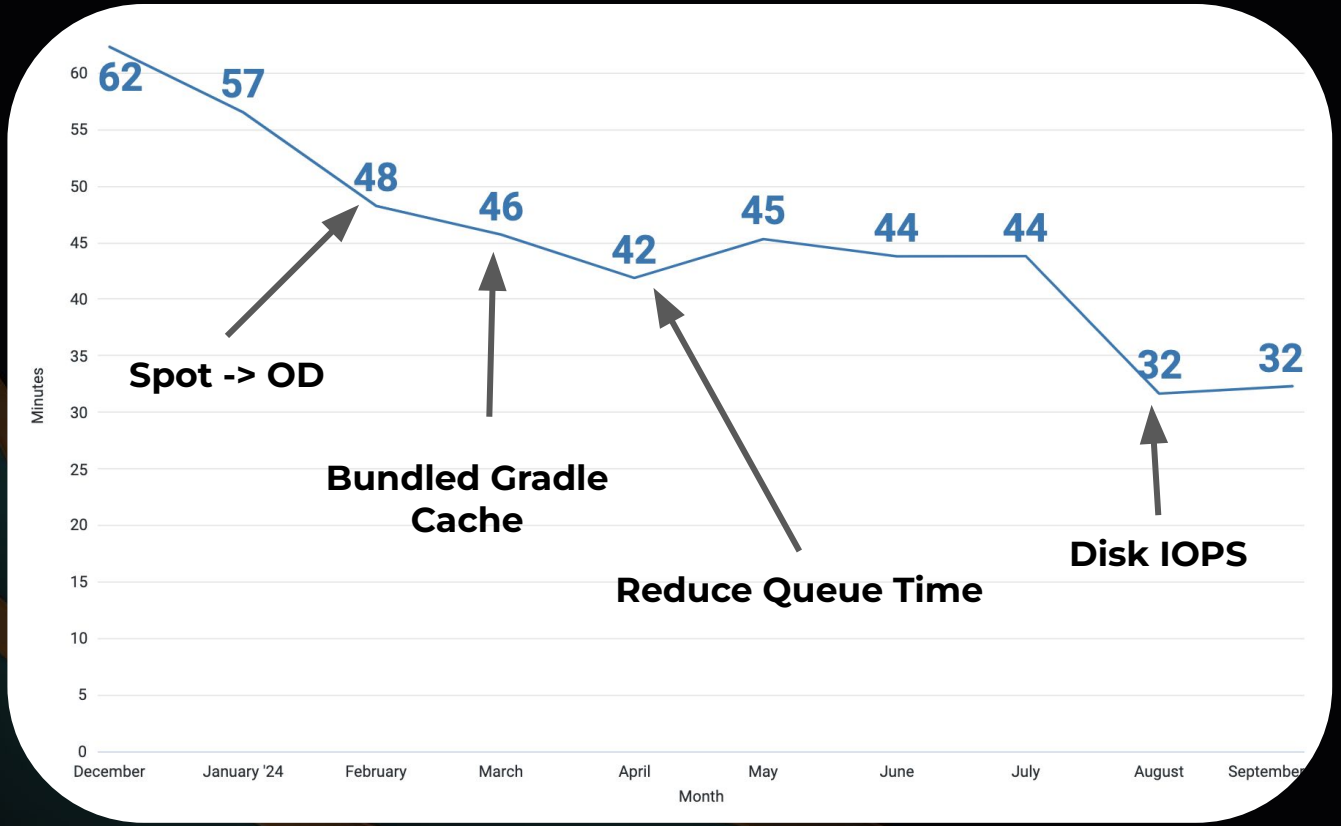
Tasks avoided	28309 (34.5%)	17m 19.389s
From cache	28309 (34.5%)	17m 19.389s
Up to date	0 (0.0%)	0.000s
Tasks executed	33638 (41.0%)	12m 23.870s

After resolving IOPS bottleneck

- Duration of compile jobs cut in half
- ~12min reduction in PR build Walltime



Recap: PR Build Walltime YTD



Other Takeaways

- Focus on Walltime for Dev productivity
  - Even though we didn't optimize for cost efficiency, we improved it.

- Visualize your build for clues and insights

- Be willing to work outside of your platform & "domain". Work with your Cloud Econ and CII teams

THANK YOU

Michael Yoon
myoon@block.xyz