Microsoft

Deep Dive:

Azure Resource Manager Outage

Brendan Burns
Benjamin Pannell

Summary of Impact

Azure Resource Manager (ARM) is the central Control Plane API gateway for Azure.

It is an integral part of our customer experience and provides critical functionality including governance, access control, caching, and a single global management endpoint.

35 hrs	3%
Duration of impact	Requests impacted globally

Nature of Impact

A subset of Microsoft Azure customers experienced timeouts and failures for long-running management operations.

This had knock-on impact for other internal services which rely on Azure Resource Manager to provide functionality.

Scope of Impact

Primary impact in West US, West US 2, South Central US, North Europe, West Europe, East Asia, and Southeast Asia regions. Marginal impact in other regions globally.

Who are we?

Brendan Burns

CVP, Azure OSS and Cloud Native

Co-founder of the Kubernetes open-source project, and CVP in charge of Microsoft Azure APIs, governance and management, as well as the Azure Kubernetes Service and cloud-native open source. I've built and run high-scale, mission-critical distributed systems for more than a decade.

Benjamin Pannell

Tech Lead, Azure Control Plane SRE

I've spent the last 4 years working as an SRE at Microsoft, supporting critical control plane services and helping to grow (and being supported by) an exceptional team of engineers. Previously, I've worked as an SRE on a global gaming platform, and as the lead software engineer on an agricultural vehicle tracking product.

What to expect

This is the full incident review used by internal teams to learn from this outage.

We're going to share information with you in the order it became visible to us.

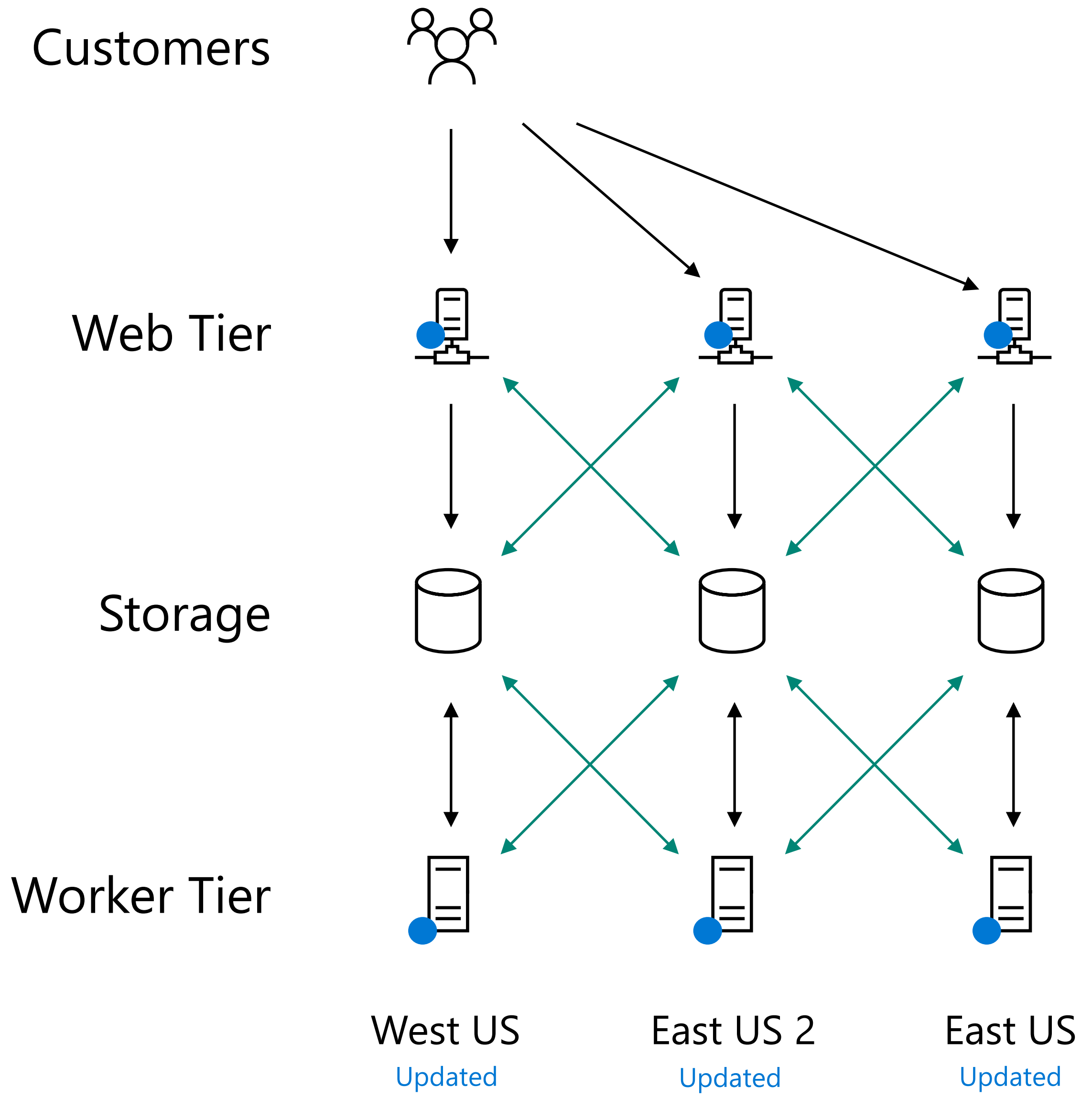
We'll include relevant context about the system in blocks like this.

We'll call out factors which exacerbated the incident in blocks like this.

1

Before the Outage

What you need to know about the system and the run-up to this outage.



System Architecture

Azure Resource Manager is effectively a “Web-Queue-Worker” service.

We operate with fully redundant infrastructure in every Azure region.

And we use a fully-connected topology to provide resiliency to regional failures of any component.

We conduct deployments in phased batches over an 8-day period.

Updating a legacy component

We introduced a change to improve support for automated new-region buildout, targeting a legacy job type.

The change was introduced with feature flags which relied on a new configuration system used elsewhere in the codebase.

...and the full end-to-end test suite passed, including the new tests to cover this change.

A transitive test dependency included files required for this code to function, however these were not present in the final deployment artifacts.

Change Introduced
10th December

December Holiday Period

Rollout Started
6th January - Canary

7th - Low Traffic

10th - Medium Traffic

11th – High Traffic

12th – Rest of World 1

Exposure

The rollout timeline

Azure teams pause feature rollouts to help reduce outages for both our engineers and customers during the December holiday period.

