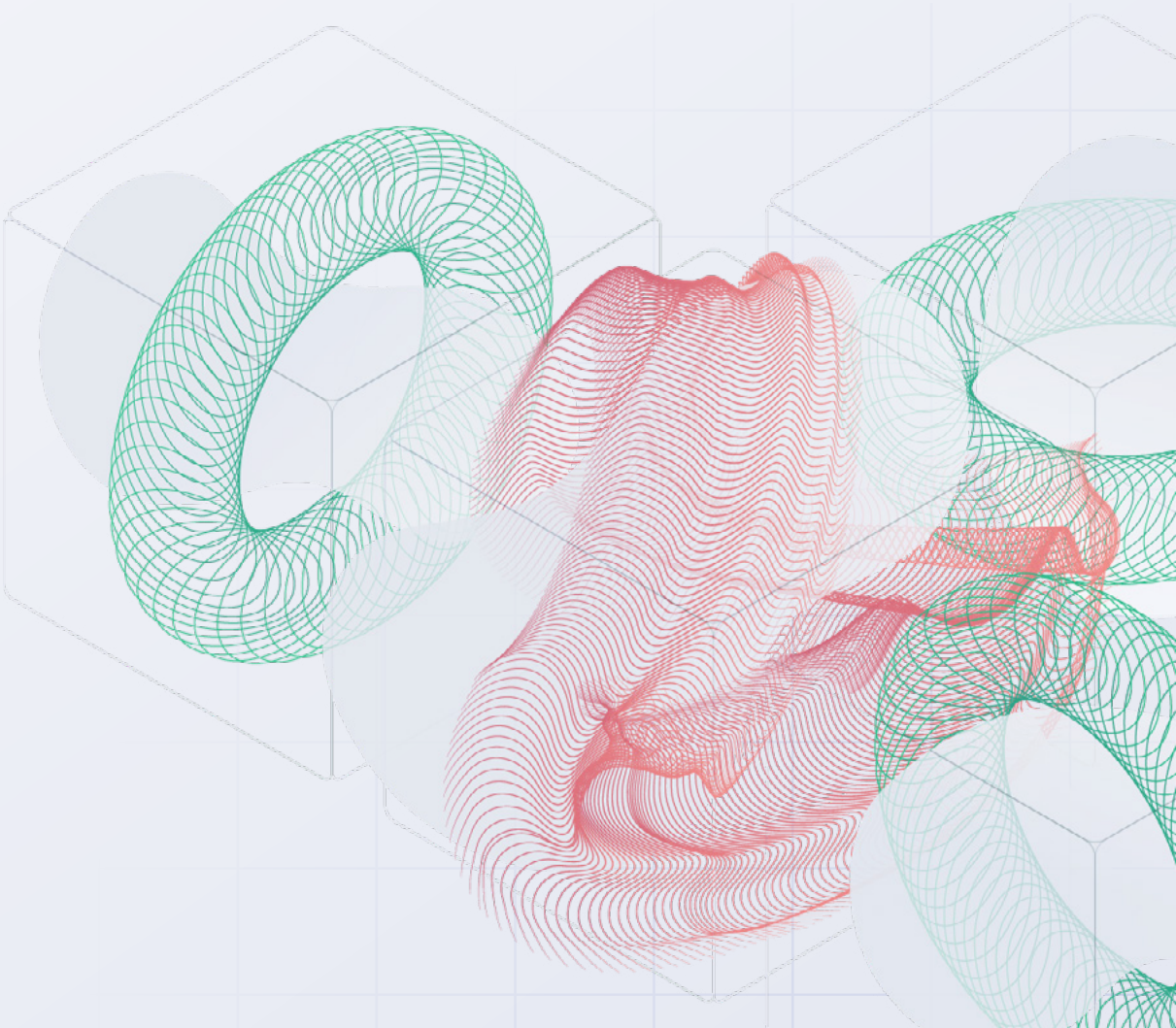
WHITE PAPER

Improving Reliability During Kubernetes Migrations

HOW RELIABILITY IS KEY TO A SUCCESSFUL KUBERNETES MIGRATION



Gremlin

Table of Contents

Executive Summary

Adopting Kubernetes has many benefits—optimized infrastructure costs, more efficient computing resource utilization, faster development cycles, and extensive automation to name a few—but migrating an application from an existing or legacy environment to Kubernetes is no small feat, especially for engineers who are new to cloud-native concepts like containers and microservices.

