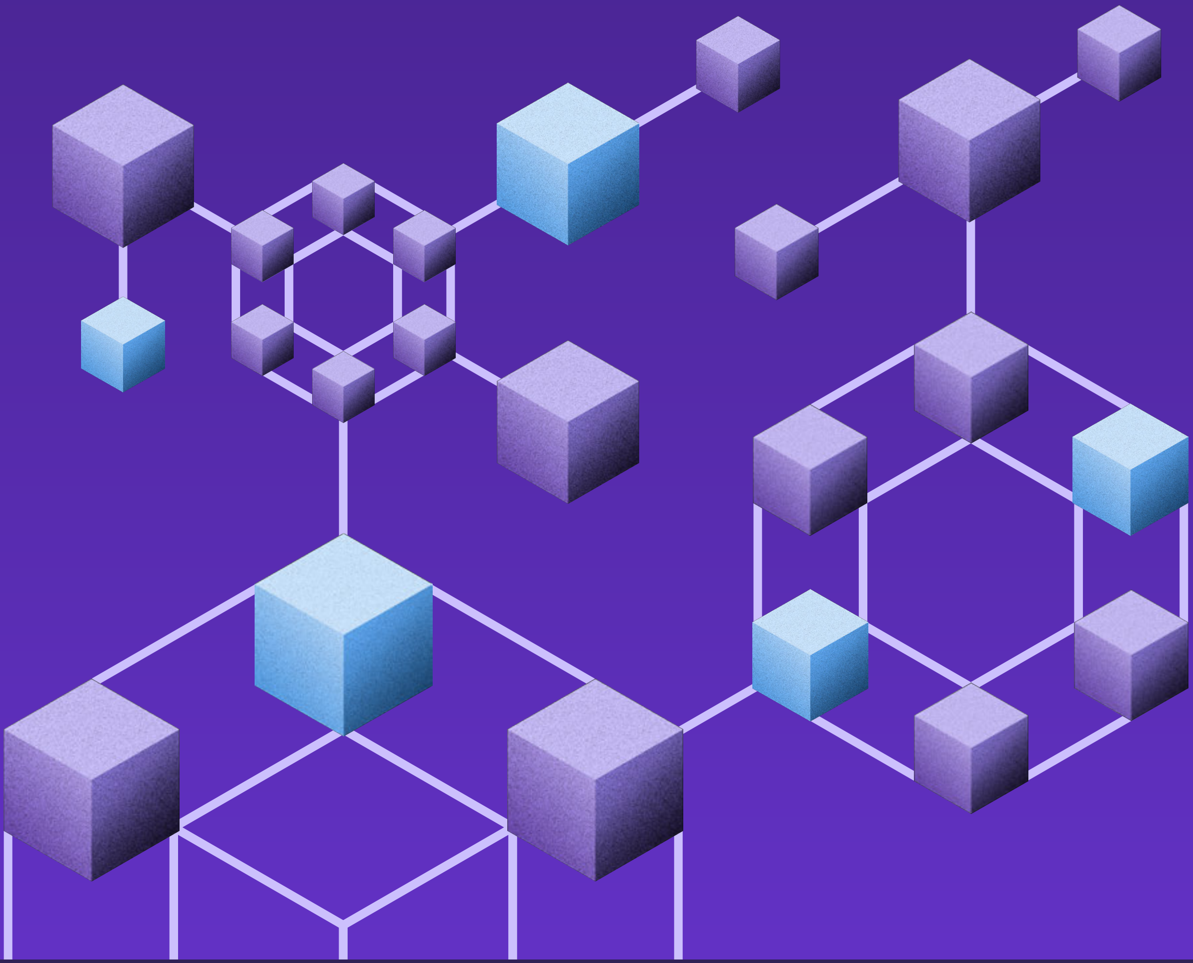
Gremlin

Kubernetes Reliability at Scale

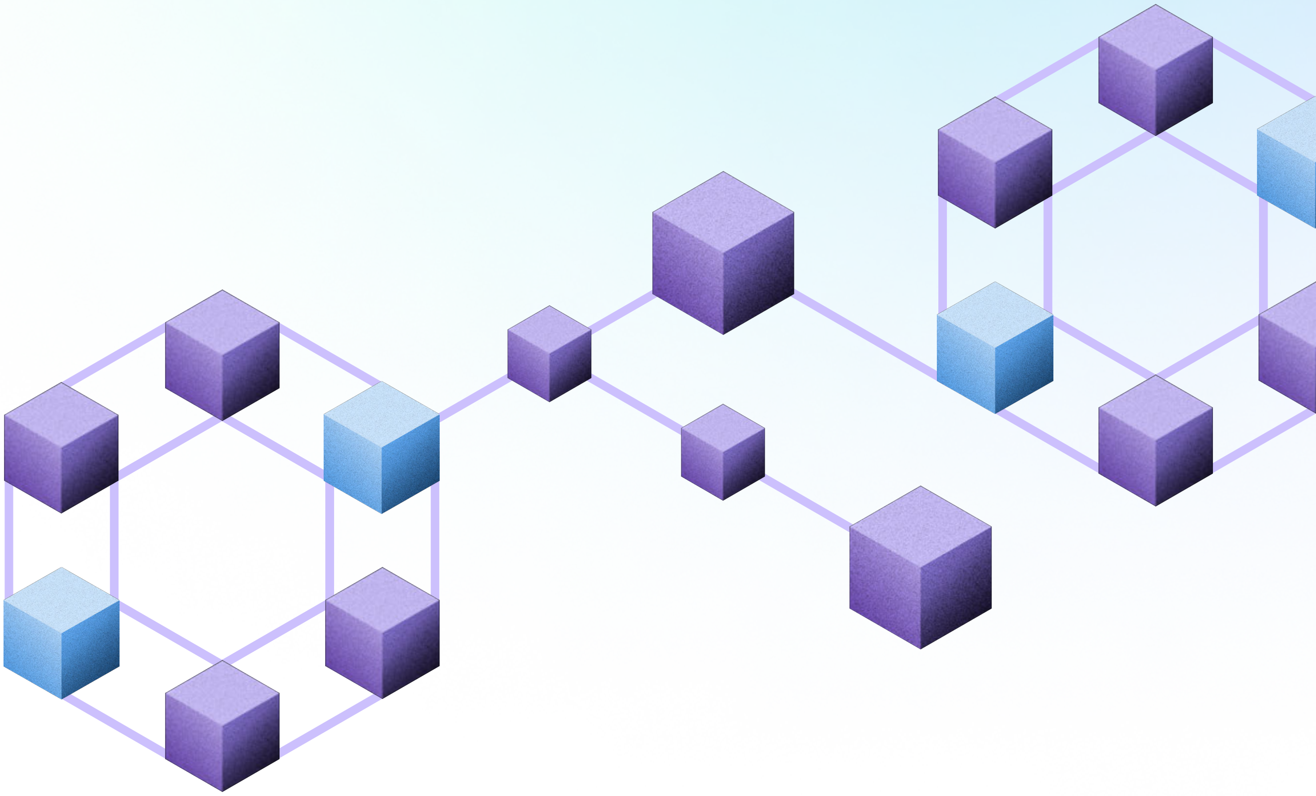
HOW TO IMPROVE UPTIME WITH RESILIENCY MANAGEMENT



ENTERPRISE RELIABILITY

EBOOK

Table of contents

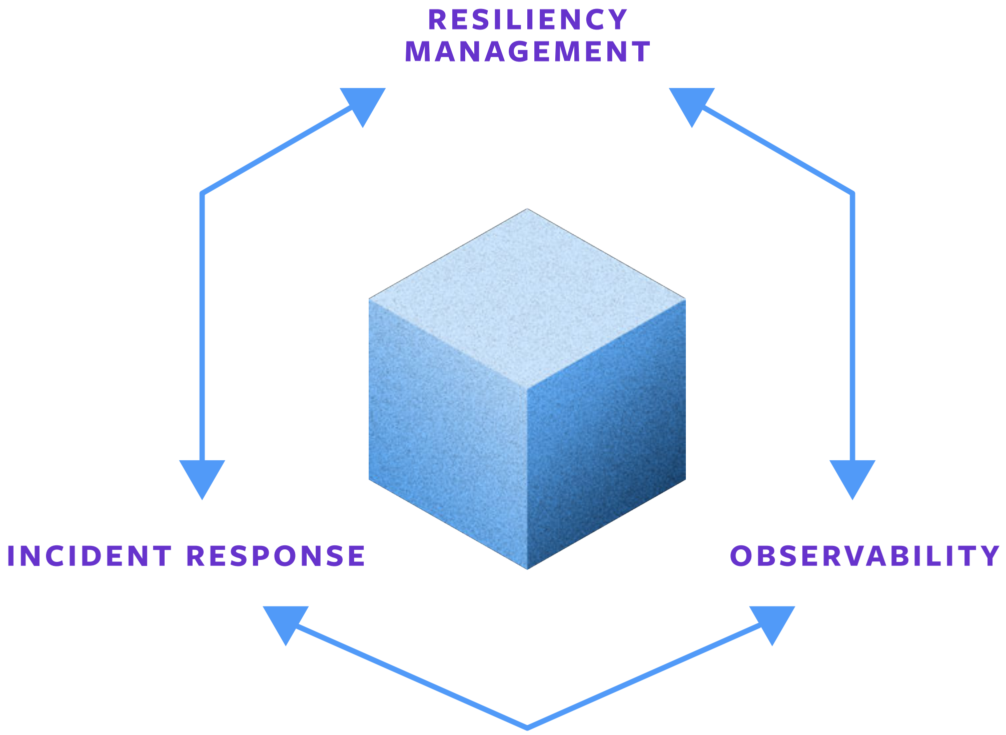


Introduction: The Kubernetes Reliability Gap

There's more pressure than ever to deliver high-availability Kubernetes systems. Consumers expect applications to be available at all times and have zero patience for outages or downtime. At the same time, businesses have created intricate webs of APIs and dependencies that rely on your applications.

Unfortunately, building reliable systems is easier said than done. Every system has potential points of failure that lead to outages—known as reliability risks. And when you're dealing with the complex, ephemeral nature of Kubernetes, there's an even higher possibility that risks will go undetected until they cause incidents.

Kubernetes Reliability



The traditional approach to reliability starts with using observability to instrument your systems. Any issue or non-optimum spike in your metrics creates an alert, which is then resolved using your incident response runbook.

This reactive approach can only surface reliability risks after the failure has occurred. This creates a gap between where you think the reliability of your system is and where you find out it actually is when there's an outage.

In order to meet the availability demands of your users, you need to fill in that gap with a standards-based approach to your system's resiliency.

By focusing on the resiliency of your Kubernetes redeployment, you can surface reliability risks proactively and address them before they cause outages and downtime.

Working together, these three practices allow your teams to get visibility into the performance of their deployments, proactively detect risks, and quickly resolve any failures that slip through the cracks—raising the reliability and availability of your system.

This book covers how to build your own resiliency management practice for Kubernetes. You'll learn:

- Common Kubernetes reliability risks and how to identify critical risks.
- How to monitor for critical risks and validate resilience to them.
- The key processes, standards, and roles needed to ensure resiliency across your organization.
- How to integrate resiliency management smoothly into your existing processes.

By using the framework and practices in this book, you'll be able to improve your Kubernetes uptime by preventing incidents and outages, accelerate key IT initiatives by improving your Kubernetes reliability posture, and shift-left reliability by integrating resiliency testing into your Software Development Lifecycle (SDLC).

AVAILABILITY VS. RESILIENCY VS. RELIABILITY

Customers and leadership often think in terms of availability, which comes from efforts to improve the resiliency and reliability of systems.