This delay meant that the engineers responsible for the change had lost context on the risks by the time it rolled out.

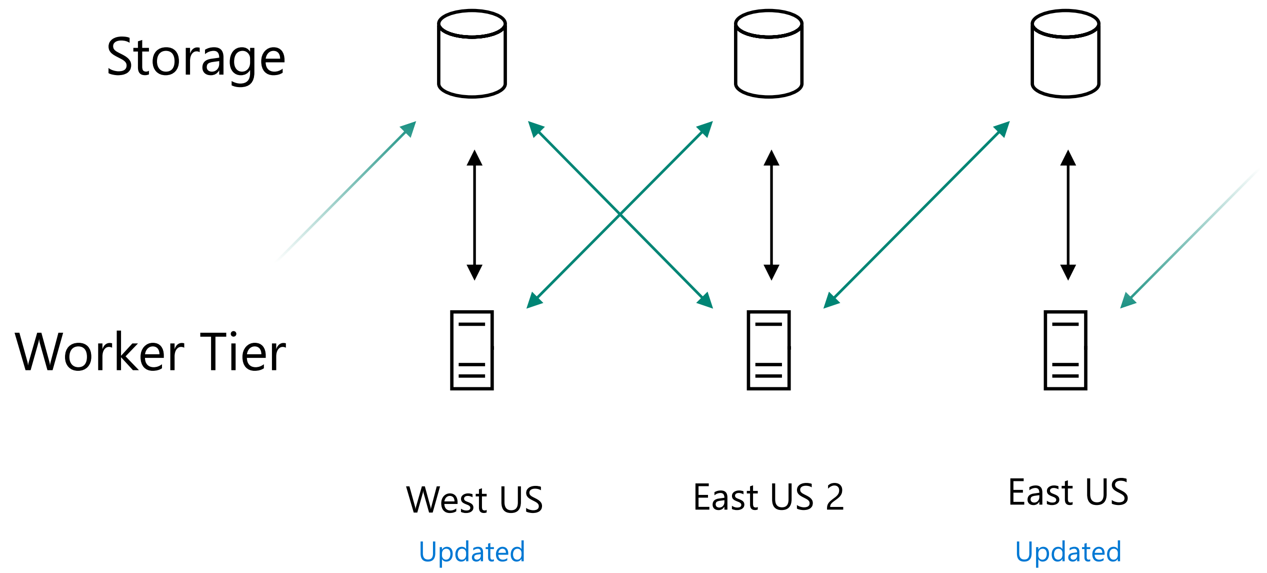
The deployment process continued without any observable issues for the next week.

2

First signs of trouble

How we detected, and then responded, when things first started deviating from our expected baseline.

Worker redundancy

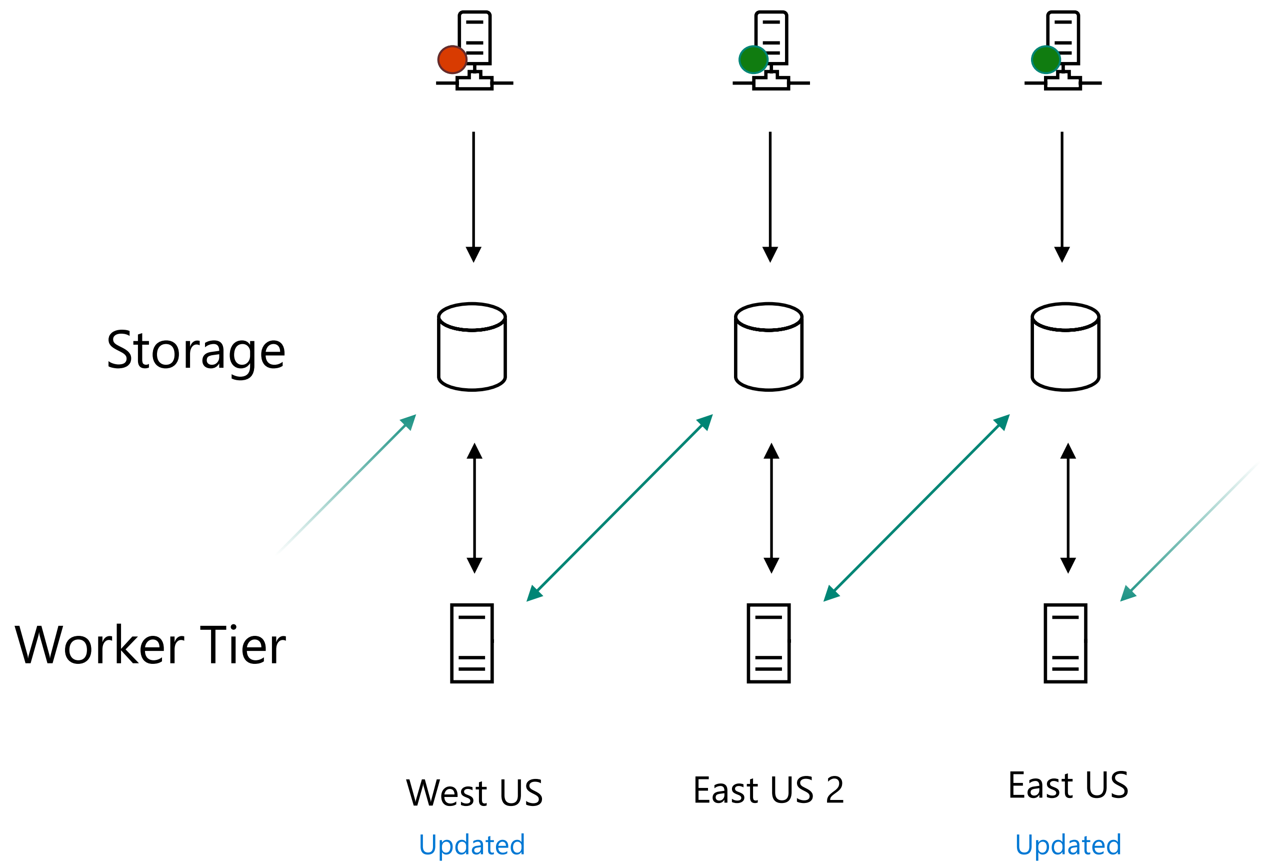


Azure services often pair regions to provide resiliency in the face of regional failures. ARM natively supports this pairing at the worker layer, employing job-stealing for active-active redundancy.