Migrating from a legacy architecture to Kubernetes requires fundamental changes to how applications are designed, developed, and deployed. Engineers need to learn an entirely new architecture and platform, build and manage distributed clusters, and work with new tools like the Kubernetes Dashboard and command-line tool. During this process, unexpected reliability problems will emerge such as container crash loops, application slowdown caused by network latency, and misconfigured deployments. For teams caught unprepared, a migration can quickly become expensive, time-consuming, and high-risk. The question is: how do we prepare for a successful migration to Kubernetes when there are so many unknown variables?

An initiative of this size and complexity requires a proactive and considerate approach to reliability involving:

- **Careful research and planning on the new architecture.**
- **Proactively identifying and considering potential risks and failure modes.**
- **Testing for these and other failure modes throughout the migration process and developing processes for addressing failure modes.**
- **Continuously building on your reliability practice during and after migration.**

In this whitepaper, we explore the reliability challenges of migrating an application from a legacy architecture to Kubernetes, and how to identify, address, and prevent these issues throughout the migration process. These steps will help you address faults before they can become production outages, help your engineers better understand how Kubernetes works, and result in a more successful migration and a more resilient system.

Introduction

To say Kubernetes has taken the software world by storm would be an understatement. The Cloud Native Computing Foundation—the organization that manages the Kubernetes project—found that 96% of organizations worldwide are either using or evaluating Kubernetes along with 5.6 million developers. According to the Flexera 2022 State of the Cloud Report, 96% of organizations are using public cloud platforms. This puts Kubernetes adoption on par with public cloud adoption.

The percentage of organizations using Kubernetes is equal to the percentage of organizations using public cloud platforms.

This is especially impressive as Kubernetes requires an entirely new approach to software deployment and maintenance than what most developers were used to. Kubernetes is built around the concept of microservices and short-lived, ephemeral units called containers, rather than long-lived binaries running on dedicated servers or virtual machines. It also introduces several additional layers of abstraction and complexity, which adds administrative overhead to certain tasks and requires developers and SREs to think multiple layers deep when interacting with their applications. The benefit is that you get a system that's extremely flexible, highly scalable, includes self-healing mechanisms, and has an enormous ecosystem of tools and providers.

Of course, major technical changes don't come for free. Moving to a new architecture comes with risks, especially to the resilience of your applications and infrastructure. We'll explore these risks in the next section.

A new paradigm: From monoliths to microservices

The biggest leap teams need to make when migrating from monoliths to Kubernetes is understanding the concept of microservices. A microservice is a lightweight, independently manageable unit of functionality that combines with other microservices to form a complete application. By design, microservices are:

- Short-lived, lasting anywhere from several days to just a few seconds.
- Able to rapidly start and stop.
- Deployable across multiple hosts simultaneously.
- Easily replicated and recreated, either by an administrator or using automated tools.

A microservice is a lightweight, independently manageable unit of functionality that combines with other microservices to form a complete application.

Compare this to a monolith, which is typically one large executable package containing everything needed to run an entire application. Monolithic applications are typically long-lasting, slow to start, difficult to scale, and time consuming to redeploy or rebuild.

The challenge of migrating to microservices

At a high level, migrating a monolith to a microservice involves breaking up the monolith into its smallest individual components, making those services independently deployable, and connecting those services via API calls. Each service performs a discrete function and can be managed independently of the others, letting you modify them without directly impacting other teams.

As an example, imagine you're a developer working for a major bank. The bank's entire backend application is a monolith, which multiple development teams contribute to. Each team is responsible for a different function; for example, there's the account management team, the ledger team, and the security team. If any one of these teams makes a change to their part of the application, the entire application must be rebuilt, retested, and redeployed to production. Your organization might use techniques like merge windows and code freezes to make this process easier, but it's still very cumbersome and time-consuming, and can delay important changes by days or even weeks.

Now, let's imagine this application in a microservice architecture. Each team now maintains their own independently operable service that provides a discrete unit of functionality. Services communicate with each other using standard API calls, while Kubernetes tracks the deployment status, network address, and health of each service. Now, if any one team needs to push an important update, they can simply update their service without impacting other services. Changes can go live immediately after validation, and individual teams effectively become the owners of their services.