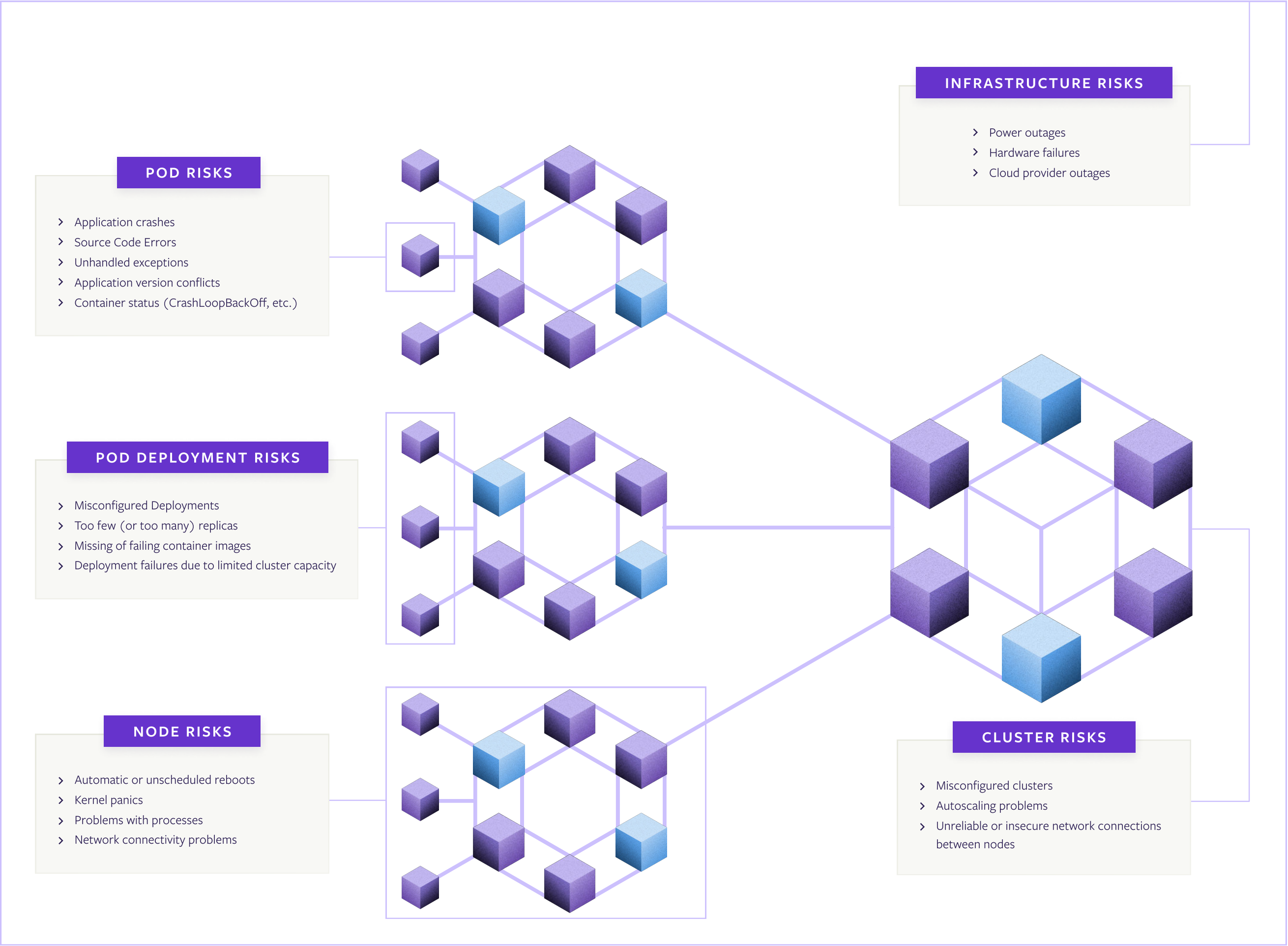
- **Availability** - A direct measure of uptime and downtime. Often measured as a percentage of uptime (e.g. 99.99%) or amount of downtime (e.g. 52.60 min/yr or 4.38 min/mo). This is a customer-facing metric mathematically computed by comparing uptime to downtime.
- **Resiliency** - A measure of how well a system can recover and adapt when there's disruptions, increased (or decreased) errors, network interruptions, etc. The more resilient a system is, the more it can respond correctly when changes occur.
- **Reliability** - A measure of the ability of a workload to perform its intended function correctly and consistently when it's expected to. The more reliable your systems, the more you and your customers can have confidence in them.

Reliability determines the actions your organization takes to ensure systems perform as expected, resiliency is how you improve the ability of your systems to respond as expected, and availability is the result of your efforts.

1. How Kubernetes can fail

Kubernetes is deployed in a series of distinct layers that are key to its resiliency and adaptability. The layers of pods, nodes, and clusters provide distinct separation that's essential for being able to scale up or down and restart as necessary while maintaining redundancy and availability.

Kubernetes reliability risks



INFRASTRUCTURE RISKS

- Power outages
- Hardware failures
- Cloud provider outages

POD RISKS

- Application crashes
- Source Code Errors
- Unhandled exceptions
- Application version conflicts
- Container status (CrashLoopBackOff, etc.)

POD DEPLOYMENT RISKS

- Misconfigured Deployments
- Too few (or too many) replicas
- Missing of failing container images
- Deployment failures due to limited cluster capacity

NODE RISKS

- Automatic or unscheduled reboots
- Kernel panics
- Problems with processes
- Network connectivity problems

CLUSTER RISKS

- Misconfigured clusters
- Autoscaling problems
- Unreliable or insecure network connections between nodes

Unfortunately, these same layers also increase the potential points of failure. Their interconnected nature can take a small error, such as a container consuming more CPU or memory than expected, then compound it across other nodes and cause a wide-scale outage.

The first step in Kubernetes resiliency management is to look at the potential reliability risks inherent in every Kubernetes deployment. As you build resiliency standards, you'll want to account for these reliability risks.

Underlying infrastructure risks

Every software application is dependent on the infrastructure below it for stability, and Kubernetes is no exception. These include risks like:

- Power outages
- Hardware failures
- Cloud provider outages

Architecting to minimize these risks includes multiple Availability Zone redundancy, multi-region deployments, and other best practices. You'll need to be able to test whether these practices are in place for your system to be resilient to these failures.

Cluster risks

Clusters lay out the core policies and configurations for all of the nodes, pods, and containers deployed within them. When reliability risks occur at the cluster level, these often become endemic to the entire deployment, including:

- Misconfigured control plane nodes
- Autoscaling problems
- Unreliable or insecure network connections between nodes

Detecting and testing for the risks will include looking at your cluster configuration, how it responds to increases in resource demands, and its response to changing network activity.

Node risks

Any reliability risks in the nodes will immediately inhibit the ability to run pods. While cloud-hosted Kubernetes providers can automatically restart problem nodes, if there's core issues with the control plane, the problem will just be replicated again. These issues could include:

- Automatic or unscheduled reboots
- Kernel panics
- Problems with the kubelet process or other Kubernetes-related process
- Network connectivity problems

Node-based reliability risks are usually focused around the ability of the node to communicate with the rest of the cluster, get the resources it needs, and correctly manage pods.

Pod deployment risks

The node may be configured correctly, but there can still be issues and risks related directly to deploying the pods itself within the node, such as:

- Misconfigured deployments
- Too few (or too many) replicas
- Missing or failing container images
- Deployment failures due to limited cluster capacity

These risks are tied directly to the ability (or inability) of nodes to correctly deploy pods within them, and are often tied to limited resources or finding/communicating with container images.

Pod risks

Even if everything goes well with pod deployment, there can still be issues that affect the reliability of the application running within the pods, including:

- Application crashes
- Source code errors
- Unhandled exceptions
- Application version conflicts

Many of these “last-mile” risks can be uncovered by monitoring and testing for specific Kubernetes states, such as CrashLoopBackOff or ImagePullBackOff.

How to determine which Kubernetes risks are critical

Unfortunately, there’s never enough time or money to fix every single reliability risk. In the well-known balance between expense, quality, and speed, the demands of business make it impossible. Instead, you need to find that balance where you’re addressing the critical reliability risks that could have the greatest impact and deprioritizing the risks that would have a minor impact. When you consider the number of moving pieces and potential reliability risks present in Kubernetes, this kind of prioritization and identification becomes even more important.

The exact list will change from organization to organization and service to service, but you can start by looking at some of the core resiliency features in Kubernetes:

- **Scalability** - Can the system quickly respond to changes in demand?
- **Redundancy** - Can applications keep running if part of the cluster fails?
- **Recoverability** - Can Kubernetes recover if something fails?
- **Consistency/Integrity** - Are pods using the same image and running smoothly?

Any issue that interferes with these capabilities poses a critical risk to your Kubernetes deployment. Resiliency management takes a systematic approach to surfacing these reliability risks across your organization.

Finding and resolving risks at scale needs a standardized approach

Kubernetes and rapid-deployment Software Development Lifecycles (SDLCs) go hand in hand. At the same time, the interconnected nature of Kubernetes means that building reliability requires clear standards and governance to ensure uniform resiliency across all your various services.

Traditionally, these two practices have been at odds. Governance and heavy testing gates tend to slow down deployments, while a high-speed DevOps approach stresses a fast rate of deployment and integration.

If you're going to bring the two together, you need a different approach, one based on standards, but with automation and technological capabilities that allow every team to uncover reliability risks in any Kubernetes deployment with minimal lift.

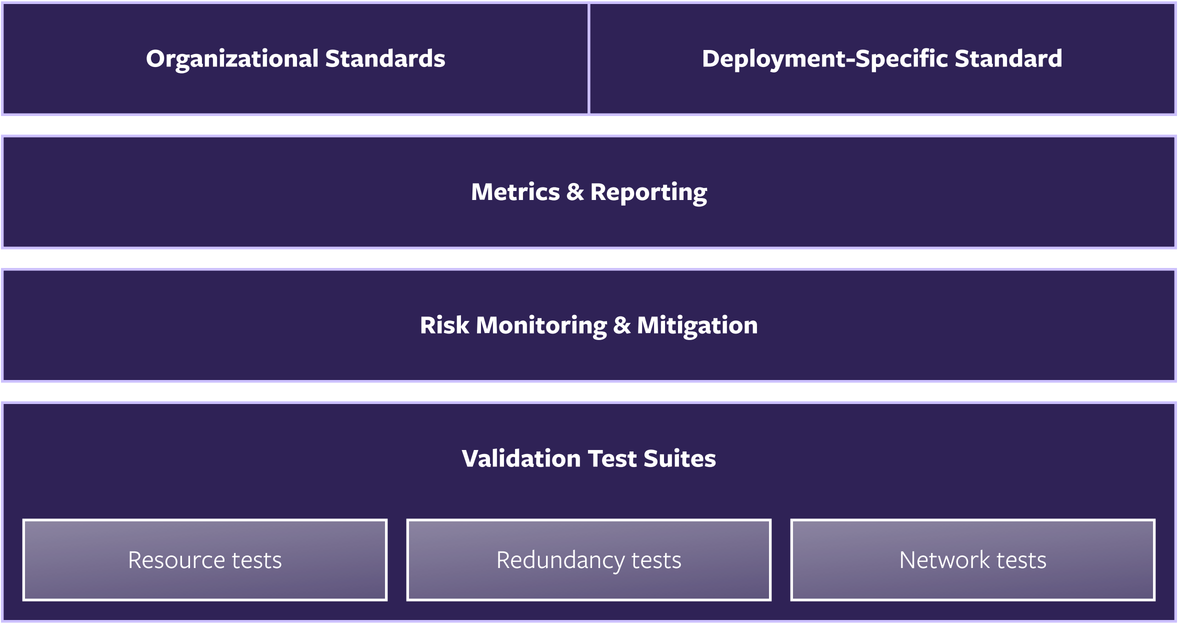
The approach needs to be able to surface known risks from all layers of a Kubernetes deployment, as well as uncover unknown, deployment-specific risks. Just as importantly, this practice needs to include standardized metrics and processes that can be used across the organization so all Kubernetes deployments are held to the same reliability standards.

2. Framework for Kubernetes resiliency

Any approach for Kubernetes resiliency would have to combine the known possible reliability risks above with the criteria for critical risk identification and organization-wide governance.

When these are paired with the technology of Fault Injection testing, it creates a resiliency management framework that combines automated validation testing, team-based exploratory testing, and continuous risk monitoring with the reporting and processes to remediate risks once found.

Framework for Kubernetes Resiliency



Organizational Standards | Deployment-Specific Standard

Metrics & Reporting

Risk Monitoring & Mitigation

Validation Test Suites

Resource tests | Redundancy tests | Network tests

Resiliency standards

Some reliability risks are common to all Kubernetes deployments. For example, every Kubernetes deployment should be tested against how it's going to respond during a surge in demand for resources, a drop in network communications, or a loss of connection to dependencies.

These are recorded under **Organizational Standards**, which inform the standard set of reliability risks that every team should test against. While you start with common reliability risks, this list should expand to include risks unique to your company that are common across your organization. For example, if every service connects to a specific database, then you should standardize around testing what happens if there's latency in that connection.

Deployment-Specific Standards are deviations from the core Organizational Standards for specific services or deployments. The standards can be stricter or looser than organizational standards, but either way, they're exceptions that should be noted. For example, an internal sales tool might have a higher latency tolerance for connecting to a database because your team is more willing to wait, while an external checkout service might be required to move to a replicated copy of the database faster than normal to avoid losing sales.

Metrics and reporting

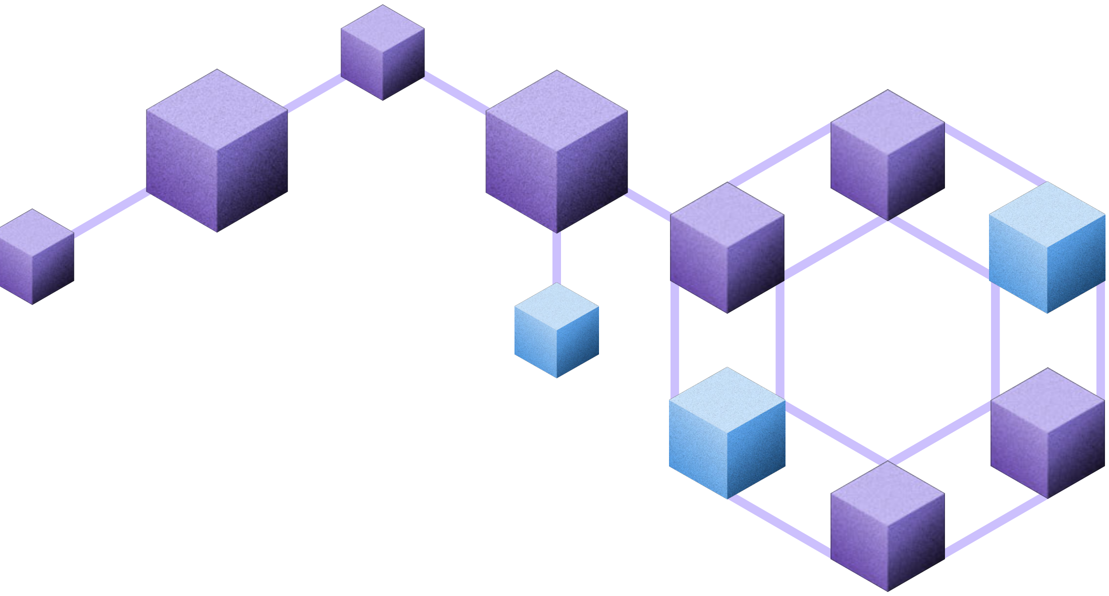
Reliability is often measured by either the binary "Currently up/currently down" status or the backwards-facing "uptime vs. downtime" metric. But neither of these measurements will help you see the posture of potential reliability risks before they become outages—and whether you've addressed them or gained more risks over time.