Business needs have necessitated groupings of three or four regions in some cases.

West US and East US are the first set of large paired regions to be updated in our release sequence.

Initial detection

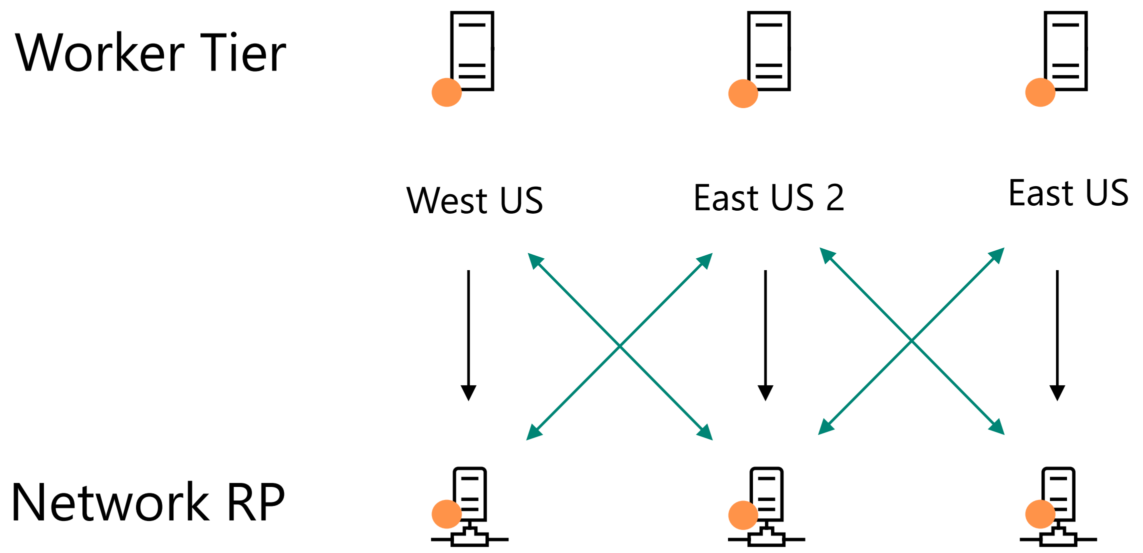


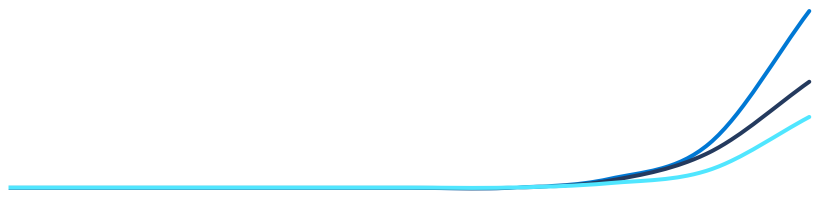
Azure services use synthetic monitoring to continuously test common user flows in production environments and report on unexpected failures.

We responded by tracing the execution of the failed requests through our system.

We use an internal version of Azure Monitor Logs for this and maintain 3 months of logs (~40PB) with the ability to execute complex queries against them in seconds.

Downstream Failures





HTTP 404 Error Rate (by region)

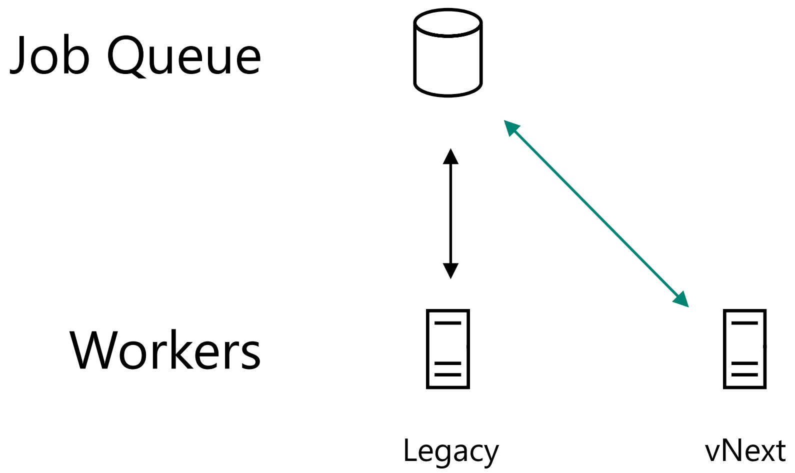
ARM acts as an API-gateway for many backend services called “Resource Providers” (RPs) which expose different types of Azure resources to customers.

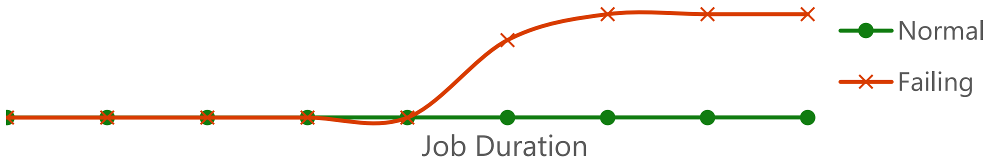
Significant increase in HTTP 404s from a downstream service responsible for Network resources.

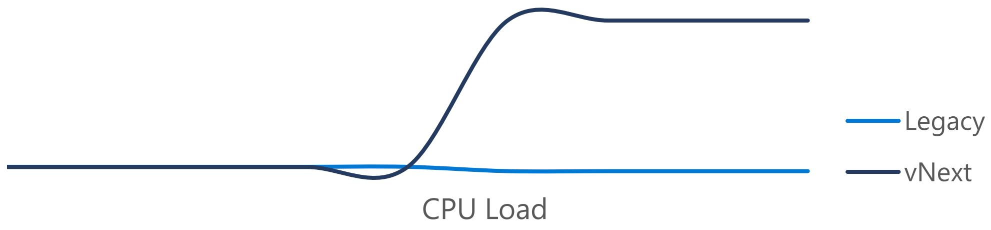
These caused failures for long-running customer operations.

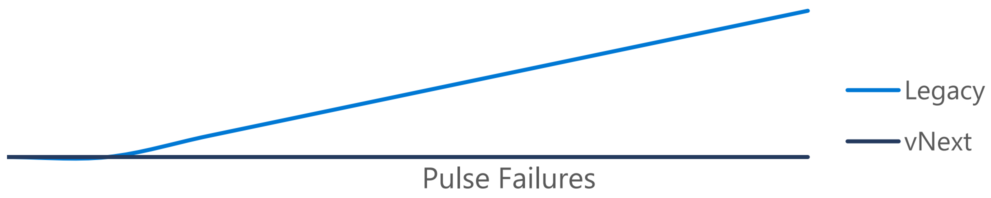
These were the result of cache eviction for status metadata and a symptom of the true problem, rather than the cause.