Architecture	Key benefits	Key drawbacks
Monolith	• Easy to understand • Easy to deploy • Centralized code	• Slower development cycles • Slower deployments due to larger packages
Microservice	• Flexible • Supports faster development cycles • Provides teams with more independence and autonomy	• More complicated to deploy and manage • High risk of emergent failures

What are containers, and how do they relate to microservices?

Microservices are a concept, and containers are the technology most commonly used to implement them. Containers are software packages that bundle code along with everything needed to run that code, such as files and critical dependencies. While container technology has been around since 1979, it went mainstream with Docker in 2013. Docker—and other container management tools—made it easier for developers to configure, build, and run containers, kickstarting the growth of microservices-based applications.

In 2014, Google released Kubernetes as an open source project. Kubernetes is a container orchestrator, which means it builds on tools like Docker by handling the distribution of containers in a clustered environment. Kubernetes lets engineers set up complex networking rules between containers, add redundancy and fault tolerance, set up automatic container monitoring and health checks, and much more, with relatively little effort.

Kubernetes high-level concepts

As a container orchestrator, Kubernetes manages the deployment, configuration, and monitoring of containers across one or more hosts. Instead of managing individual containers directly, Kubernetes uses the concept of Pods, where one or

more containers are bundled together into a single unit with its own process space, hardware resource pool, and IP address. Pods provide greater control than containers by letting us share resources between containers directly, define container restart policies, and more. For example, if we want to add a logging component to our application, we can deploy a dedicated logging container alongside our application container within the same Pod. Now, whenever we deploy this Pod, both our application container and logging container deploy together.

The real power of Kubernetes comes in the form of ReplicaSets and Deployments. ReplicaSets ensure that a Pod has a certain number of copies (or replicas) running at any given time. If a Pod belonging to a ReplicaSet fails, Kubernetes automatically deploys a new Pod to replace it. Deployments expand ReplicaSets by adding features such as rolling updates, canary deployments, and rolling back changes. Lastly, Kubernetes is declarative, meaning we define the desired state of our cluster and Kubernetes performs the actions necessary to achieve and maintain that state. This state is most commonly represented using YAML files called manifests, but can also be applied directly using the kubectl command line tool.

Putting all of this together, deploying an application in Kubernetes involves:

01 **Packaging the application's code, dependencies, and configuration into a container.**

02 **Creating a manifest for a Deployment where we define our Pod and deployment parameters.**

03 **Applying the manifest to our cluster using kubectl, the Kubernetes Dashboard, or another tool like Helm.**

Let's go back to the bank example. Your team is actively working on migrating the bank's mobile deposit feature to Kubernetes. After rewriting the code to communicate with other services via API calls, you compile it into a container along with all of its software dependencies. A separate application provides image recognition functionality for scanning checks, so you compile that application into a second container image. You then create a YAML file where you define a Pod containing both the application container and image recognition container, so that both are deployed simultaneously. You also define a Deployment, which specifies:

- What network ports should be exposed.
- What security permissions the service requires.
- How much computing capacity (CPU, RAM, and disk space) you need.
- How many replicas should be created in case one instance fails.
- How to run health checks against the service, and how to recover if a check fails.
- How to deploy updates to the service while maintaining availability.

These are the basics that you need to know in order to understand the rest of this guide. There is much more to Kubernetes such as Services, Ingress Controllers, and Volumes. We recommend reading the Kubernetes documentation to learn more about the different Kubernetes object types.

Reliability concerns when migrating from monoliths to microservices

Before we can discuss reliability issues specific to Kubernetes, we need to address microservice reliability as a whole. Microservices are fundamentally different from monoliths, and while there are common reliability concerns shared between them, there are many that are unique to microservices due to their added complexity. These include:

- Treating servers as disposable and interchangeable.
- Managing containers that are ephemeral.
- Managing data persistence in a constantly changing environment.
- Moving from local communication to network-based communication.
- Educating and training teams on developing microservices.

Treating servers as disposable and interchangeable

It's not uncommon for system administrators to provision, configure, and maintain dedicated servers by hand. The practice of logging into a remote system using a tool like SSH and deploying a patch or restarting services is often the easiest, but it results in systems that are highly customized, difficult to recreate, undocumented, and extremely costly to the business when they fail.