This is why it's essential to have metrics, reporting, and dashboards that show the results of your resiliency tests and risk monitoring. These dashboards give the various teams core data to align around and be accountable for results. By showing how each service performs on tests built against the defined resiliency standards, you get an accurate view of your reliability posture that can inform important prioritization conversations.

Risk monitoring and mitigation

Some Kubernetes risks, such as missing memory limits, can be quick and easy to fix, but can also cause massive outages if unaddressed. The complexity of Kubernetes can make it easy to miss these issues, along with other known reliability risks common across all Kubernetes deployments, which means you can operationalize their detections.

Many of these critical risks can be located by scanning configuration files and container statuses. These scans should run continuously on Kubernetes deployments so these risks can be surfaced and addressed quickly.



Validation testing using standardized test suites

Utilizing Fault Injection, resiliency testing safely creates fault conditions in your deployment so you can verify that your systems respond the way you expect them to, such as a spike in CPU demand or a drop in network connectivity to key dependencies.

Using the standards from the first part of the framework, suites of reliability tests can be created and run automatically. This validation testing approach uncovers places where your systems aren't meeting standards, and the pass/fail data can be used to create metrics that show your changing reliability posture over time.

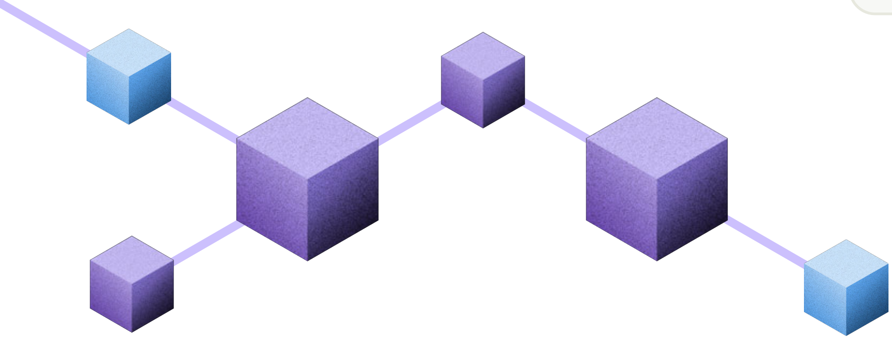
VALIDATION TESTING VS. EXPLORATORY TESTING

Fault Injection testing is commonly used in one of two ways:

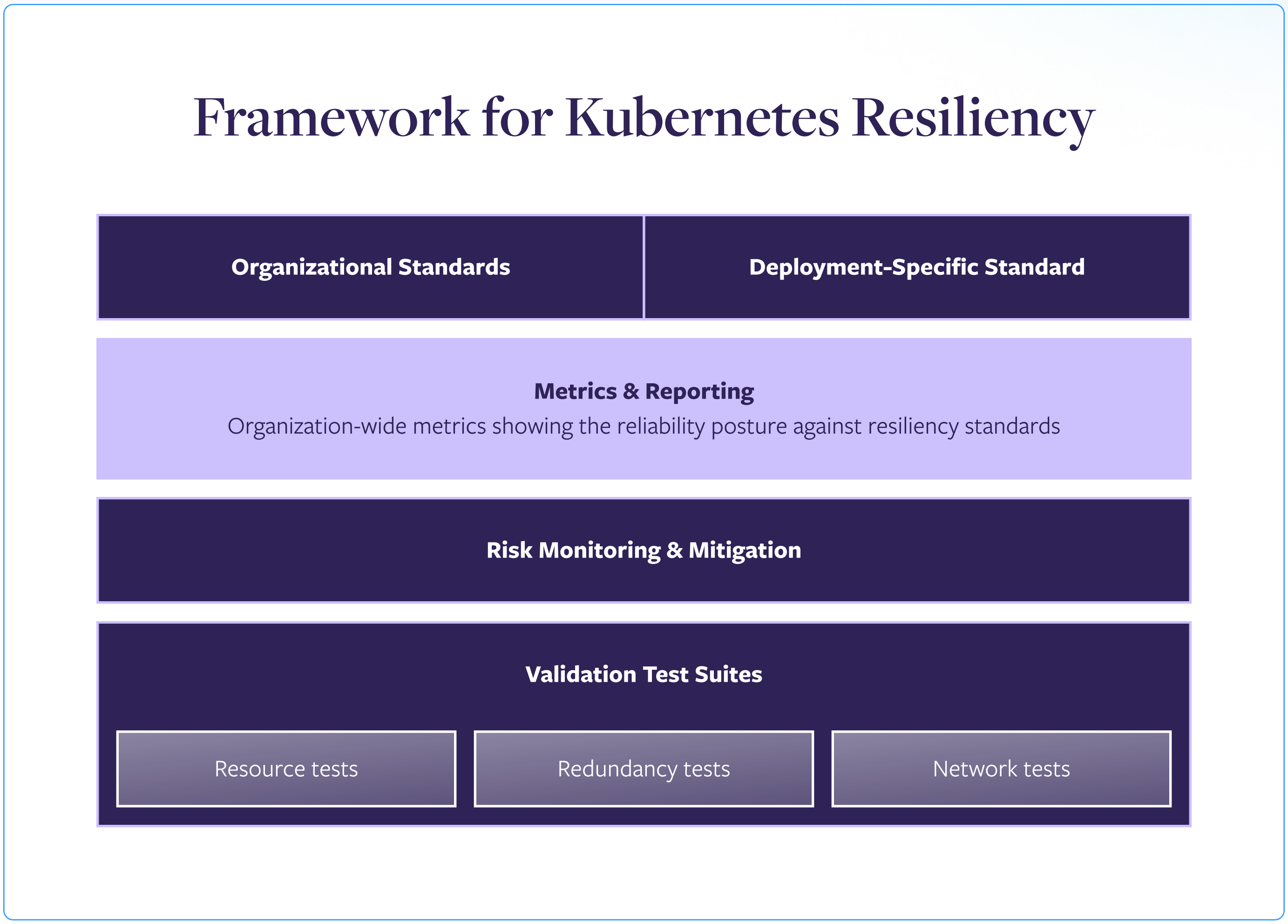
- **Validation testing** - You have a standard, state, or policy that you're testing against to validate that your system meets the specific requirements. For example, if a pod reaches its CPU limit, is a new pod spun up to take the extra load? In this case, you inject the fault and verify that your system reacts the way it's supposed to.

- **Exploratory testing** - You want to uncover specific unknown failures, such as what happens if an init container is unavailable for 5 minutes. Does the pod load without it, or does it crash? Exploratory testing should be done by someone trained in how to safely define the experiment to minimize the impact.

Learn more about the difference between them in The two kinds of failure testing blog post.



3. Metrics and reporting



Framework for Kubernetes Resiliency

Organizational Standards

Deployment-Specific Standard

Metrics & Reporting
Organization-wide metrics showing the reliability posture against resiliency standards

Risk Monitoring & Mitigation

Validation Test Suites

Resource tests

Redundancy tests

Network tests

Why it’s important to track reliability

Ask the executives and engineers at any organization and they’ll all agree that reliability could be improved. But when you ask them how, that’s usually where the discussion breaks down.

Identify risks that need to be addressed

Most organizations lack a consistent, agreed-upon method for identifying reliability risks that can be shared and understood across their teams. It's not that the information isn't out there—almost every engineer knows the common ways their service will fail—it's that there's no centralized way for all reliability risks and potential failure points to be cataloged, tested for, and compared between services.

Tracking resilience tests gives you that central alignment. When you track the results over time, individual teams can show exactly what risks are and aren't present in their services, taking that knowledge out of the engineer's heads and putting it into a place where the entire organization can benefit from it.

Prove the results of engineering efforts

Reliability tracking also provides a framework to prove the effectiveness of a team's efforts. Without a framework, a well-intentioned engineer could spend hours addressing an issue that they know could lead to an outage, but end up with little to no recognition or acknowledgement of their efforts. This is because they're attempting to prove a negative. Yes, they prevented an outage, but how can they show that they stopped an outage that didn't happen or fixed an issue that's no longer there?

By tracking reliability risks over time, engineers and operators can show the effectiveness of their efforts by pointing to the test that previously failed but now passes, proving that the risk is no longer present.

Create a common reliability metric across the organization

Finally, tracking reliability risks creates a metric that can be used to track reliability over time across the organization. This is where standards and testing come together to produce actionable organizational alignment.

By laying out the standardized test suites everyone should follow, you create a list of reliability risks everyone should track.

Over time, this creates a metric where the entire team can align around common reliability metrics and get an accurate picture of the reliability posture of their entire Kubernetes system.

Reliability scores and dashboards

Tracking the results of resiliency tests makes it possible for each Kubernetes service to be given a reliability score. These scores, in turn, can create dashboards where the scores of all Kubernetes services are rolled up for review and alignment, thus creating a view of the entire deployment's reliability posture.

How reliability scoring works

The current status of every resiliency test falls into one of three results:

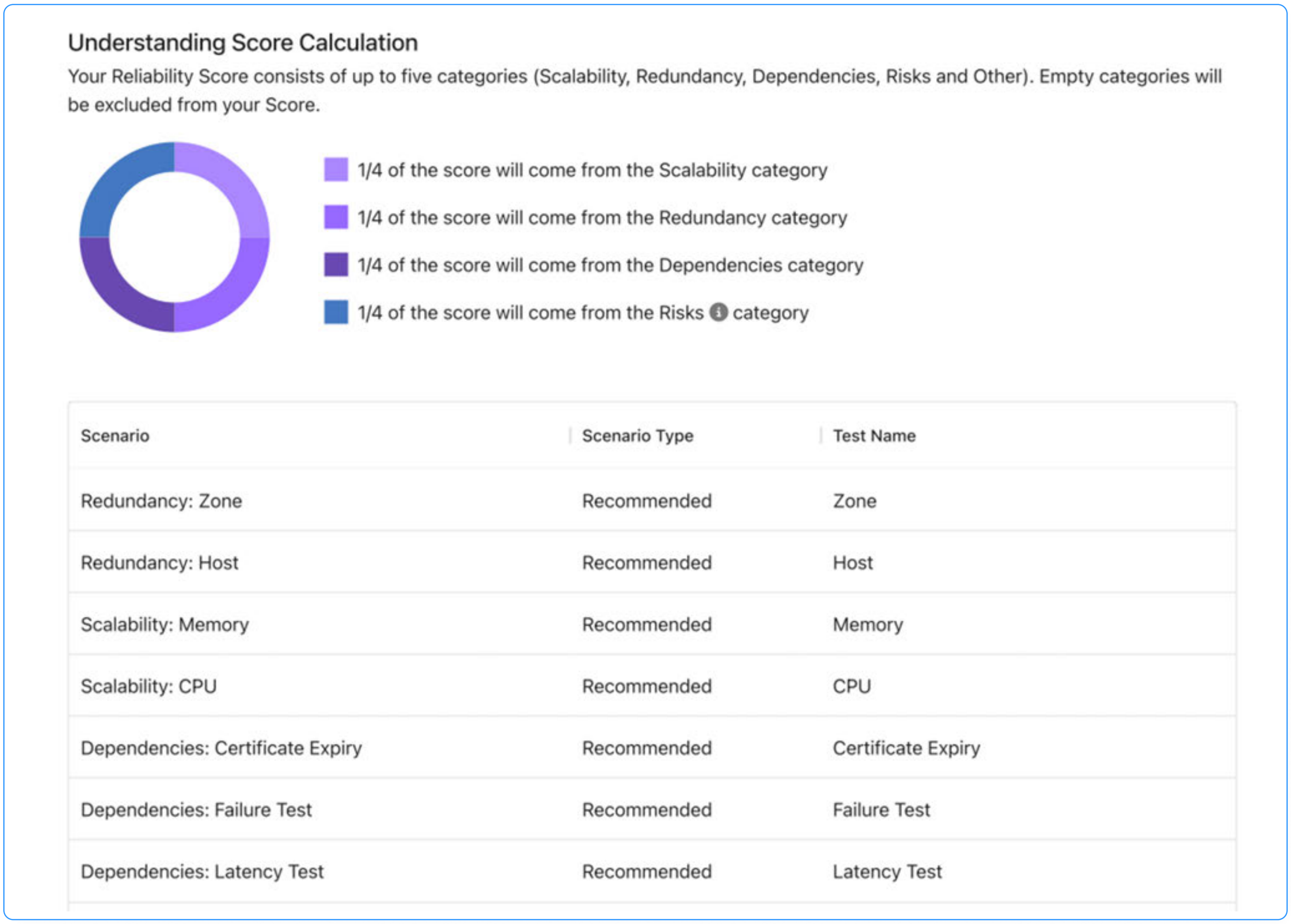
1. **Passed**
   The deployment performed as expected and no reliability risk exists.

2. **Failed**
   The deployment did not perform as expected, and a reliability risk is known to exist.

