Choosing the right deployment pattern for your application

medium.com/@siavash.sajjady/choosing-the-right-deployment-pattern-for-your-application-490b061448b8

Siavash Sajadi December 25, 2022

Deployment patterns are an essential aspect of software development and deployment, as they provide a set of best practices and considerations for releasing new versions of an application in a production environment. In this article, we'll take a closer look at some common deployment patterns, including their technical implementation and the latest methods for using them.

Consequences of not using deployment patterns in software development and deployment:

If you don't have experience with deployment patterns, it's important to familiarize yourself with them before developing and deploying software. Deployment patterns provide a set of best practices and considerations for releasing new versions of an application in a production environment, and understanding these patterns can help ensure that the deployment process is efficient, reliable, and easy to manage.

Not using deployment patterns or not understanding how to use them effectively can lead to issues such as:

- Downtime or disruption during the deployment process
- Difficulty rolling back to a previous version if necessary
- Difficulty testing and verifying the stability of new versions before releasing them
- Difficulty gathering feedback and monitoring the performance of new versions

Imagine a team is developing an e-commerce website and they are preparing to release a new version of the site that includes a major redesign and several new features. The team decides to deploy the new version of the site by simply replacing the old version with the new one.

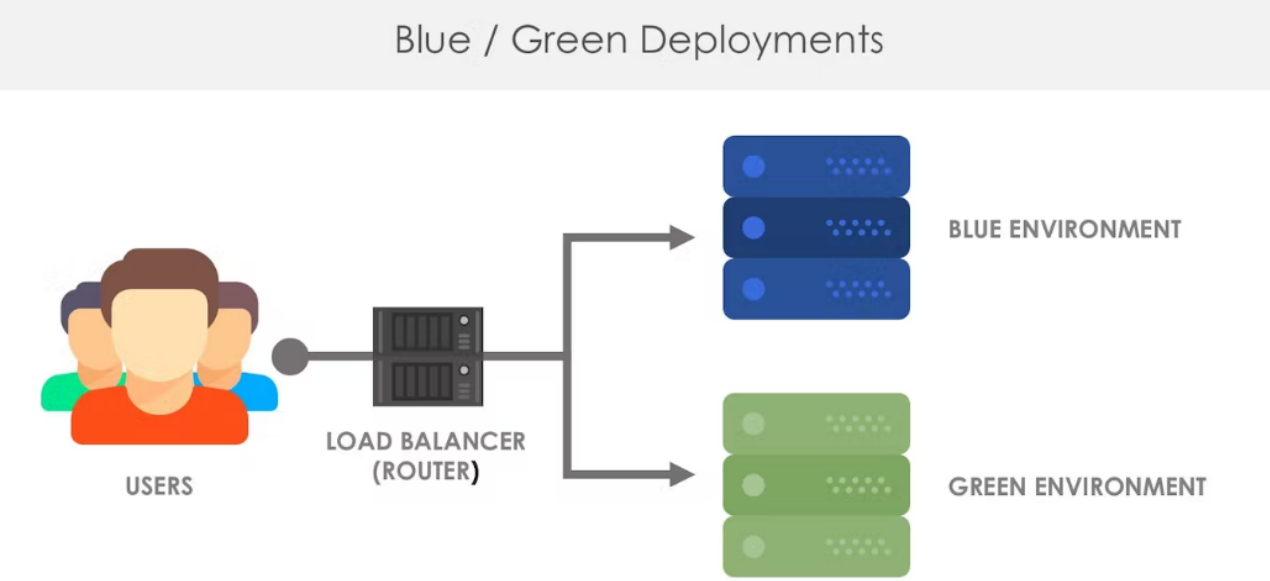
However, the team did not use any deployment pattern or properly test the new version before releasing it. As a result, when the new version of the site goes live, it experiences significant issues such as slow loading times, errors, and broken pages. The team is unable to roll back to the previous version of the site, and they are forced to spend significant time and resources fixing the issues with the new version.

If the team had used a deployment pattern such as blue-green deployment or canary deployment, they could have avoided these issues by properly testing and verifying the new version before releasing it to the entire user base. They could have also used a pattern such as rolling deployment to minimize disruption to users during the deployment process.

By understanding and using deployment patterns, you can avoid these issues and ensure that the deployment process is smooth and successful. There are several deployment patterns to choose from, including blue-green deployment, canary deployment, rolling deployment, and A/B testing. It's important to choose the appropriate pattern based on the needs of the application and the requirements of the environment in which it will be deployed.

> If you're new to deployment patterns and are looking to learn more, there are many resources available online that can provide more detailed information and guidance. You may also want to consider seeking guidance from more experienced developers or consulting with an expert in the field.

Blue-green deployment:



The *blue-green deployment* pattern is a widely used method for releasing new versions of applications in a production environment. It involves running two identical production environments, known as “blue” and “green,” in parallel. When a new version of the application is ready for deployment, it is first released to the green environment. If the deployment is successful, the green environment becomes the active production environment, while the blue environment is put on standby. If there are any issues with the deployment, the blue environment remains active and the green environment is used for debugging and troubleshooting.