Meanwhile, cloud computing platforms like AWS, Azure, and GCP popularized the idea of systems that are temporary and interchangeable by design. This shifted the paradigm from having a small set of distinct, hand-crafted systems (i.e. "pets") to having a fleet of interchangeable systems managed through automation (i.e. "cattle"). If a cloud server fails and needs replacing, engineers don't need to dedicate significant amounts of time and resources towards fixing it when they can simply replace it with a new instance. Managed Kubernetes services like Amazon EKS, Azure Kubernetes Service, and Google Kubernetes Engine take this further by completely abstracting away infrastructure management and automatically provisioning nodes for us as we deploy our applications to Kubernetes.

This means that we can focus exclusively on managing our Kubernetes workloads, but it also reduces our visibility into how our systems are operating. If one of our Kubernetes nodes fails, we have to rely on our cloud provider to detect the failure and spin up a new node, and we have to rely on Kubernetes itself to reschedule our workloads onto a healthy node. If not, it will likely lead to a production incident that engineers will have to troubleshoot in real-time.

Managing ephemeral containers

As a consequence of this shift from "pets to cattle", we can no longer deploy services with the assumption that they'll run perpetually. Modern applications—rather, the individual services that make up an application—must be designed with the assumption that they might be terminated, restarted, or migrated at any time. This isn't just because servers can shut down, but because Kubernetes can increase or decrease the number of container instances in response to system load and user demand. This short-lived, transitory nature of containers is called ephemerality.

Ephemerality offers many benefits: for example, if a container crashes, Kubernetes can automatically replace it with a fresh container running the exact same code. However, it can also introduce new failure modes. If a service doesn't support being automatically restarted, or if it crashes shortly after starting, then Kubernetes' self-healing mechanisms could end up being a detriment.

As an example, let's go back to our banking application's mobile deposit service. Based on historical trends, we know that our service gets used in high volumes at certain times. Usually there's a massive surge on Thursday evenings due to it being payday for most people, while on other days, we might only get a few dozen deposits. Using Kubernetes, we can configure a Horizontal Pod Autoscaler to increase the

number of Pods once resource usage hits a certain threshold. To avoid running out of compute capacity, we can also configure our cloud provider to add nodes to our cluster using the Cluster Autoscaler. This lets us automatically deploy new servers and instances of our mobile deposit service on Thursday afternoons, then scale back down to normal capacity by Friday morning. This way, we avoid the risk of an outage due to resource exhaustion, avoid wasting excess capacity on the other days of the week, and maintain a good customer experience.

Managing persistent data

Since cloud computing made servers replaceable and containers made services ephemeral, managing persistent data for containers has become a challenge. Services like databases, file stores, and email servers all require access to long-term data storage, but how do we store data for a service when we don't even know which servers it'll be running on?

While there are multiple solutions available, there are also tradeoffs. We could use a distributed filesystem like Gluster, a cloud storage service like Amazon S3, or even a more traditional option like NFS to ensure our containers can access their files no matter where they are in the cluster. However, we'll need to consider what happens when those services are unavailable due to an outage, or when network latency makes file retrieval impractically slow. We'd also need to consider what happens when:

- Multiple instances of the same service try to access the same data simultaneously. Do we run into file conflicts, locks, or integrity problems?
- Poor network speed affects data transfer between the container and storage, especially if we're migrating from high-speed local (i.e. disk-based) storage to network-based storage.
- The cost and logistics of storing, replicating, and backing up a cloud-based distributed data storage platform increases as we scale up our application.

If a container can't function without access to its files, it's at risk of failing if the file service also fails. We should proactively test to make sure any containers that use persistent storage continue running if that storage is unavailable, or fail gracefully if possible.

Moving from local communication to network communication

In a monolithic application, passing data from one function of the application to another is often extremely fast since the data resides within the same memory space.

With microservices, two parts of the application may not even be running on the same server, let alone sharing memory. We have to assume services will communicate over the network, and even the fastest gigabit networks are orders of magnitude slower than RAM. This can have a massive impact on application throughput and latency, especially for services that communicate frequently or pass large amounts of data. When splitting services into containers, consider how tightly interlinked (or dependent) one service is on another and how latency could impact this connection. It might be more beneficial to deploy these two services together in a single Pod.