3. **Not run**
   The test hasn't been run recently enough to be certain of the result. A known reliability risk may or may not exist—which is, in itself, a reliability risk.

When you're looking at a service's reliability posture, you're only concerned about whether a reliability risk is present. If a risk is present, then you need to evaluate whether engineering time and effort should be spent resolving the risk. If not, then you can count on your system to be resilient in that area without further engineering effort.

By looking at it this way, test results can be pooled into a binary state where a point is scored for any passed tests (no reliability risk present) and a zero is scored for a failed or not-run test (known or possible reliability risk present).



Understanding Score Calculation

Your Reliability Score consists of up to five categories (Scalability, Redundancy, Dependencies, Risks and Other). Empty categories will be excluded from your Score.

- 1/4 of the score will come from the Scalability category
- 1/4 of the score will come from the Redundancy category
- 1/4 of the score will come from the Dependencies category
- 1/4 of the score will come from the Risks ℹ category

Scenario	Scenario Type	Test Name
Redundancy: Zone	Recommended	Zone
Redundancy: Host	Recommended	Host
Scalability: Memory	Recommended	Memory
Scalability: CPU	Recommended	CPU
Dependencies: Certificate Expiry	Recommended	Certificate Expiry
Dependencies: Failure Test	Recommended	Failure Test
Dependencies: Latency Test	Recommended	Latency Test

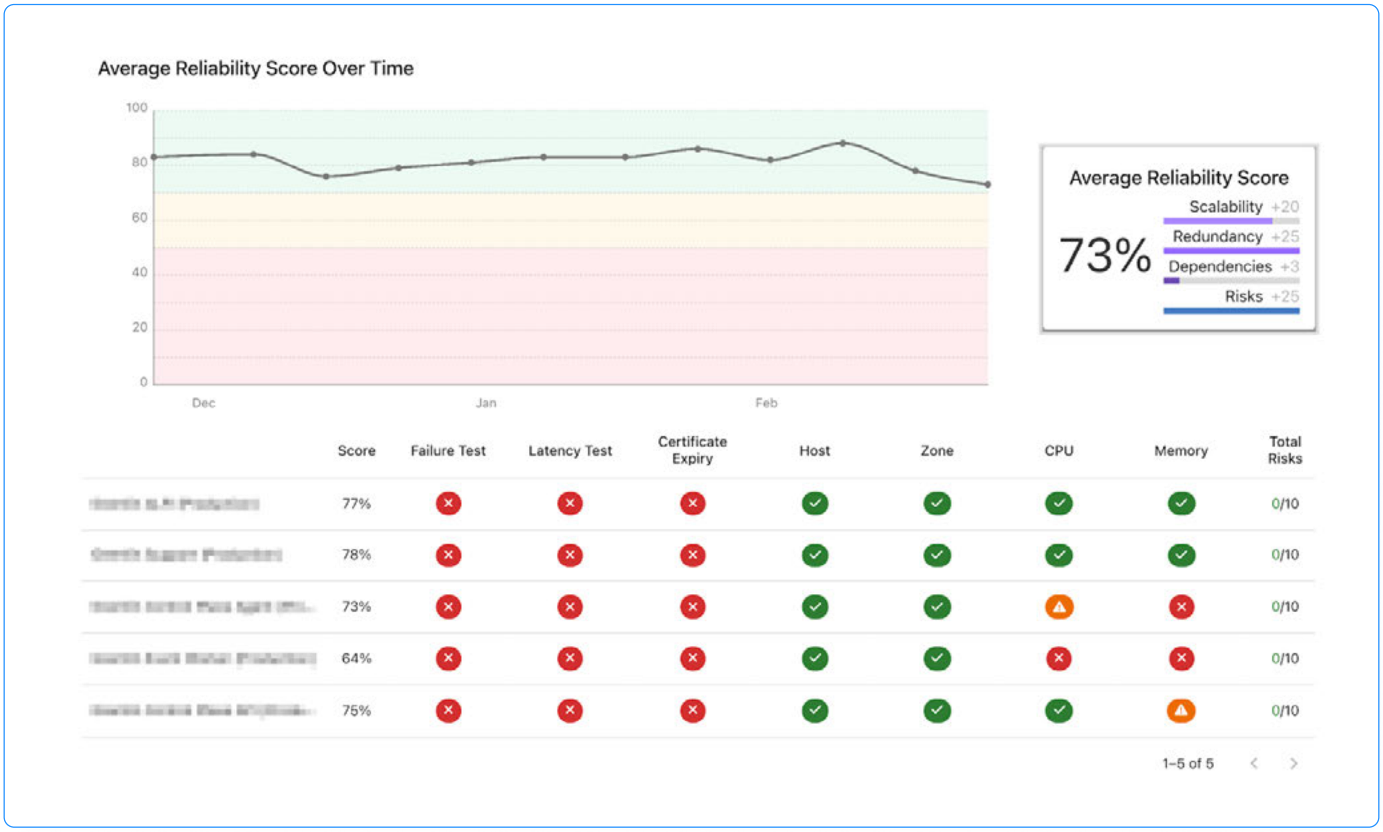
Example of how scores can be computed in Gremlin

When we compile the results of an entire suite of tests, a score is created.

How regular testing creates a metric of scores

When you run a series of tests to build a reliability score, this creates a numeric data point that shows the reliability posture of your Kubernetes deployment at a specific time.

By regularly running resiliency test suites, you create a metric of your reliability posture over time. Like any metric, this can be plotted to show trends, then each data point can be drilled down to the individual test results.



Average Reliability Score

73%

Scalability +20

Redundancy +25

Dependencies +3

Risks +25

	Score	Failure Test	Latency Test	Certificate Expiry	Host	Zone	CPU	Memory	Total Risks
[redacted]	77%	✗	✗	✗	✓	✓	✓	✓	0/10
[redacted]	78%	✗	✗	✗	✓	✓	✓	✓	0/10
[redacted]	73%	✗	✗	✗	✓	✓	⚠	✗	0/10
[redacted]	64%	✗	✗	✗	✓	✓	✗	✗	0/10
[redacted]	75%	✗	✗	✗	✓	✓	✓	⚠	0/10

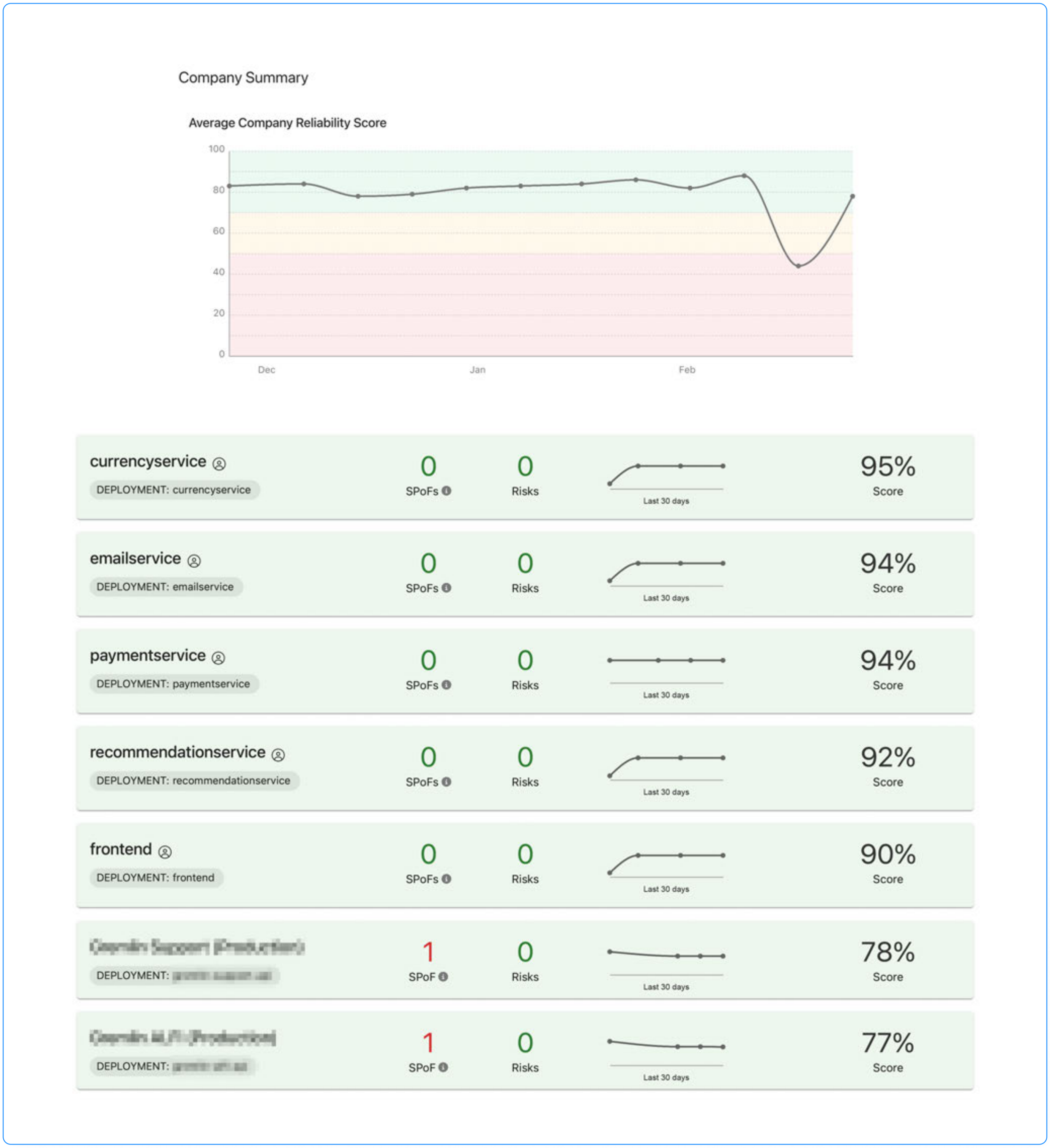
1–5 of 5 ‹ ›

Example of how scores can be computed in Gremlin

Every organization will have different requirements, and your standards owner should set your specific testing cadence, but a good goal is to work towards weekly testing of production systems. A weekly cadence gives you an accurate view that will always be recent enough to be considered current, and by testing in production, you'll be getting an accurate view of your actual Kubernetes deployment under real-world conditions

Create dashboards for alignment and reporting

By combining reliability scores with regular testing, you create reliability metrics. So the next step is to create a system for reporting those metrics with dashboards.



Company Summary

Average Company Reliability Score

currencyservice DEPLOYMENT: currencyservice	0 SPoFs	0 Risks	Last 30 days	95% Score
emailservice DEPLOYMENT: emailservice	0 SPoFs	0 Risks	Last 30 days	94% Score
paymentservice DEPLOYMENT: paymentservice	0 SPoFs	0 Risks	Last 30 days	94% Score
recommendationservice DEPLOYMENT: recommendationservice	0 SPoFs	0 Risks	Last 30 days	92% Score
frontend DEPLOYMENT: frontend	0 SPoFs	0 Risks	Last 30 days	90% Score
[redacted] DEPLOYMENT: [redacted]	1 SPoF	0 Risks	Last 30 days	78% Score
[redacted] DEPLOYMENT: [redacted]	1 SPoF	0 Risks	Last 30 days	77% Score

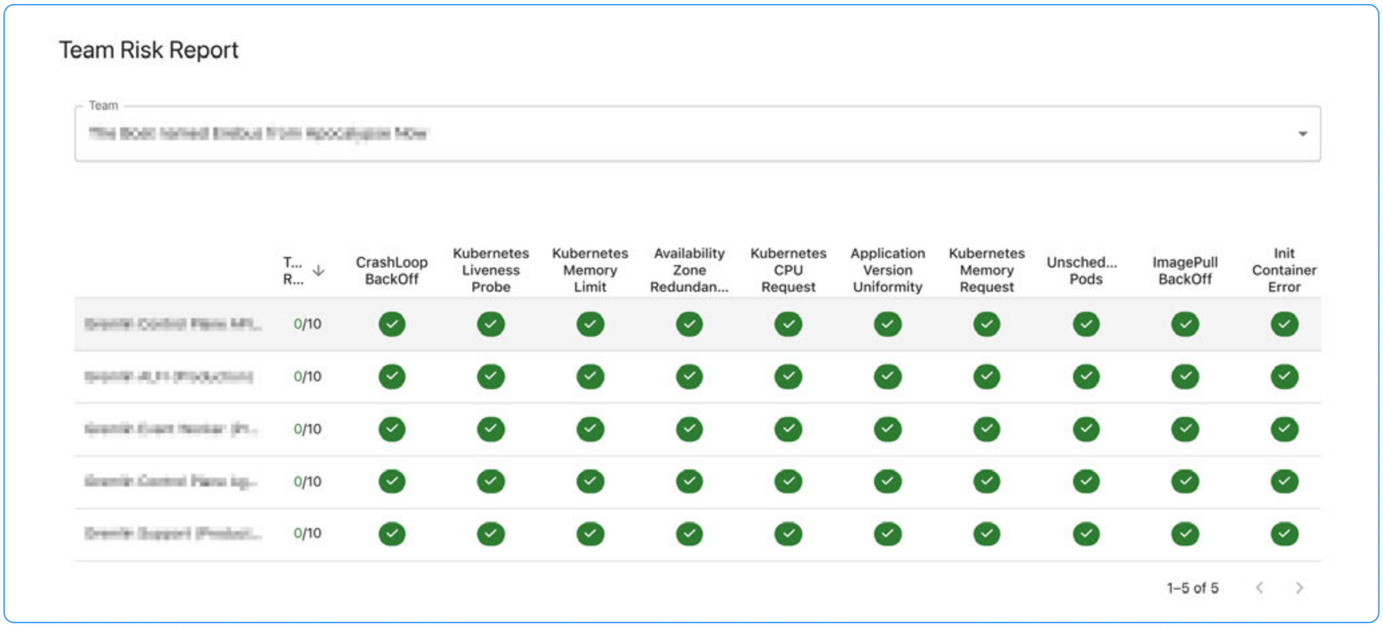
Example of a dashboard for multiple services in Gremlin

These dashboards should be used in regular reliability alignment meetings (or as part of existing engineering review meetings) for the entire team—including leadership—to review the current reliability posture of your Kubernetes deployment.