The blue-green deployment pattern can be implemented using a load balancer to route traffic between the two environments. The load balancer can be configured to send a certain percentage of traffic to the green environment and the remainder to the blue environment. When the new version is deployed to the green environment and is deemed stable, the load balancer can be updated to send all traffic to the green environment. This allows for a seamless switchover between environments without disrupting the production environment.

> One of the main advantages of the blue-green deployment pattern is that it allows for quick rollbacks in the event of a deployment issue. Since the blue environment is kept in a fully operational state, it can be quickly activated if there are problems with the new version deployed to the green environment. This reduces the risk of prolonged downtime and ensures that the production environment remains stable.

To implement blue-green deployment, the following steps are typically followed:

1. Create a new instance of the application and deploy the new version (the green version) to it.
2. Test and verify the new version to ensure that it is stable and ready for production.
3. Switch the traffic from the old version (the blue version) to the new version (the green version). This is typically done by updating a load balancer or routing configuration to point to the new version.
4. Monitor the performance of the new version to ensure that it is stable and functioning as expected.
5. If there are any issues with the new version, switch the traffic back to the old version until the issues are resolved.
6. If the new version is stable and performing as expected, decommission the old version and continue using the new version.

Blue-green deployment allows teams to minimize disruption to users during the deployment process, as the traffic is switched from the old version to the new version in a controlled manner. It also provides an easy way to roll back to the previous version if necessary, which can be beneficial in situations where there are significant risks associated with the deployment process.

> Today, blue-green deployment is commonly used in cloud environments, where it is easier to create and manage multiple instances of an application. To implement blue-green deployment in a cloud environment, the application team can create a new instance of the application and test it before switching the traffic from the old instance to the new instance.

What are the considerations for choosing blue-green deployment over other deployment patterns?

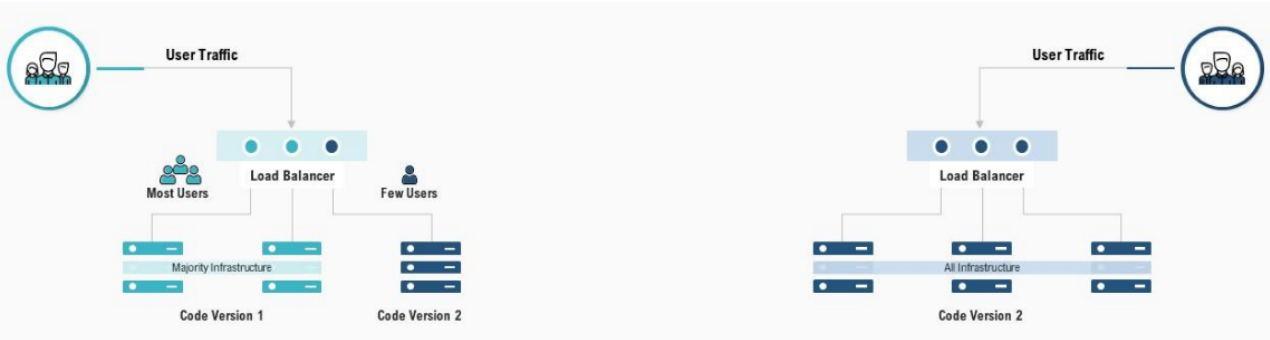
There are several considerations to keep in mind when choosing blue-green deployment over other deployment patterns:

- Cost: Blue-green deployment may require the use of additional resources, such as additional instances of the application or additional environments, which can impact cost.
- Testing and verification: Blue-green deployment requires more upfront testing and verification of the new version before it is released, which can impact the time required for the deployment process.
- Rollback: Blue-green deployment provides an easy way to roll back to the previous version if necessary, which can be beneficial in situations where there are significant risks associated with the deployment process.
- Disruption: Blue-green deployment minimizes disruption to users during the deployment process, which can be beneficial in situations where it is important to minimize downtime.

These considerations should be taken into account when deciding whether blue-green deployment is the appropriate deployment pattern for a given project. It's important to choose the deployment pattern that best fits the needs and goals of the application and the team developing it.

Canary deployment:

Canary deployment is a deployment pattern that involves releasing a new version of an application to a small group of users first, before rolling it out to the rest of the user base. This allows the application team to test the new version with a limited number of users and gather feedback before releasing it to the entire user base.



> Canary deployment also dates back to the earliest days of software development, when it was common to release new versions of an application to a small number of users first to ensure that the new version was stable and bug-free.

To implement canary deployment, the application team can create a new environment or subset of users that will receive the new version of the application. They can then monitor the performance of the new version and gather feedback from the small group of users before rolling it out to the rest of the user base.

What are the considerations for choosing canary deployment over other deployment patterns?