Looking back at our bank example, each mobile deposit that a customer makes requires them to send a photo of the check or money order they want to deposit. Our mobile deposit service needs to then forward this photo to the separate image processing service for analysis. Each photo can be anywhere from a few hundred KB to several MB, and since we'll be receiving photos for every deposit, this bandwidth quickly adds up.

Deploying the image processing service as its own Pod means Kubernetes could schedule it on completely separate nodes from our mobile deposit service, which would add a significant amount of back and forth network communication. However, if we deploy the image processing service container in the same Pod as our mobile deposit service container, we avoid this need, thereby speeding up communication and reducing bandwidth usage.

Moving from local communication to network communication

One of the most overlooked factors of Kubernetes reliability isn't Kubernetes itself, but the developers and engineers who will be using it. Shifting from a legacy monolith to a Kubernetes-based microservice architecture forces engineers to rethink the way they build, deploy, and maintain their applications. Engineers will need time to learn the new system and best practices before the migration process can even begin, but this means the risk of bugs getting introduced will be high as they get up to speed.

Some of the adjustments engineers will need to make include:

- Not thinking of their applications as single units, but as collections of smaller, independent units called services.
- Moving away from manual administration tools like SSH and more towards automation using features like ReplicaSets, Pod lifecycle management, and liveness probes.

- Adopting Site Reliability Engineering (SRE) and DevOps best practices, which have evolved alongside microservice applications and cloud-native architectures.

For example, the mobile deposit development team at your bank is used to having to merge code changes into a single codebase along with several other teams, then deploy them out as a single update. Now that you're using Kubernetes, your team can deploy their own changes independently of other teams. This requires a significant shift in how your team manages code quality, pushing updates, and integrating with other services.

Reliability concerns when adopting Kubernetes

Kubernetes is a complex system, which means there are many different places where problems can arise. Even relatively small problems–such as a container consuming more CPU or memory than expected–can result in outages due to the complex and interdependent systems in place in a Kubernetes cluster.

There are many different levels where failures can occur, including:

- **Infrastructure-level failures:** these include power outages, hardware failures, and if we're using a managed Kubernetes service, cloud provider outages.
- **Cluster-level failures:** these include misconfigured clusters, autoscaling problems, and unreliable or insecure network connections between nodes.
- **Node-level failures**: these include automatic or unscheduled reboots, kernel panics, problems with the kubelet process or other Kubernetes-related process, and network connectivity problems.
- **Deployment-level failures:** these include misconfigured Deployments, too few (or too many) replicas, missing or failing container images, and Deployment failures due to limited cluster capacity.
- **Application-level failures:** these include application crashes, source code errors, and unhandled exceptions.

We also need to consider emergent behaviors, where a failure in one component impacts an entirely different component. For example, if our cluster autoscaler isn't working properly, it could cause our nodes to run out of compute resources, which causes failed Deployments and latency for our actively running applications.

One thing that's certain is that Kubernetes clusters are susceptible to a wide range of different failure modes across multiple levels of operation, from the underlying infrastructure all the way up to the applications running in our containers.

Migration strategies

With these reliability concerns in mind, how do we go about migrating to Kubernetes in the least risky or failure-prone way? There are many different migration strategies, each with their own benefits and drawbacks. We'll focus on three of the most common: rehosting (also known as "lift and shift"), refactoring, and replatforming.

Rehosting ("Lift and shift")

Rehosting, or "lift and shift," effectively means taking an application as-is and redeploying it onto another platform. Engineers can take their monolith, package it into a container, and deploy that container using Kubernetes. There are benefits to this, such as not having to rewrite or redesign the application and not having to spend a significant amount of upfront time learning an entirely new architecture. Down the road, once the team is comfortable operating their application on Kubernetes, we can consider refactoring.

However, rehosting is much less effective than other methods. While the application is technically running on Kubernetes, it's missing some key benefits of containers and microservices, such as lower startup times, more efficient resource utilization, and separation of concerns into individual services. If the container running the monolith fails, then the entire application fails. In addition, containers with larger applications generally take longer to start and stop than more lightweight containers, which can cause problems when recreating or migrating containers across nodes.