The goal with these dashboards isn't to assign blame or point out failures. Instead, they should be used to plan engineering work and applaud successes. For example, if a team shipped a new feature and their reliability score decreased, this might be expected with the large amount of new code added to the system. The decrease in score then shows the team that time should be spent ensuring reliability of the new feature before moving onto the next one. At the same time, if they come back two weeks later and the score has increased, then they should be celebrated for how much they improved the new feature's reliability.

Tracking detected Kubernetes reliability risks

The nature of Kubernetes cluster and node configurations make it possible to continuously scan and monitor for known critical reliability risks, such as misconfigurations that would disable autoscaling. These risks and how to monitor for them are discussed in more detail below, but these also create their own reliability metric unique to Kubernetes.



Team Risk Report

Team: [redacted]

	T... R... ↓	CrashLoop BackOff	Kubernetes Liveness Probe	Kubernetes Memory Limit	Availability Zone Redundan...	Kubernetes CPU Request	Application Version Uniformity	Kubernetes Memory Request	Unsched... Pods	ImagePull BackOff	Init Container Error
[redacted]	0/10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
[redacted]	0/10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
[redacted]	0/10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
[redacted]	0/10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
[redacted]	0/10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

1–5 of 5 ‹ ›

A sample team risk report from Gremlin

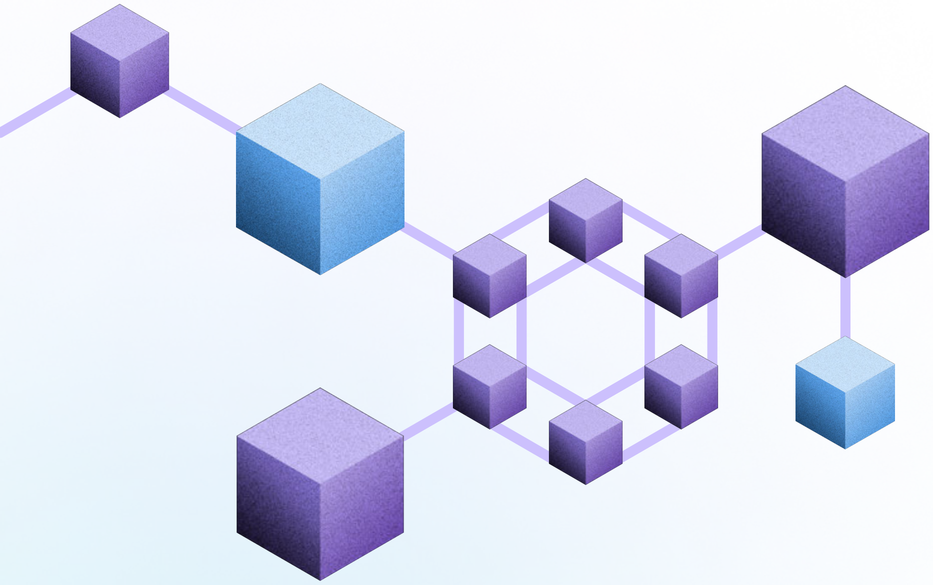
As with reliability scores, these detected risks can be broken down into a binary metric: either the risk is present or it isn't. And just like with reliability scores, tracking the detection of these risks over time creates a reliability metric. Like with reliability metrics gained from testing, these scanned reliability metrics should be reviewed in regular alignment meetings, then be used to show when teams have successfully addressed the risks to make the systems more reliable.

FURTHER READING:

By tracking reliability metrics, you enable your organization to operate on a whole new level of reliability, one where they have the processes and tooling in place to find reliability risks, prioritize which risks need the most attention, and then report back the results to the greater business.

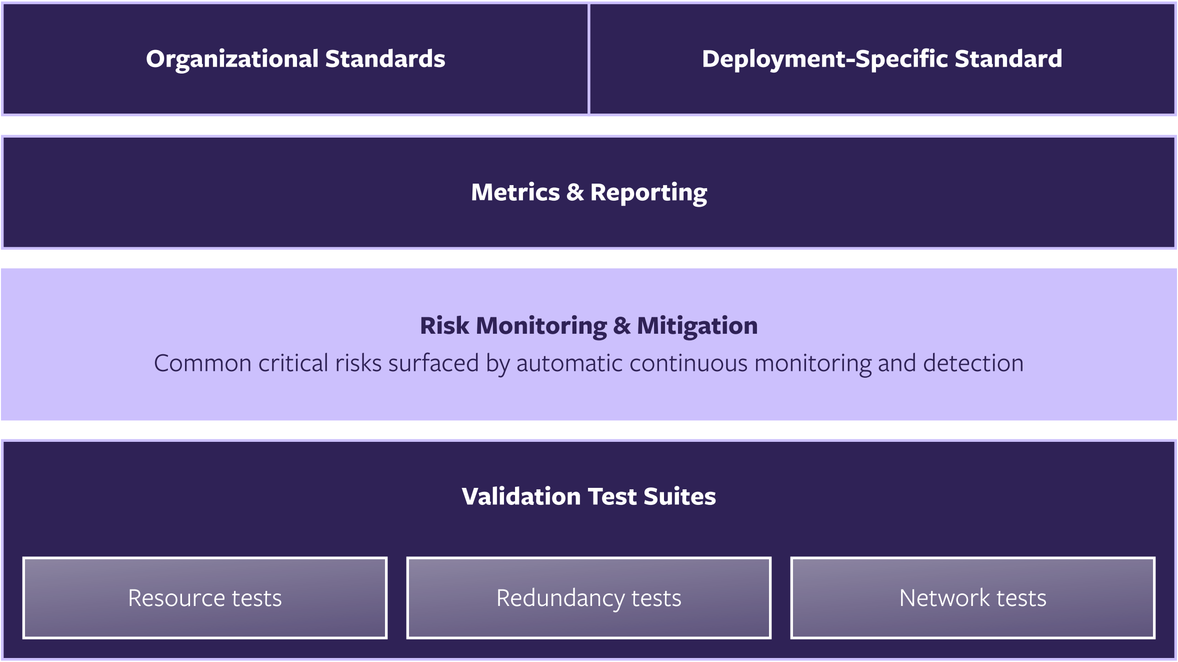
Find out more in these resources from Gremlin:

- Navigating the Reliability Minefield: Find and Fix Your Hidden Reliability Risks whitepaper
- More Reliability, Less Firefighting: How to Build a Proactive Reliability Program on-demand webinar



4. Risk monitoring and mitigation

Framework for Kubernetes Resiliency



Organizational Standards

Deployment-Specific Standard

Metrics & Reporting

Risk Monitoring & Mitigation

Common critical risks surfaced by automatic continuous monitoring and detection

Validation Test Suites

Resource tests

Redundancy tests

Network tests

While resiliency testing is necessary for uncovering some reliability risks, the nature of Kubernetes makes it possible to scan for key misconfigurations, bad default values, or anti-patterns that create reliability risks within the cluster. You can deploy a tool across your cluster to detect Kubernetes resources and analyze configurations across the deployment. This makes it possible to automatically detect key reliability risks and surface them before they start causing behavior that could lead to an outage.

This kind of automated risk monitoring is different than observability or resiliency testing. With observability, these risks will present themselves when they create unexpected behavior in your systems. Instead of finding out about the risk ahead of time, you're reacting after it's already caused an incident or outage. Resiliency testing, on the other hand, artificially injects faults that would trigger the risk. This allows you to uncover the risk proactively before it causes problems, but the tests themselves have to be run.

Kubernetes risk monitoring uses the cluster, node, and pod data to uncover critical reliability risks automatically without testing or waiting for an observability alert. Many of these are caused by configuration issues or require small changes to images that can be relatively quick to address. By setting up a system to monitor for these key risks, you can proactively surface them without the delay of other methods.

The nature of Kubernetes and the complexity of deployments has the potential to create a large number of risks, but there's a core group of ten that should be included in any risk monitoring practice. These are the most common critical risks that could cause major outages if left unaddressed. When building out your Kubernetes reliability tooling and standards, start by making sure these ten reliability risks are being detected and covered. From there, you can add other reliability risks to your monitoring list.

Resource risks

Running out of resources directly impacts system stability. If your nodes don't have enough CPU or RAM available, they may start slowing down, locking up, or terminating resource-intensive pods to make room.

Setting requests is the first step towards preventing this, because they specify the minimum resources needed to run a pod. Limits are somewhat the opposite and set an upper cap on how much RAM a pod can use, preventing a memory leak from taking all of a node's resources.

Missing CPU requests

A common risk is deploying pods without setting a CPU request. While it may seem like a low-impact, low-severity issue, not using CPU requests can have a big impact, including preventing your pod from running.

FURTHER READING

Find out how to detect missing CPU requests and how to resolve the reliability risk: How to ensure your Kubernetes Pods have enough CPU

Requests serve two key purposes:

1. They tell Kubernetes the minimum amount of the resource to allocate to a pod. This helps Kubernetes determine which node to schedule the pod on and how to schedule it relative to other pods.
2. They protect your nodes from resource shortages by preventing over-allocating pods on a single node.

Without this, Kubernetes might schedule a pod onto a node that doesn't have enough capacity for it. Even if the pod uses a small amount of CPU at first, that amount could increase over time, leading to CPU exhaustion.

Missing memory requests

A memory request specifies how much RAM should be reserved for a pod's container. When you deploy a pod that needs a minimum amount of memory, such as 512 MB or 1 GB, you can define that in your pod's manifest. Kubernetes then uses that information to determine where to deploy the pod so it has at least the amount of memory requested.

When deploying a pod without a memory request, Kubernetes has to make a best-guess decision about where to deploy the pod.

If the pod gets deployed to a node with a limited amount of free memory remaining, and the pod gradually consumes more memory over time, it could trigger an out of memory (OOM) event that terminates the pod. This could even make the pod unschedulable, which manifests as the CrashLoopBackOff status.

FURTHER READING

Learn more about finding and resolving memory request risks: How to ensure your Kubernetes Pods have enough memory

Missing memory limits

A memory limit is a cap on how much RAM a pod is allowed to consume over its lifetime. When you deploy a pod without memory limits, it can consume as much RAM as it wants, just like any other process. If it continually uses more and more RAM without freeing any (known as a memory leak), eventually the host it's running on will run out of RAM.

At that point, a kernel process called the OOMKiller jumps in and terminates the process before the entire system becomes unstable.

While the OOMKiller should be able to find and stop the appropriate pod, it's not always guaranteed to be successful. If it doesn't free enough memory, the entire system could lock up, or it could kill unrelated processes to try and free up enough memory.

FURTHER READING

Find out how to set memory limits and prevent memory leaks: How to detect and prevent memory leaks in Kubernetes applications

Setting a limit and a request creates a range of memory that the pod could consume, making it easier for both you and Kubernetes to determine how much memory the pod will use on deployment.

Redundancy risks

Unfortunately, containers often crash, terminate, or restart with little warning. Even before that point, they can have less visible problems like memory leaks, network latency, and disconnections. Liveness probes allow you to detect these problems, then terminate and restart the pod.

On the node level, you should set up Kubernetes in multiple availability zones (AZs) for high availability. When these risks are remediated, your system will be able to detect pod failures and failover nodes if there's an AZ failure.

These two reliability risks directly affect your deployment's ability to have the redundancy necessary to be resilient to pod, node, cluster, or AZ failure.

Missing liveness probes

A liveness probe is essentially a health check that periodically sends an HTTP request (or sends a command) to a container and waits for a response. If the response doesn't arrive, or the container returns a failure, the probe triggers a restart of the container.

FURTHER READING

Learn how to detect missing liveness probes and make sure they're defined: How to keep your Kubernetes Pods up and running with liveness probes

The power of liveness probes is in their ability to detect container failures and automatically restart failed containers. This recovery mechanism is built into Kubernetes itself without the need for a third-party tool. Service owners can define liveness probes as part of their deployment manifests, and their containers will always be deployed with liveness probes.

In theory, the only time a service owner should have to manually check their containers is if the liveness probe fails to restart a container (like the dreaded CrashLoopBackOff state). But in order to restart the container, a liveness probe has to be defined in the container's manifest.

No Availability Zone redundancy

By default, many Kubernetes cloud providers provision new clusters within a single Availability Zone (AZ). Because these AZs are isolated, one AZ can experience an incident or outage without affecting other AZs, creating redundancy—but only if your application is set up in multiple AZs.

If a cluster is set up in a single AZ and that AZ fails, the entire cluster will also fail along with any applications and services running on it. This is why the AWS Well-Architected Framework recommends having at least two redundant AZs for High Availability.

Kubernetes natively supports deploying across multiple AZs, both in its control plane (the systems responsible for running the cluster) and worker nodes (the systems responsible for running your application pods).