Looking for Oddities









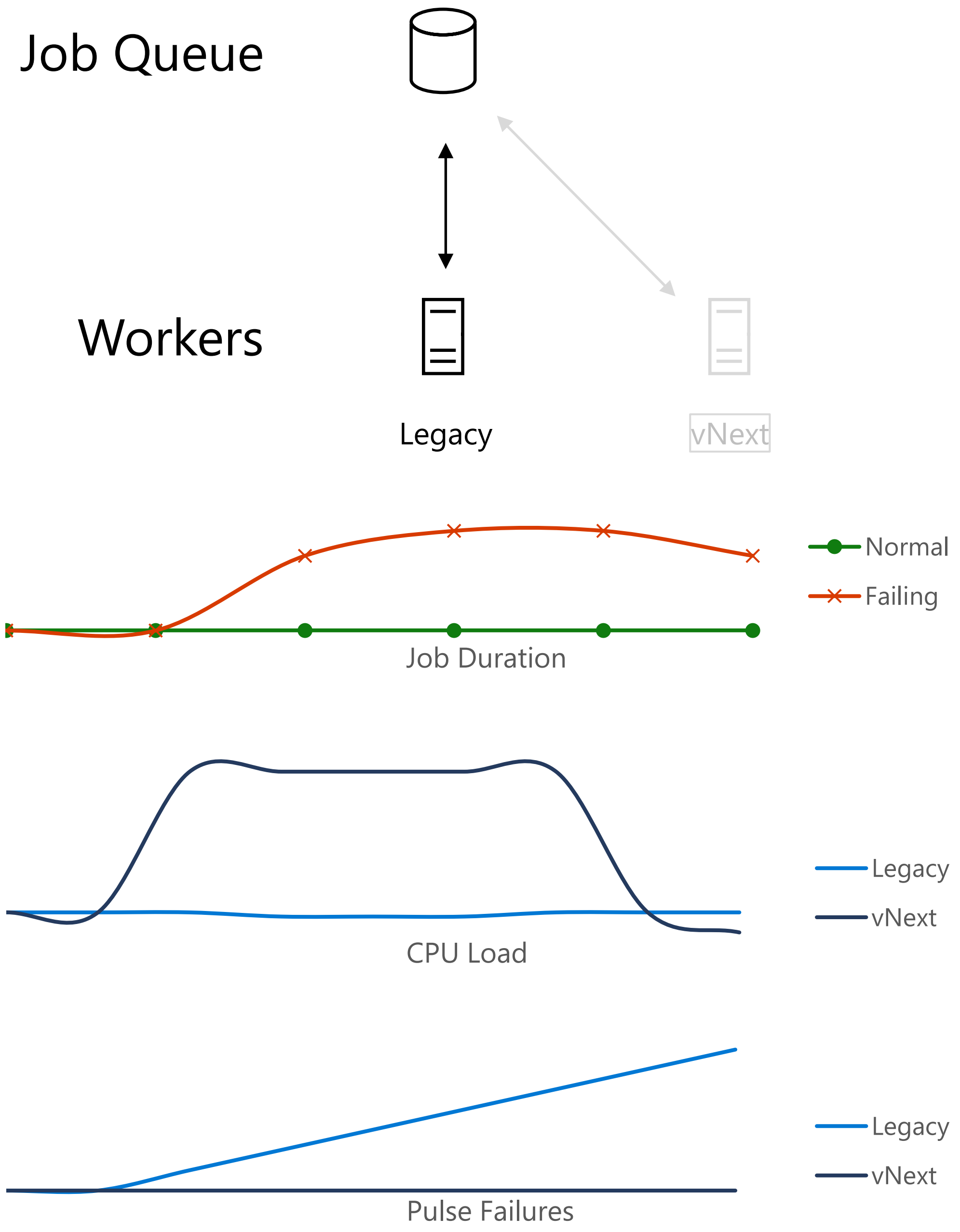
vNext is an ongoing effort to modernize our hosting platform to improve long-term stability and supportability.

We observed an increase in job duration in impacted regions, matching the symptoms reported by customers.

This correlated with a jump to ~100% CPU on our vNext workers, and an increase in “pulse” failures.

“Pulsing” involves re-schedule jobs for immediate execution, bypassing the queue, to improve performance. It will only occur if the worker has available capacity (CPU and work slots).