Refactoring

Refactoring is the ideal option for teams that want to leverage microservices to their full potential. Refactoring means restructuring or rewriting an application so that it's optimally designed for the architecture it will be deployed on. For Kubernetes, this can include:

- Splitting off functionality into separate containers.
- Creating well-defined APIs for communication across containers and services.
- Building fault tolerance into each service (for example, can our bank's mobile deposit service continue running if the image recognition service is unavailable?).
- Running extensive performance, reliability, and quality assurance tests to ensure the new design is both performant and stable.

Refactoring is the preferred method since it lets us get the most out of Kubernetes, but it's also time-consuming and error prone. Not only are we restructuring our application at its core, but our engineers are likely beginners with Kubernetes and don't yet understand all of the traps, risks, and pitfalls that intermediate and senior engineers are aware of. There are plenty of stories of Kubernetes deployments failing due to unpredictable and sometimes bizarre problems emerging from complex interactions between Kubernetes, applications, and other tools.

When refactoring, it's especially important to prioritize reliability testing throughout the development process. This helps proactively uncover failure modes so that our engineers can address them before they happen in production. In turn, this saves our customers and our incident response teams from the stress of a live system outage.

Replatforming

The last migration method we'll discuss is replatforming, where instead of migrating the entire application all at once, we migrate one component at a time. For example, imagine our bank is still using a monolithic architecture, and we want to add a financial planning feature to our application. Instead of building this new feature into the monolith, we can build and deploy the feature as a microservice using Kubernetes. This lets us "test the waters" of Kubernetes without having to rewrite our entire application or retrain our entire development team at once. Once we've evaluated the success of this new service, we can take our learnings and apply them to other parts of our application in a later refactor or rehosting.

Replatforming also significantly reduces the impact that reliability issues could have on our application by limiting it to a single service. If the service fails, we can still fall back to our legacy monolith while we troubleshoot and resolve the problem. We should make sure to discuss and document the reliability issues we find so that we can prevent them when migrating other services in the future. Over time, more and more functionality will move from the legacy application to Kubernetes until we can eventually deprecate the monolith. By that point, we should have a resilient, well-understood Kubernetes-based infrastructure.

Reliability recommendations while you undergo your Kubernetes migration

We've discussed the unique design of Kubernetes, the reliability concerns that it introduces, and how different migration strategies can help address or exacerbate these issues. To reduce the risk of failures and other reliability issues, consider these recommendations when starting your migration.

01 Recognize that reliability is a practice, not a destination

Reliability doesn't end once you deploy your Kubernetes application to production. It's an ongoing practice that must be continuously iterated on, especially as you scale up your cluster and applications. Remember, Kubernetes is a complex platform with many moving parts, countless potential configurations, many dependencies, and several layers of software and infrastructure. Failure modes can emerge from anywhere at any time, which is why it's important for engineers to learn how to identify and address them.

In addition, as time passes and your clusters grow, new failure modes will emerge and old failure modes will re-emerge as regressions. Building an ongoing reliability practice ensures that you can detect and address both types of failure modes before they cause production issues. This doesn't mean that Kubernetes can't ever be reliable, but engineers need to be aware that complexity and change create the potential for failures. Kubernetes is an ever-growing, ever-changing system, so engineers need to be always vigilant for risks to reliability.

02 Encourage engineers to learn and understand Kubernetes' design

The more an engineer knows about how a system works, the better equipped they are to make it more resilient. They'll have exposure to more features and resources designed to help improve reliability, and are less likely to make beginner mistakes. This doesn't mean they need to know everything about Kubernetes to make it fault-tolerant—it's extremely difficult to know every single aspect of a system as complex as Kubernetes—but understanding how components such as containers, Pods, nodes, and the underlying infrastructure impact each other can prevent many common types of failures.

We covered some of the key aspects of Kubernetes' design earlier in this whitepaper, such as Pods, ephemerality, data persistence, and network communication. There are

many more layers to Kubernetes, such as cluster provisioning, container scheduling, container restart policies, affinity rules, and taints and tolerations. As unexpected problems emerge, make sure to document these problems and their solutions, then share these insights with other engineering teams. Teaching others about potential pitfalls and failure modes will help them avoid making the same mistakes when migrating their own services, and may provide some additional insight into Kubernetes' behavior that you didn't notice on your own.

Consider also classifying these failure modes according to their source component. If a problem is related to Pods, networking, permissions, etc, identify it as such. If you notice a large number of problems emerging from certain components, you know to focus extra time and attention on not only making that component more resilient, but doing additional research to understand how that component works and how to design it optimally.