Setting up a cluster for AZ redundancy usually requires additional setup on the user's side and leads to higher cloud hosting costs, but for critical services, the benefits far outweigh the risk of an incident or outage.

FURTHER READING

Find out how to set up Availability Zone redundancy and scan for missing redundancy: How to deploy a multi-availability zone Kubernetes cluster for High Availability

Container deployment risks

If a container crashes, Kubernetes waits for a short delay and restarts the pod. Kubernetes will retry a few times before eventually giving up and giving the container a CrashLoopBackOff status. Similarly, when Kubernetes fails to pull the container image, it will retry for a few minutes until it gives up, then give the container a status of ImagePullBackOff.

There are also times when a pod simply can't be scheduled to run. Commonly, this happens because the cluster doesn't have the resources, or your pod requires a persistent volume that isn't available.

Containers in these states should be able to be restarted when a failure occurs, but are unable to, creating a risk to the resiliency of your deployment.

Pods in CrashLoopBackOff

CrashLoopBackOff is the state that a pod enters after repeatedly terminating due to an error. Normally, if a container crashes, Kubernetes waits for a short delay and restarts the pod.

FURTHER READING

Get tips for CrashLoopBackOff troubleshooting, detecting it, and verifying your fixes: How to fix and prevent CrashLoopBackOff events in Kubernetes

The time between when a pod crashes and when it restarts is called the delay. On each restart, Kubernetes exponentially increases the length of the delay, starting at 10 seconds, then 20 seconds, then 40 seconds, continuing in that pattern up to 5 minutes. If Kubernetes reaches the max delay time of 5 minutes and the pod still fails to run, Kubernetes will stop trying to deploy the pod and gives it the status CrashLoopBackOff.

CrashLoopBackOff can have several causes, including:

- Application errors that cause the process to crash.
- Problems connecting to third-party services or dependencies.
- Trying to allocate unavailable resources to the container, like ports that are already in use or more memory than what's available.
- A failed liveness probe.

There are many more reasons why a CrashLoopBackOff can happen, and this is why it's one of the most common issues that even experienced Kubernetes developers run into.

Images in ImagePullBackOff

Before Kubernetes can create a container, it first needs an image to use as the basis for the container. An image is a static, compressed folder containing all of the files and executable code needed to run the software embedded within the image.

Normally, Kubernetes downloads images as needed (i.e. when you deploy a manifest). Kubernetes uses the container specification to determine which image to use, where to retrieve it from, and which version to pull.

FURTHER READING

Learn about detecting and troubleshooting ImagePullBackOff, then verifying your fixes: How to fix and prevent ImagePullBackOff events in Kubernetes

If Kubernetes can't pull the image for any reason (such as an invalid image name, poor network connection, or trying to download from a private repository), it will retry after a set amount of time. Like a CrashLoopBackOff, it will exponentially increase the amount of time it waits before retrying, up to a maximum of 5 minutes. If it still can't pull the image after 5 minutes, it will stop trying and set the container's status to ImagePullBackOff.

Unschedulable pod errors

A pod is unschedulable when it′s been put into Kubernetes′ scheduling queue, but can′t be deployed to a node. This can be for a number of reasons, including:

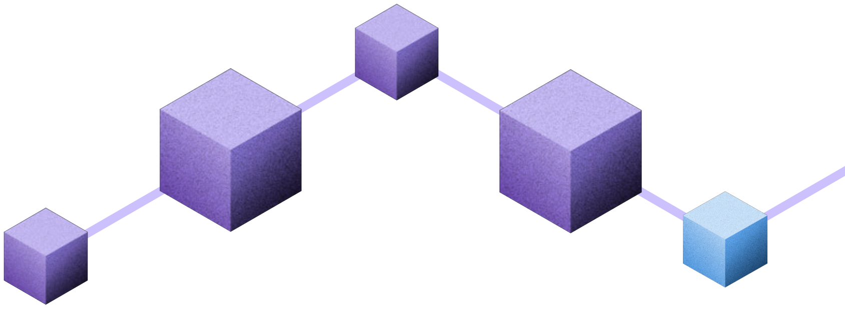
- The cluster not having enough CPU or RAM available to meet the pod′s requirements.
- Pod affinity or anti-affinity rules preventing it from being deployed to available nodes.
- Nodes being cordoned due to updates or restarts.
- The pod requires a persistent volume that′s unavailable, or bound to an unavailable node.

Although the reasons vary, an unschedulable pod is almost always a symptom of a larger problem. The pod itself may be fine, but the cluster isn′t operating the way it should, which makes resolving the issue even more critical.

Unfortunately, there is no easy direct way to query for unschedulable pods. Pods waiting to be scheduled are held in the ″Pending″ status, but if the pod can′t be scheduled, it will remain in this state. However, pods that are being deployed normally are also marked as ″Pending.″ The difference comes down to how long a pod remains in ″Pending.″

FURTHER READING

Find out how to detect and resolve unschedulable pod issues: How to troubleshoot unschedulable Pods in Kubernetes



Application risks

Whenever you update your application, there are hidden reliability risks. Updates typically roll out gradually, not all at once. What happens if your team releases another update before the first rollout finishes? What happens if you push a release while Kubernetes is upgrading itself? You might end up with two different versions running side-by-side.

Another common application risk is introduced by using init containers. These are handy for preparing an environment for the main container, but introduce a potential point of failure where the init container can't run and causes the main container to fail.

Both of these risks occur at the application level, which means infrastructure or cluster-level detection could miss them.

Application version non-uniformity

Version uniformity refers to the image version used when declaring pods. When you define a pod or deployment in a Kubernetes manifest, you can specify which version of the container image to use in one of two ways:

FURTHER READING

Learn more about version non-uniformity and how to resolve it: How to ensure consistent Kubernetes container versions

- **Tags,** which are created by the image's creator to identify a single version of a container. Multiple container versions can have the same tag, meaning a single tag could refer to multiple different container versions over time.
- **Digests,** which are the result of running the image through a hashing function (usually SHA256). Each digest identifies one single version of a container. Changing the container in any way also changes the digest.

Tags are easier to read than digests, but they come with a catch: a single tag could refer to multiple image versions. The most infamous example is `latest`, which always points to the most recently released version of a container image. If you deploy a pod using the `latest` tag today, then deploy another pod tomorrow, you could end up with two completely different versions of the same pod running side-by-side.

Init container errors

An init container is a container that runs before the main container in a pod. They're often used to prepare the environment so the main container has everything it needs to run.

For example, imagine you want to deploy a large language model (LLM) in a pod. LLMs require datasets that can be several GB. You can create an init container that downloads these datasets to the node so that when the LLM container starts, it immediately has access to the data it needs.

Init containers are incredibly useful for setting up a pod before handing it off to the main container, but they introduce an additional point of failure.

Init containers run during the pod's initialization process and must finish running before the main container starts. To add to this, if you have multiple init containers defined, they'll all run sequentially until they've either completed successfully or failed.

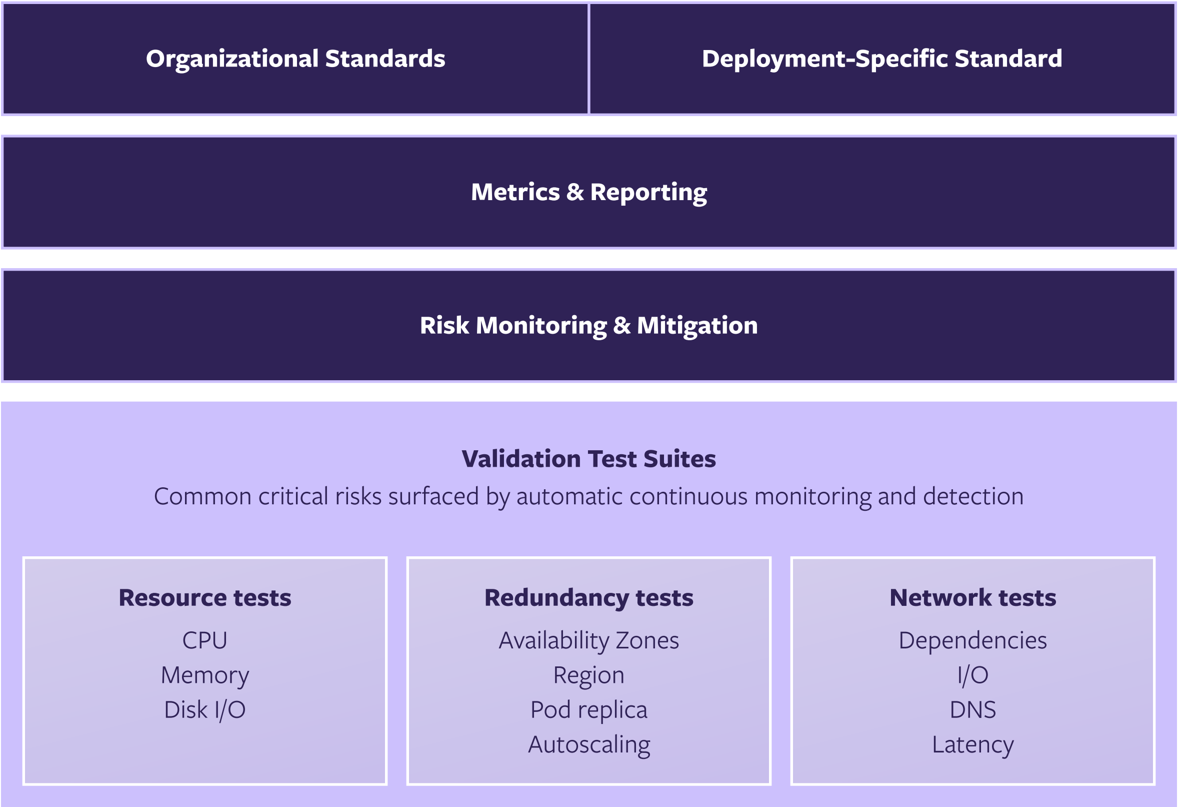
If an init container fails and the pod's restartPolicy is not set to Never, the pod will repeatedly restart until it succeeds. Otherwise, Kubernetes marks the entire pod as failed with the status Init:CrashLoopBackOff.

FURTHER READING

Find out more about init container errors, how to detect them, and how to troubleshoot them: How to fix Kubernetes init container errors

5. Validating resilience through testing

Framework for Kubernetes Resiliency



Organizational Standards

Deployment-Specific Standard

Metrics & Reporting

Risk Monitoring & Mitigation

Validation Test Suites

Common critical risks surfaced by automatic continuous monitoring and detection

Resource tests

CPU
Memory
Disk I/O

Redundancy tests

Availability Zones
Region
Pod replica
Autoscaling

Network tests

Dependencies
I/O
DNS
Latency

This is done by using Fault Injection testing. Fault Injection works by creating controlled failure in a computing component, such as a host, container, or service. By observing how their components respond to failure, engineering teams can take action to make their services more resilient. It can be used in experiments to uncover new reliability risks and failure modes, or it can be used in standardized groups, known as test suites, to validate workload behavior.

When to test in your SDLC and which exact tests to run will vary depending on your individual organization's standards and the maturity of your resilience practice. But there is a core set of resiliency tests that should be run for every Kubernetes deployment, as well as best practices to help determine when in your SDLC your teams should run resiliency tests.

Exploratory testing

Exploratory testing is used to better understand your systems and suss out the unknowns in how it responds to external pressures. Many of the experiments performed under the practice of Chaos Engineering make use of exploratory tests to find unknown points of potential failure.

To minimize the impact on your systems, exploratory tests should always be done in a controlled manner. While a trustworthy Fault Injection tool will contain safeguards like automatic rollback in case of problems, the injection of faults can potentially cause disruption when doing exploratory tests. Be sure to follow Chaos Engineering best practices like limiting the blast radius and carefully defining the boundaries of the experiment. Ideally, these tests should start with individual services, then expand broader into the organization as you become more confident in the results and impact of the test.

For example, a common type of exploratory test is making sure your Kubernetes deployments scale properly in response to high demand. You can set up a Horizontal Pod Autoscaling (HPA) rule on your deployment to increase the number of pods when CPU usage exceeds a certain percentage. Then, you can use a Fault Injection tool to apply CPU pressure directly to the deployment, while monitoring the number of pods.

If Kubernetes deploys an additional pod, then you know your system will scale properly under similar conditions in production. If not, tweak your HPA rules and repeat the test until the system behaves the way you expect. Then those HPA rules can become part of your resilience standards, and future tests will be used to validate that the rules are in place.

Further reading

Chaos Engineering has come a long way over the years. Use these resources to find out more about the history of Chaos Engineering, its uses, and how to get certified so you can start designing your own exploratory testing experiments:

- Chaos Engineering: the history, principles, and practice
- What is Chaos Engineering?
- Gremlin Enterprise Chaos Engineering certification

Validation testing

Once you have a standardized set of known failures and reliability risks, you can test your resilience to them with validation testing. Using Fault Injection, validation tests inject specific failure conditions into your systems to verify resilience to failures. Unlike exploratory testing, which is done manually, validation testing works best when it can be automated on a schedule. Ideally, they should be tested weekly, but many organizations will start with monthly testing, then gradually increase the frequency as they become more comfortable with the testing process.

Based on the standards defined earlier in the framework, you should have a list of known failures and reliability risks. By collecting validation tests that correspond to these failures, you can create a standardized set of test suites that can be run across your organization. As discussed above, the results of these testing suites can be collected over time to create metrics, then charted to create dashboards that can be used to align and prioritize reliability risk remediation efforts.