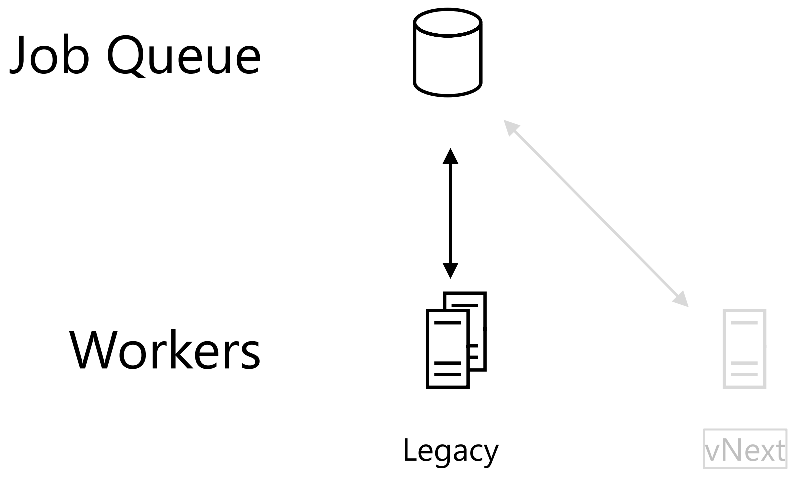
Blame the New Thing

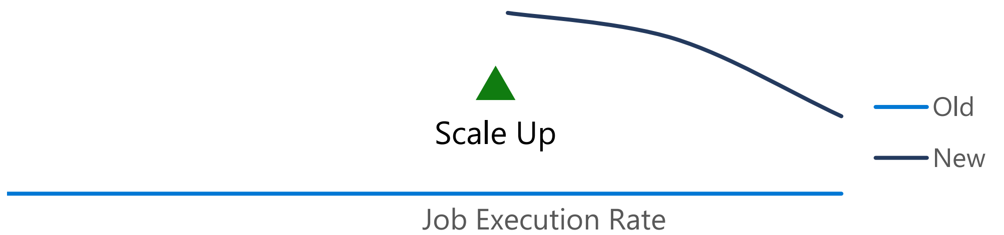


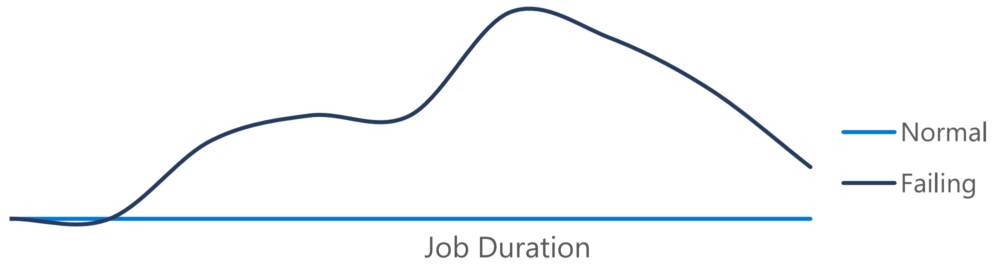
We decided that the likely cause was the vNext workers, which had experienced problems previously, and decided to remove these.

We observed the expected drop in CPU and job duration, which matched our expectations.

Removing the vNext workers removed the only workers that were truly healthy, exacerbating the true impact.







Simultaneous Change

To offset the loss of capacity caused by removing our vNext workers, we decided to scale up our Legacy workers to compensate.

These workers immediately started processing a significant number of jobs and we observed long running operations completing successfully again.

Making two changes simultaneously made it harder for engineers to determine which had the positive impact.

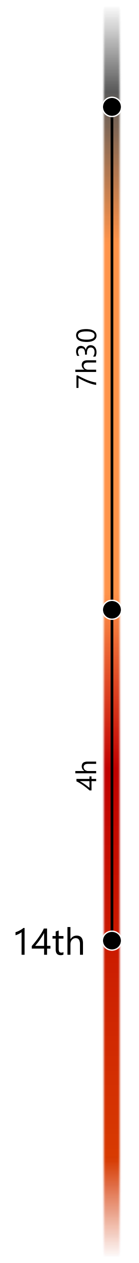
With job duration falling and operations succeeding, we determined that the issue was mitigated.

3

Expanding impact

Responding to a situation which continued to deteriorate.

The timeline so-far



Initial Detection
11:51 UTC

16:00 UTC – Shift Handover

16:02 UTC – Rest of World 2 rollout starts

Initial Mitigation Steps
19:18 UTC

00:00 UTC – Telemetry Appears Healthy

Initial mitigation steps were taken ~7h30 after initial detection.

Over the next 4 hours, engineers monitored increasingly “healthy” looking telemetry, declaring mitigation at ~00:00 UTC.

In the background, our automated deployment system continued to rollout the latest release to the second half of the world, since all previous regions were reporting healthy telemetry.

14th

01:10 UTC – Reports of issues in other regions

01:59 UTC – Rest of World 2 rollout completes

All vNext Workers Removed
02:10 UTC

03:00 UTC – Correlation between failures and new code identified

Ctrl+C; Ctrl+V

An hour later, we started to receive reports of issues in other regions.

We immediately applied the same mitigation globally – removing vNext workers.

These workers were not required to meet our capacity needs, and the previous improvement supported their removal.

Shortly thereafter, engineers started to spot a correlation between operation failure rates and the latest code release.

This correlation only manifested after the second half of the world was updated.

14th

03:00 UTC – Correlation between failures and new code identified

Rollback Started
03:34 UTC

06:00 UTC – Targeted rollback completed
Most customer impact mitigated

Rollback Completed
18:00 UTC – All customer impact mitigated

Rolling Back

We decided to rollback to the previous release.

When performing rollbacks, we follow a phased rollout process to avoid making a bad situation worse.

To minimize customer impact, we opted to rollback the most heavily impacted regions in the first batch of rollouts.

Our engineers believed that given the extended time the previous release had been running, it was highly unlikely to worsen the situation in these badly impacted regions.

Investigation

Deciphering a complex failure mode through delegation of responsibility.