03 Proactively discover failure modes

It may seem counterintuitive to deliberately look for ways Kubernetes can fail, but it's the best way to verify the resilience of a system. Proactively looking for failure modes gives you the opportunity to fix or document those faults before they can become production issues or outages, while also teaching you more about how the system works.

An effective way of doing this is by running a GameDay, which is a period of time set aside for a team to run one or more reliability tests on a system, discuss the technical outcomes, and use the insights gained to improve the system's resilience. GameDays allow teams to test their assumptions about how a system works and discover new ways of improving those systems. They also give engineers the chance to practice responding to real-world failures so that if these same failures happen in production, we can respond quickly and effectively. GameDays involve four or more team members, with each team member taking on a role:

- The Owner manages the overall GameDay and decides on the plan, schedule, and whether to stop the GameDay.
- The Coordinator prepares and executes the experiments in your testing tool of choice and coordinates other participating roles to keep everyone on track.
- Observers gather data from monitoring and observability tools, inform other participants on key observations, and verify the results. Unlike the other roles, a GameDay can have more than one Observer.

- The Reporter records notes, key observations, and results in a GameDay tracking tool.

GameDays should focus on a specific test or assumption. Let's go back to our banking application as an example. Imagine we've just deployed our new financial planning service in Kubernetes and have it running alongside our monolith. Consider what would happen if communication between our new service and our monolith suddenly stopped. How will our service adapt to this failure? Will it try to reconnect endlessly, will it eventually timeout, or will it crash? Will our users see an error message, an endless loading screen, stale data, or something else that we didn't anticipate? We can't know for certain unless we either run into this issue in production, or we proactively test this scenario by running a GameDay and using our learnings to remediate the problem before it can impact customers.

Click here to learn more about GameDays at GREMLIN.COM/GAMEDAY

04 Be open to learning from incidents

Incidents aren't the end of the world. When something breaks in production, that doesn't mean the migration was a failure. Failures will happen, and they're more likely to happen during a massive transition like a migration from a monolith to Kubernetes. Think about all of the aspects that could impact reliability:

- Legacy components still need to be maintained during the migration.
- New services need to be able to communicate with legacy applications.
- Engineers are still learning how to deploy resilient workloads and apply best practices. This includes exploring new tools and mechanisms for observability and self-healing.

When something fails, take it as an opportunity to study the problem, identify the root cause, then design and implement practices to prevent it from recurring. Foster a blameless culture within your engineering organization, one where teams don't blame each other for causing problems, but instead focus on the policies, procedures, and technologies that allowed the problem to happen in the first place.

Conclusion

Migrating from a legacy architecture to Kubernetes is difficult even under the best circumstances. However, while it's not beginner-friendly, it offers a lot of assistance when it comes to keeping your applications running. Automatic container restarts, replica controllers, automatic worker node management, and cluster redundancy are all ways that Kubernetes can help you make your applications more resilient.

Remember that reliability isn't a one-and-done project, but an ongoing practice. Trying to learn Kubernetes, build a hardened cluster, and migrate a legacy application all at once will end up in an endless migration process. That's why it's important to continually practice reliability by actively seeking failure modes, encouraging engineers to adopt and prioritize reliability practices, and not blaming teammates when things go wrong, but instead reviewing problems and sharing insights as a team.

Once you're ready to go beyond the basics and do more reliability testing on Kubernetes, make sure to read The first 5 chaos experiments to run on Kubernetes. You can also watch our webinar, Running your first 5 chaos experiments on Kubernetes.

"In a blameless culture, we start from the belief that people are doing their best given the resources and information they have available."

James Thigpen
DIRECTOR OF ENGINEERING, GREMLIN

Click here to learn more about teams & culture from JAMES THIGPEN

Gremlin

Gremlin is the enterprise Chaos Engineering platform on a mission to help build a more reliable internet. Their solutions turn failure into resilience by offering engineers a fully hosted SaaS platform to safely experiment on complex systems, in order to identify weaknesses before they impact customers and cause revenue loss.

ADDITIONAL RESOURCES

- Kubernetes documentation
- How to install and use Gremlin with Kubernetes
- If you're adopting Kubernetes, you need Chaos Engineering