Standardized test suites for every Kubernetes deployment

There are certain resiliency tests that should be run for every Kubernetes deployment. Based on the key traits in common with any Kubernetes cluster, these should form the core of your resiliency test standards. These core tests fall under three groups.

Resource tests

Any Kubernetes deployment needs to be resilient to sudden spikes in traffic, demand, or resource needs. These two tests will verify that your services are resilient to sudden resource spikes. Depending on your architecture, you may also want to add a Disk I/O scalability test to this mix.

- **CPU Scalability:** Test that your service scales as expected when CPU capacity is limited. This should be done in three stages of 50%, 75%, and 90% CPU consumption.

  Estimated test length: 15 minutes

- **Memory Scalability:** Test that your service scales as expected when memory is limited. Memory consumption should be done in three stages: 50%, 75%, and 90% capacity.

  Estimated test length: 15 minutes

Redundancy tests

Make sure that your deployments are resilient to infrastructure failures. These tests shut down a host or access to an availability zone to verify that your deployment has the redundancy in place to stay up when a host or zone goes down. If your standards call for multi-region redundancy, then you should add tests that make regions unavailable.

- **Host Redundancy:** Test resilience to host failures by immediately shutting down a randomly selected host or container.

  Estimated test length: 5 minutes

- **Zone Redundancy:** Test your service's availability when a randomly selected zone is unreachable from the other zones.

  Estimated test length: 5 minutes

Dependency and network tests

The microservices nature of Kubernetes architectures can create a web of dependencies. These tests help you verify that your deployments will respond correctly when dependencies have failed, network issues are delaying communications, or have expiring certificates that make them unavailable. If you have a more complex architecture, you may want to periodically run dependency discovery tests to uncover any unknown dependencies.

- **Dependency Failure:** Test your service's ability to tolerate unavailable dependencies by dropping all network traffic to a specific dependency.

  Estimated test length: 5 minutes

- **Dependency Latency:** Test your service's ability to tolerate slow dependencies by delaying all network traffic to this dependency by 100ms.

  Estimated test length: 5 minutes

- **Certificate Expiry:** Test your service's dependencies for expired or expiring TLS certificates by opening a secure connection to your dependency, retrieving the certificate chain, and validating that no certificates expire in the next 30 days. A lack of a secure connection would also pass the test, since that would mean there are no certificates.

Estimated test length: 1 minute

Customizing suites to fit your organization

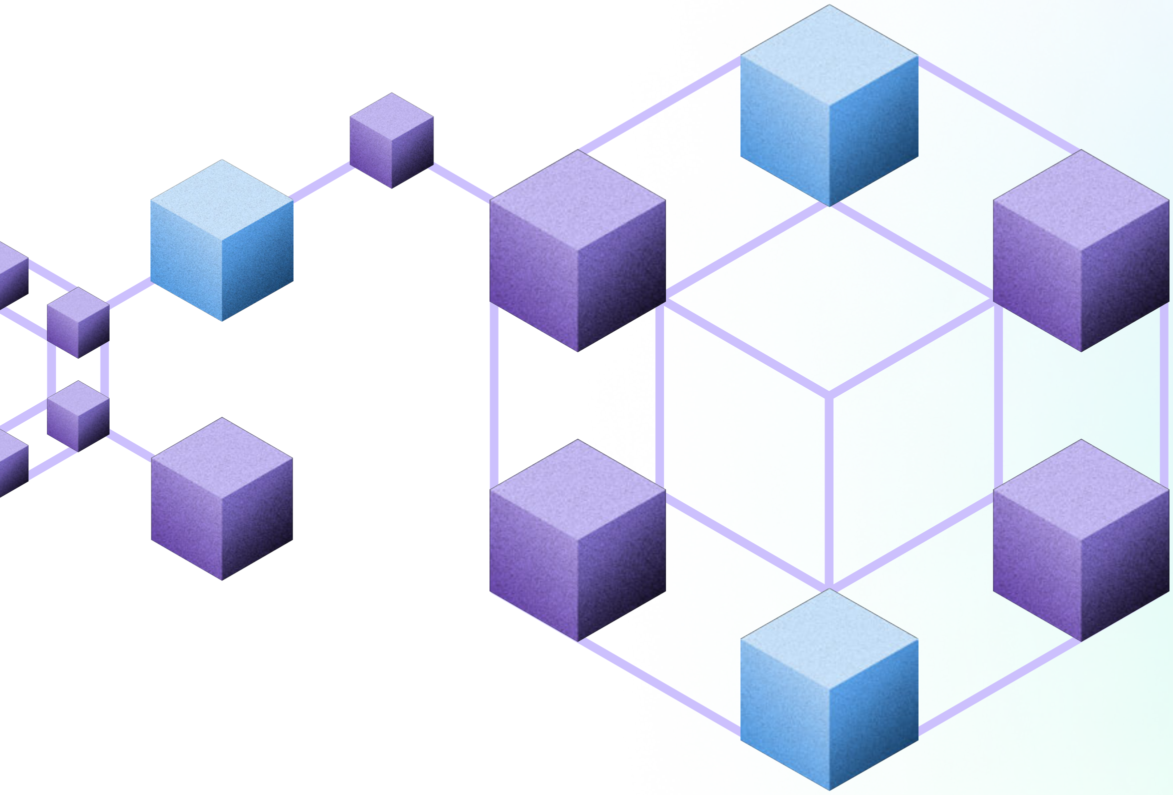
While you should start with the standardized test suites above, there are situations where you should make adjustments to better fit your organization and its reliability goals. These changes could be adding new tests designed to fit specific failures, or tweaking the parameters of existing tests, such as adjusting the allowed latency depending on a service.

When customizing suites, you should do it based off data from sources like:

- **Incidents** - When there's an outage, it's a good practice to set up tests to detect and prevent the same incident from happening in the future. For example, if you experienced a DNS-based outage, then you may want to set up weekly tests to make sure you can failover to a fallback DNS service.
- **Observability alerts** - There's plenty of application behavior that doesn't directly create an outage, but is still a definite warning sign. Perhaps a service owner has noticed that compute resource spikes that take up 85% of compute capacity don't take the system down, but still create a situation where a spike in traffic would cause an outage. In this case, you'd want to add tests that simulate compute resource usage at 85% capacity to ensure resilience to this potential failure.
- **Exploratory testing** - As covered above, it's important to work directly with service operators to adjust testing parameters and fit the needs of specific services. Using exploratory testing, operators can determine exactly what the failures are so you can design tests against them. Critical services, for example, should have higher resilience standards than internal services, and the test suite should be customized to fit these standards.

Industry models - There are many architecture models, such as the AWS Well-Architected Framework, that have specific best practices to improve reliability. If you're using these architecture standards, then you can adjust your testing suites to verify compliance with those standards.

- **Industry compliance requirements** - Highly regulated industries, such as finance, can often have resilience and reliability standards unique to their industry. Often these can be much more strict than common best practices, and you should adjust test suites accordingly to fit these compliance requirements.



6. Testing in different stages of your SDLC

The goal of validation testing is to provide an accurate picture of your current system's resiliency. As such, testing should, when possible, be done in production environments. However, resiliency testing with Fault Injection is not without its risks. These can be mitigated with the right tool, setup, and experience with testing, but for this reason, many organizations earlier in their resilience testing journey may wish to build more confidence by testing in staging first. Additionally, there are benefits to testing in pre-production stages to catch risks before release.

Tradeoffs for Each Automation Strategy

STRATEGY	PROS	CONS	
Gating Release Candidates on running tests	› Catches some resilience risks before production › Fits into existing QA/performance testing cycles	› Expensive and difficult to run production-like test environments › Can miss infrastructure and network-level risks	› Can lead to false confidence without also testing in production › Slows down QA process
Running tests after production deployments	› Runs in production, so it can capture software, infrastructure, and network risks › Fast cycle time between deployment and risk identification	› Slows down deployment process if deploying faster than test cycle length › Requires strong monitoring	› Misses risks introduced through out-of-band infrastructure and network-level changes › Some risks will make it to production but should be mitigated quickly
Scheduling tests at regular intervals	› Runs in production, so it can capture software, infrastructure, and network risks › Decoupled from software release cycle; no impact on time to deployment	› Requires strong monitoring › Some risks will make it to production but should be mitigated quickly	

Ultimately, the choice comes down to your individual organization and its familiarity with resilience testing. Consider the pros and cons of each choice before you decide on a strategy.

Testing in staging

Testing in a staging environment prevents any potential downtime caused by testing from impacting customers. However, perfectly duplicating a staging environment with the same workloads, resources, and traffic as production environments is cost-prohibitive and time intensive. Additionally, there are changes outside your control, such as network topography, that can't be accounted for in staging environments.

Ultimately, while testing in staging can catch key reliability risks, it can't give you an accurate view of the reliability of your system in production.

Testing as part of release automation

Like other kinds of testing, validation resilient testing can be done as part of a release pipeline process, such as CI/CD. But the best choice for your organization will depend on your release schedule.

Due to the nature of Fault Injection testing, a full battery of tests could take several hours. If you're releasing on a weekly or monthly schedule, holding up a deployment to run these tests could be worth it for the reliability risks you uncover. However, if you're set up for multiple releases a day, then the time spent on the tests prevents them from being used as a gating mechanism. In this case, you should consider testing in production, either post-deployment or on a regular schedule.

Remember, the goal of resiliency testing is to uncover reliability risks in production. While some of these can be uncovered before deployment, you should fit testing into your SDLC where it makes the most sense and can be the most effective at uncovering reliability risks in production before they impact customers.

Automating on a schedule

Kubernetes systems are constantly changing with new deployments, resource changes, network topography shifts, and more. A service that had very few reliability risks two weeks ago could suddenly have a much more vulnerable reliability posture due to new releases, changes in dependency services, or network shifts.

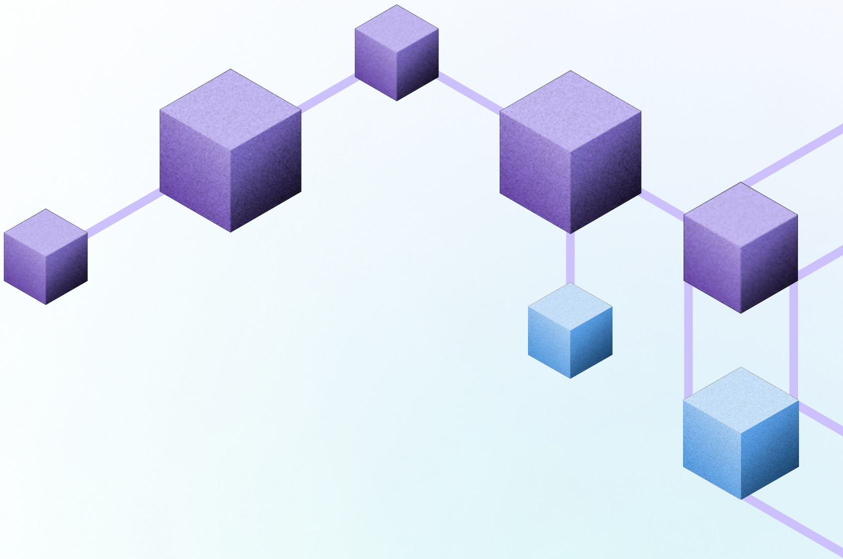
The only way to catch these changes is through regular, automated validation testing using test suites. Ideally, you should aim to have weekly scheduled tests in production, though many organizations work up to this point.

It's best to schedule these tests during a time when engineers are present and available to address any issues. You should also schedule them to run shortly before your prioritization and resourcing meetings. This will allow your teams to move quickly to address any critical reliability risks the tests uncover.

FURTHER READING

Find out more about the questions you should be asking and the trade-offs you should be weighing regarding testing across your SDLC with these resources from Gremlin:

- The two kinds of failure testing
- The case for Fault Injection testing in Production
- Fault Injection in your release automation

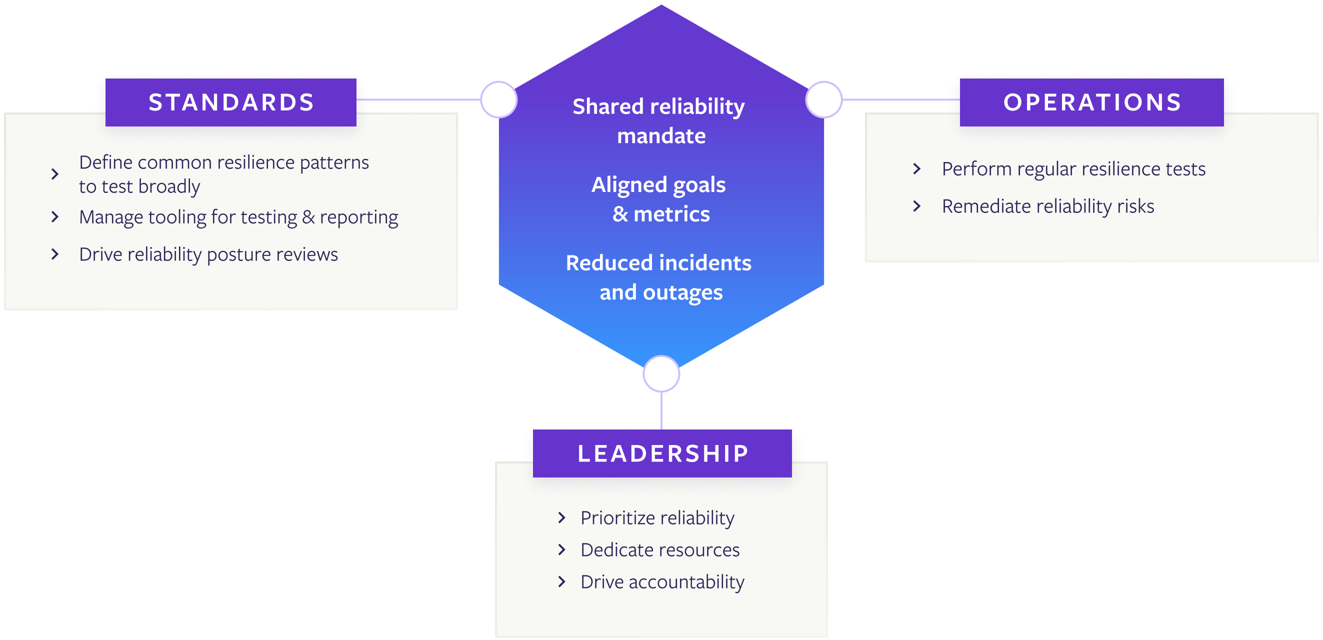


7. Roles and responsibilities

Any Kubernetes resilience effort requires contributions from three key roles if it's going to be successful. Everyone working on resiliency falls into one of these three roles, which are sorted by their responsibilities within the framework:

1. Leadership roles create prioritization and allocate resources to resiliency management.
2. Standards roles set standards, manage tooling, and oversee the execution of the framework.
3. Operations roles perform tests on services and remediate reliability risks.