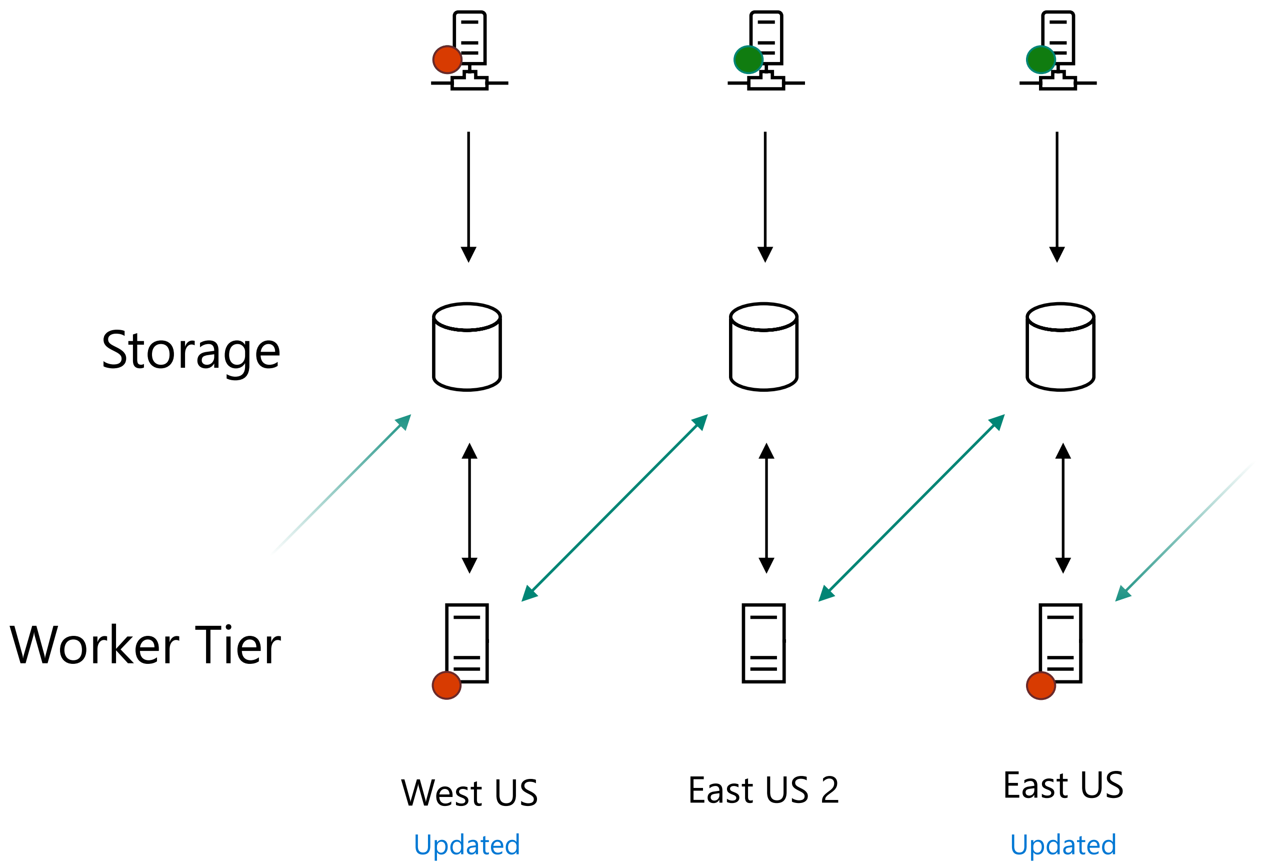
4

Creating Space

Mitigating an incident and investigating the cause are distinct responsibilities.

We delegated responsibility for the investigation to Azure Control Plane SRE, while the on-call engineers focused on mitigation.

This allowed our investigators to focus on understanding the system's state and developing hypotheses.

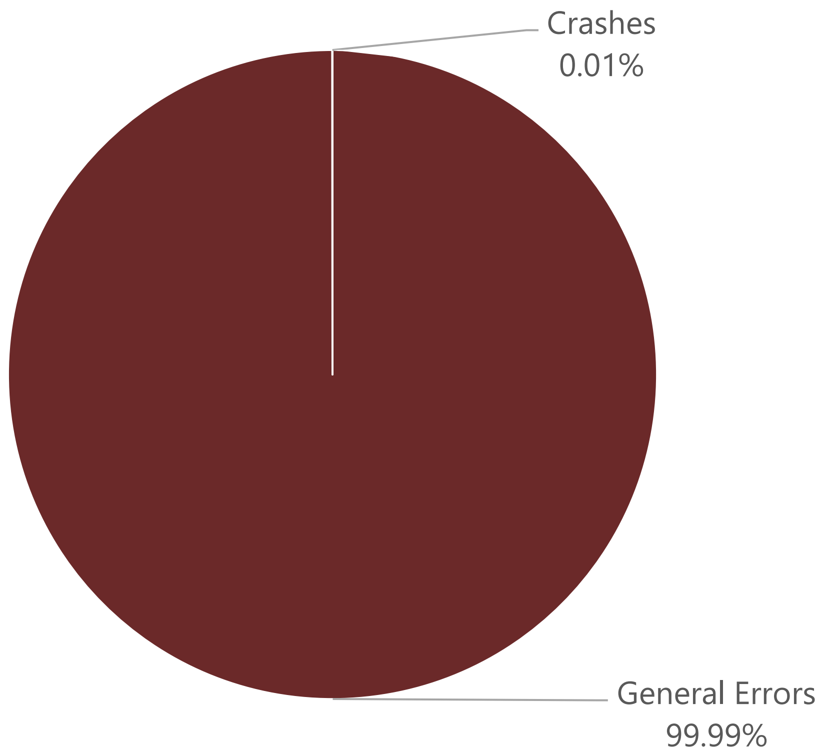


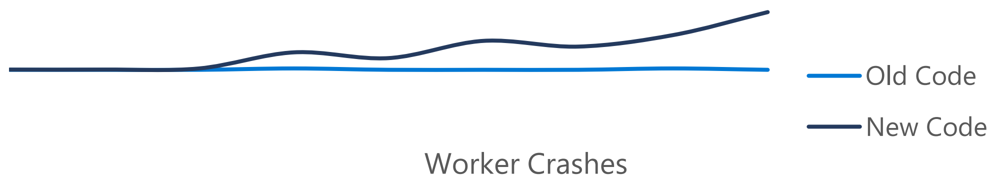
Why West US first?

Our rollout batches cause West US and East US to be updated in the first half of the Rest of World rollout.

If the workers in these regions weren't processing jobs, we would see impact only in West US (because East US 2 would provide redundancy for East US).

We operate paired regions in our test and canary environments, however traffic in these environments was low enough that it didn't trigger the bug.



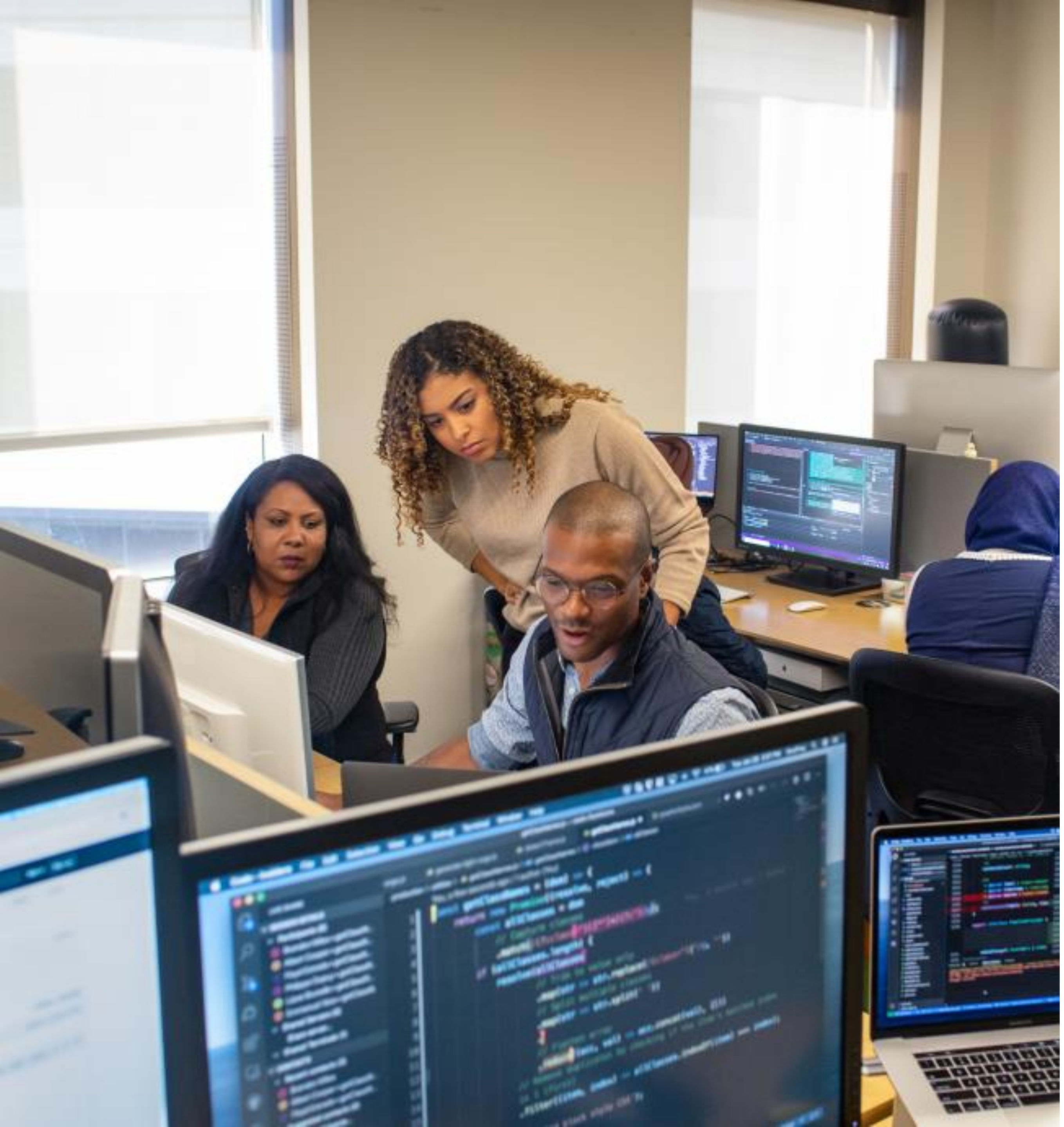


Where are the errors?

Our engineers had spent 24hrs+ looking for errors which correlated with the failures we were seeing and had found nothing significant.

We performed statistical analysis against our logs and identified an extremely low-volume event which had seen a 100x increase, recording crashes in our workers.

These crashes accounted for ~1% of the fleet, far below the levels required to impact our capacity and only appeared hours after the first recorded impact.



Why the delay?

The error causing the crash happened hours after a worker started.

It was thrown within the context of a job, which usually catches exceptions and marks the job as failed.

Our engineers were dubious that this was the cause, but worker crashes matched the observed impact perfectly (we'd just need far more of them than we were seeing).

The Failure Mode

The Code Change

```
TypeInitializationException
 └→ DirectoryNotFoundException
```

The use of a static initializer caused the exception to be thrown when referencing the job type at runtime.

The use of Reflection to initialize the job type caused the underlying exception to be wrapped.

The exception was thrown in the initialization logic, before the feature flags were evaluated. This resulted in customer impact even though the intended change was disabled.

LegacyJob.cs

```
public class LegacyJob {
    // New config system
    static IConfigProvider configProvider
        = new ConfigProvider("config");

    public DoWork() {
        if (FeatureFlag.IsEnabled) {
            // Use the configProvider
        } else {
            // Use the legacy approach
        }
    }
}
```

The Job Worker

TypeInitializationExceptions are fatal errors which should immediately crash the process.

The orchestrator will restart the process and return it to a healthy state.

Classifying the exception as a fatal failure prevented any logging, hiding its occurrence from operators. It also caused the job slot to be terminated, instead of handling this as a general issue with the job.

JobWorker.cs

```
public async Task RunJobsAsync() {
  while (true) {
    try {
      var job = await GetJobToRunAsync();
      await job.RunAsync();
    } catch (Exception ex) if (!ex.IsFatal()) {
      // Report the error and continue
      this.logger.LogError(ex);
    }
  }
}
```

The Work Dispatcher

We use .NET's Async functionality to spawn and manage a constant number of job workers for the lifecycle of the process.

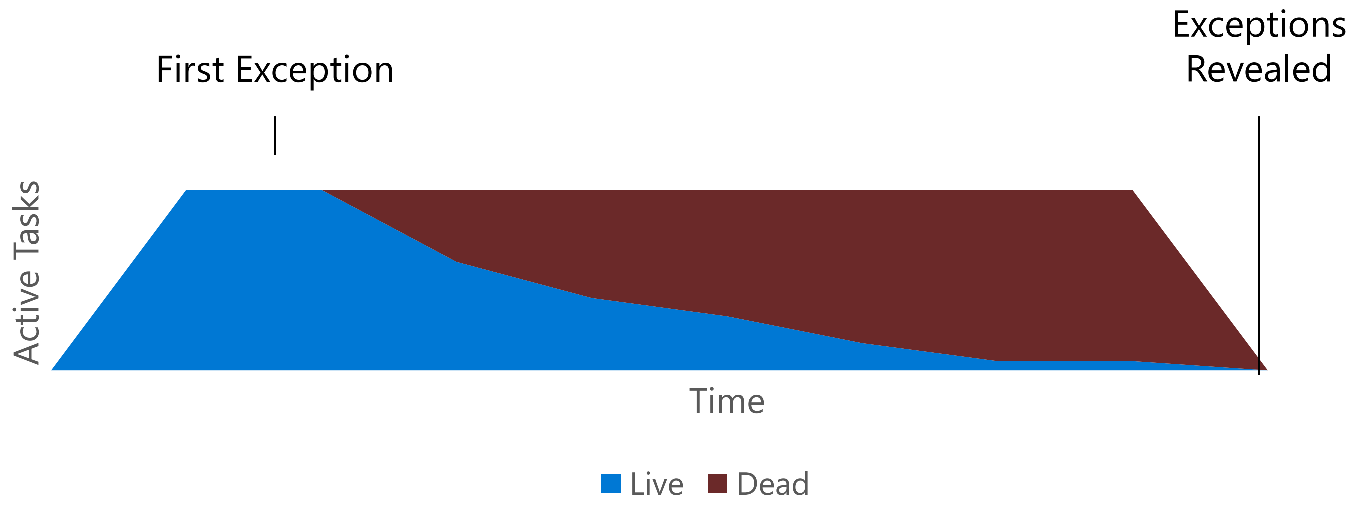
We used `Task.WhenAll` to subscribe to these worker tasks and catch any exceptions thrown by them.