Resiliency Roles



Shared reliability mandate

Aligned goals & metrics

Reduced incidents and outages

STANDARDS

- Define common resilience patterns to test broadly
- Manage tooling for testing & reporting
- Drive reliability posture reviews

OPERATIONS

- Perform regular resilience tests
- Remediate reliability risks

LEADERSHIP

- Prioritize reliability
- Dedicate resources
- Drive accountability

The roles aren't tied to specific titles, and it's common for one person or team to take on two of the roles: for example, performance engineering teams or centralized SRE teams often take on both setting the standards and performing tests and mitigations—at least initially. But without someone stepping in to take on the requirements of each role, teams often struggle to make progress.

Leadership role

The leadership role is the one responsible for setting the priorities of engineering teams and allocating resources. In some companies this is held by someone in the C-suite, while in others it's held by Vice Presidents or Directors. The defining factor is that anyone in this role has the authority to make organization-wide priorities and direct resources towards them.

Core Responsibilities:

> **Dedicate resources to reliability**
>
> Most reliability efforts fail due to a lack of prioritization from the organization. For your resilience practice to be effective, leadership roles need to allocate resources to it.

> **Ensure standards create business value**
>
> Work with those in the standards roles to make sure resiliency standards and goals tie directly back to business value. Try to find the balance where the time, money, and effort spent finding and mitigating reliability risks is creating far more value than it takes in resources.

> **Drive accountability and review metrics dashboards**
>
> When leadership is visibly engaged in reviewing reliability metrics, it lends importance to the efforts, which, in turn drives action. Leadership should hold operators accountable for improving resiliency—and applaud them when they do.

Standards role

The Standards role is responsible for driving the Kubernetes resilience efforts across the organization. They own the standards, tooling, and organizational processes for executing the framework. In some organizations, this role is in centers of excellence, such as SRE or Platform Engineering teams, while in others this role is added to an existing role like Kubernetes architects.

Core Responsibilities:

› Define reliability standards

Reliability standards should be based on a combination of universal best practices, organizational reliability goals, and unique deployment reliability risks. These should be consistent across the organization, with any service-specific deviations (such as those discovered through exploratory testing) documented.

› Manage tooling for testing & reporting

By centralizing testing and reporting tooling with the standards role, tests can be automated to minimize the lift by individual teams and metrics can be compiled to make it easier to align around reliability and prioritize fixes.

› Determine standardized validation test suites

Reliability test suites are a powerful tool for creating a baseline of resiliency across your organization. The standards role should define these, then integrate them into tooling so teams can automate running them.

› Owning operationalization processes

Metrics should be regularly reported and reviewed in meetings where the reliability posture is reviewed, then any fixes are prioritized. Whether these are standalone meetings or integrated into existing meetings, the standards role should own and run these review processes.

Operator role

This role could have a wide variety of titles, but the defining characteristics are that they're responsible for the resiliency of specific services. They make sure the tests are run, report the results, and make sure any prioritized risks are addressed.

Core Responsibilities:

Run tests and report on results

Once the initial agents or setup is done, testing should be automated to make this a lighter lift. As part of the prioritization and review meetings, operators will need to make sure the test results are reported and speak to any discussion about them.

Respond to risks detected by monitoring

Risks detected by monitoring can often be fixed with a change to the configuration or other lighter-lift fixes. In these cases, operators should be empowered to quickly address these risks to maintain Kubernetes resiliency.

Address and mitigate reliability risks

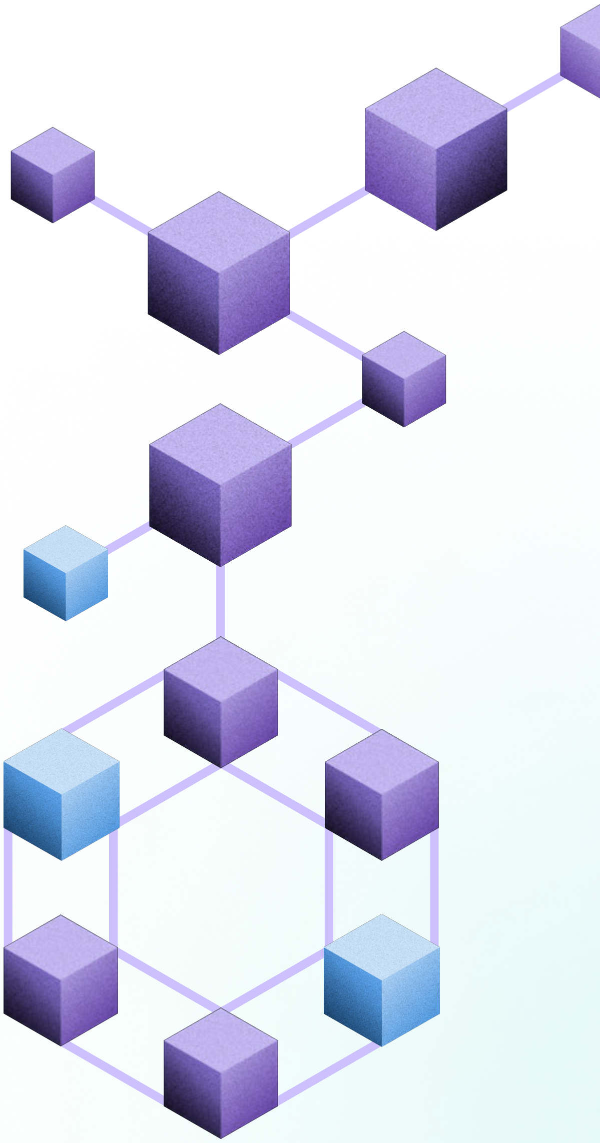
Once reliability risks have been prioritized, operators are responsible for making sure the risk is fixed. They may not be the person to perform the actual work, but they should be responsible for making sure any risks are addressed, then testing again to verify the fixes.

WHAT MAKES A BEST-IN-CLASS RELIABILITY PRACTICE?

A best-in-class reliability practice extends across teams to improve the resiliency and availability of your systems. At the same time, it enables engineering teams to spend less time fighting fires and resolving incidents so they can focus on vital work like new features or innovations.

Gremlin has worked with reliability program leaders at Fortune 100 companies to identify the traits of successful programs. Reliability programs built around these four pillars and 18 traits align organizations, get crucial buy-in, and achieve real, measurable improvements to the reliability of their systems.

Find out what it takes to build an effective reliability effort with the How to Build a Best-in-Class Reliability Program checklist.



Next steps: Your first 30 days of resiliency

Your Kubernetes resiliency management practice will mature, grow, and change over time, but it doesn't have to take months to start creating results. In fact, you can start uncovering reliability risks and having a demonstrable impact on your Kubernetes reliability with this roadmap for your first 30 days.

1. Choose the services for your proof of concept

Leadership and teams can be hesitant to roll out new programs across the entire organization, and understandably so. Many resiliency efforts start with a few services to show its efficacy before it can be more widely adopted.

You can speed the process along by getting alignment on a specific group of services being used for the pilot. These are your early-adoptor services, and the more you have everyone involved on board, the better results you're going to get.

In fact, having a small group of teams who are invested and focused can often be more effective than trying with a wider, more hesitant group right out of the gate. Once you start proving results with early adopters, then you'll get less resistance as you roll the program out more broadly.

When choosing these services, you should start with ones that are important to your business to provide the greatest immediate value. Dependencies are a common source of reliability risks, so a good choice is to start with central services that have fully-connected dependencies. You could also select services that are fully loaded with production data and dependencies, but aren't launched yet, such as services during a migration or about to be launched. The last common choice is services that are already having reliability issues, thus allowing you to prove your effectiveness and address an area of concern at the same time.

2. Set up your systems for risk monitoring and testing

Fault Injection requires a tool to be integrated into your Kubernetes deployment. If you're building your own tool, this can be pretty complicated, but a reliability platform like Gremlin streamlines the process of installing agents and setting up permissions.

Once the agent is set up, you'll want to define your risk monitoring parameters and core validation test suites. A good place to start is with the critical risks from Chapter 4 and the test suites from Chapter 5. (Gremlin has these set up as default test suites for any service.)

Generally, these core risks and test suites are a good place to start, then you can adjust them as you become more comfortable with testing. But you can also alter these test suites to fit unique standards for your organization or to include a test that validates resilience to specific issues, such as ones that recently caused an outage.

3. Use risk monitoring to detect Kubernetes risks

Almost every Kubernetes system has at least one of the critical risks above. Since risk monitoring uses continuous detection and scanning rather than active Fault Injection testing, it's a faster, easier way to uncover active critical reliability risks.

Follow these steps to quickly find risks, fix them, and prove the results:

1. Install the agent or tool in your Kubernetes cluster.
2. The scan will return a list of reliability risks, along with a mitigation status.

3. Work with the team behind the service to address unmitigated risks. Most of these, such as missing memory limits, can be a relatively light lift to fix.

4. Deploy the fix and go back to the monitoring dashboard. Any risk you addressed should be shown as mitigated.

5. Congratulations! You’ve made your Kubernetes deployment more reliable.

Since risk monitoring is automated and non-invasive, this is also an easier way to spur adoption of resiliency management with other teams. Show those teams the results you were able to create, then help them to set up their own risk monitoring.

4. Use validation testing for a baseline reliability posture report

Now that you’ve addressed some of the more pressing reliability risks, it’s time to start running Fault Injection tests. Run the validation test suites you set up to get a baseline report for the reliability posture of your early-adopter services.

These first results will usually return a lot of existing reliability risks, which can be a good thing. It means your resiliency testing is effectively uncovering reliability risks before they cause outages.

Now that you’ve addressed some of the more pressing reliability risks, it’s time to start running Fault Injection tests. Run the validation test suites you set up to get a baseline report for the reliability posture of your early-adopter services.

These first results will usually return a lot of existing reliability risks, which can be a good thing. It means your resiliency testing is effectively uncovering reliability risks before they cause outages.

5. Address high-priority risks, then verify your fixes

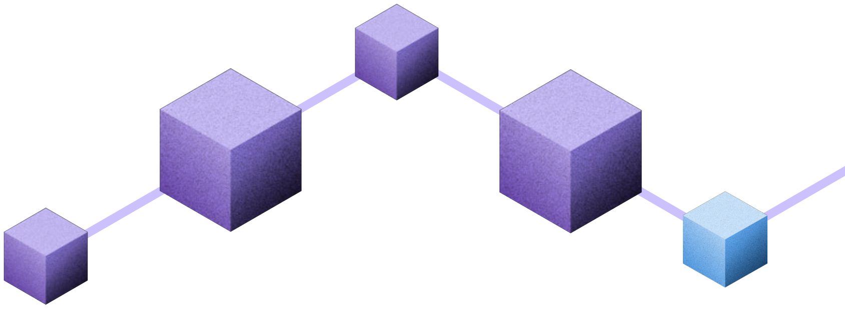
After the operators have had a chance to address the issues, run the same test suites again to see if the fixes were successful. Once you've verified the fixes, gather up the results and review them with the rest of your team.

You should have a list of critical Kubernetes risks that you've addressed, and by looking at the before and after results from risk monitoring and validation tests, you'll be able to show the effectiveness of your resiliency efforts—and show exactly how you've improved the reliability of your Kubernetes deployment.

Do it all with a 30-day trial from Gremlin

Gremlin offers a free trial that includes all of the capabilities you need to take the actions above. Over the course of four weeks, you'll be able to stretch your resiliency wings, prove the effectiveness of your efforts, and have a lasting impact on the reliability of your Kubernetes deployment.

Start your free 30-day Gremlin trial



Gremlin

Gremlin is the Enterprise Reliability Platform that helps teams proactively test their systems, build and enforce reliability and resiliency standards, and automate their reliability practices organization-wide.

Hundreds of enterprise finance, retail, and technology companies around the world trust Gremlin with the reliability of their systems.

Learn more at gremlin.com.