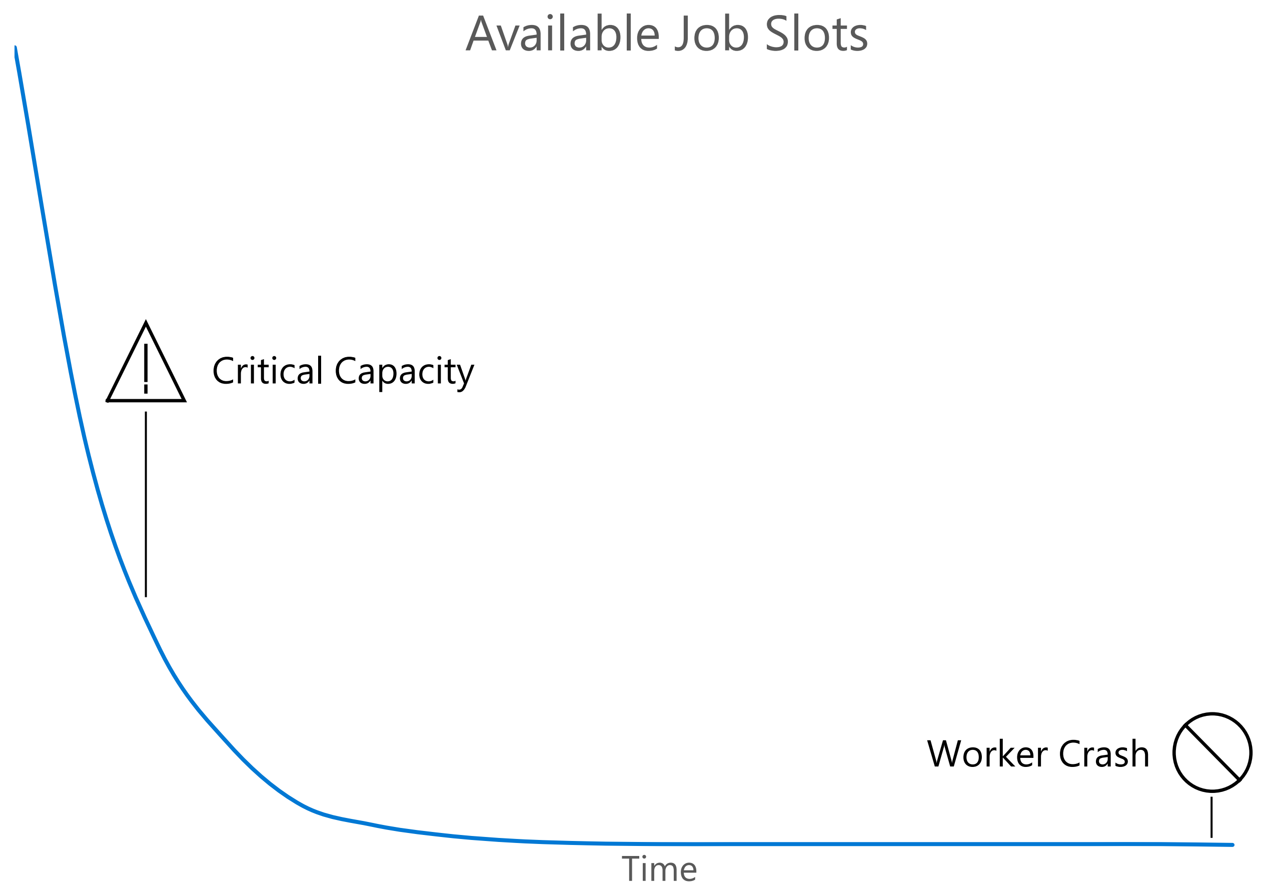
`Task.WhenAll` will only raise an exception once all tasks have finished executing. This buffered the fatal failures for hours before they were allowed to crash.

WorkDispatcher.cs

```
public async Task RunDispatcherAsync() {
  var workerTasks = new List<Task>();
  for (var i = 0; i < numberOfWorkers; i++) {
    workerTasks.Add(
      new JobWorker().RunJobsAsync());
  }

  // Wait for all workers to finish
  // processing before exiting.
  await Task.WhenAll(workerTasks);
}
```





How this failed

When this legacy job type was executed, it would poison a single job processing slot on the worker.

Workers start with ~1500 slots, and this job accounts for ~0.05% of executions.

The initial probability that a slot hits this issue is ~75%...

...the terminal probability is ~0.05%, greatly extending the tail.

How we’re improving

Key Repairs #1

Reducing Transitive Test Dependencies

- This helps reduce the differences between our test and production environments.

Reduce Reliance on Static Initializers

- Moving exception handling into constructors and/or other runtime code avoids exceptions being raised to **IsFatal** inadvertently and allows us to better degrade.

Improve visibility into uneven workloads

- We lacked good visibility into the shifts in work distribution which highlighted a problem in the first impacted regions, slowing our triage time.
- Encoding our system invariants (fair work distribution) into our monitors allows for significantly better sensitivity to unexpected shifts.

Key Repairs #2

Extended cache expiry for operation state

- Ensuring that this metadata is available for the duration of the valid job execution time ensures that future delays are less likely to result in failures for customers.

Improve visibility into queue pickup delays

- The lack of visibility into these delays contributed to us mis-identifying our vNext workers as the cause of the impact.
- Improvements in this space will allow us to better identify capacity-related problems in future.

Don’t rely on Task.WhenAll for job orchestration

- We have switched to a hybrid **Task.WhenAny** and **Task.WhenAll** approach to ensure that exceptions are handled in a timely manner.
- We have audited our other services to identify similar risks and apply similar mitigations.

Closing thoughts

“Outage severity cannot be *exclusively* measured in SLO impact.”

This outage represented a significant risk to our customers

…but it had an order of magnitude less impact on our SLOs than previous outages.

“You’re incentivized to optimize for your common failure modes.”

This outage was a novel and unexpected failure

…and our standard telemetry and processes were not equipped to deal with it.

“Complex systems run in degraded mode.”

- Richard I. Cook, MD

Our system operated for 8+ years with these latent defects

…and selection pressure eventually resulted in the perfect storm.

Microsoft

Thank you