There are several considerations to keep in mind when choosing canary deployment over other deployment patterns:

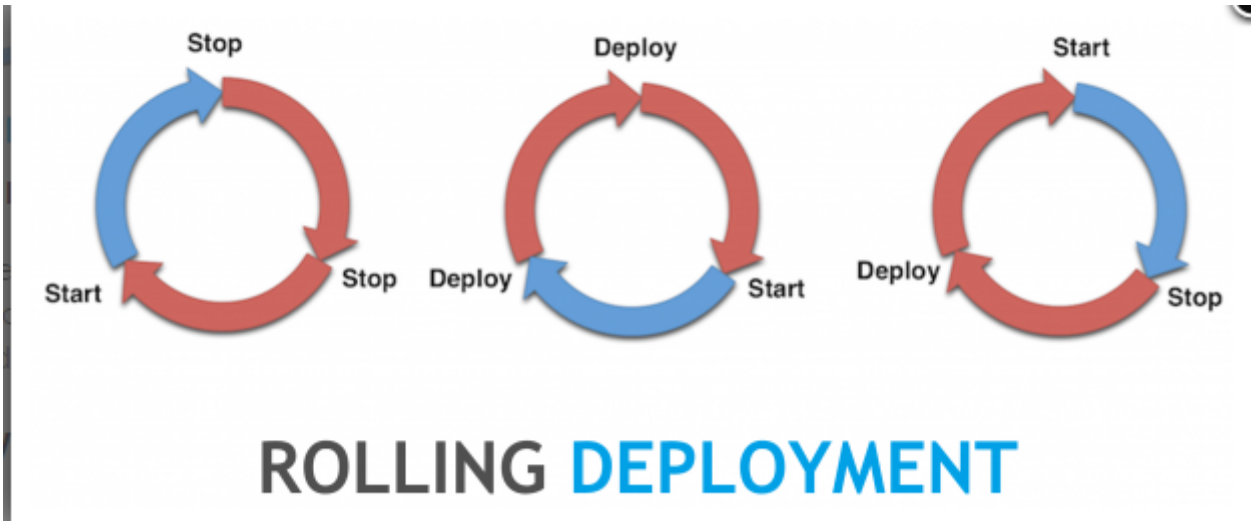
- Continuous feedback: Canary deployment allows for continuous feedback and monitoring of the new version during the deployment process, which can be beneficial in situations where it is important to gather ongoing feedback on the new version.
- Risk management: Canary deployment allows for controlled and gradual rollouts of the new version, which can help to reduce risk and minimize the impact of any issues that may arise.
- Testing and verification: Canary deployment requires less upfront testing and verification of the new version before it is released, which can reduce the time required for the deployment process.
- Cost: Canary deployment may require ongoing monitoring and feedback gathering, which can impact cost.
- Disruption: Canary deployment may involve some disruption to users during the deployment process, as the new version is gradually rolled out to a portion of the user base.

> One of the main advantages of the canary deployment pattern is that it allows for early detection and resolution of issues with the new version. It is also less resource-intensive than the blue-green deployment pattern, as it does not require maintaining multiple production environments. However, it may not be suitable for applications with a high volume of users or transactions, as the canary group may not be representative of the entire user base.

In a cloud environment, canary deployment can be implemented more easily and quickly than in a traditional on-premises environment, as it is easier to create and manage multiple instances of an application in the cloud. This makes it an attractive option for teams looking to gather ongoing feedback and monitor the performance of the new version during the deployment process. It can be implemented in a cloud environment by following these steps:

1. Create a new instance or instances of the application in the cloud environment and deploy the new version to it.
2. Configure a load balancer or routing configuration to direct a portion of the traffic to the new instance or instances of the application. This allows the new version to be gradually rolled out to a portion of the user base.
3. Monitor the performance of the new version to gather feedback and ensure that it is stable and functioning as expected.
4. Gradually increase the percentage of traffic directed to the new instance or instances as the new version is tested and verified.
5. If there are any issues with the new version, roll back to the previous version by reconfiguring the load balancer or routing configuration to direct all traffic back to the old instance or instances.
6. If the new version is stable and performing as expected, decommission the old instance or instances and continue using the new version.

Rolling deployment:



Rolling deployment is a deployment pattern that involves gradually replacing old versions of an application with the new version, one group of servers or instances at a time. This allows the application team to roll out the new version gradually, minimizing disruption to users and allowing them to closely monitor the performance of the new version.

Rolling deployment is often used in environments where it is important to minimize downtime, such as in mission-critical applications or applications with a large user base. To implement rolling deployment, the application team can divide their servers or instances into groups and deploy the new version to one group at a time. They can then monitor the performance of the new version and roll it out to additional groups if it is performing well.

Imagine a team is developing a social media platform and they are preparing to release a new version of the platform that includes several new features and improvements. The team decides to use rolling deployment to minimize disruption to users during the deployment process.

To implement rolling deployment, the team creates a new instance of the platform and deploys the new version to it. They then test and verify the new version to ensure that it is stable and ready for production.

Once the new version has been tested and verified, the team gradually rolls out the new version to a portion of the user base, while the old version continues to serve the rest of the users. As the new version is rolled out, the team monitors its performance to ensure that it is stable and functioning as expected.

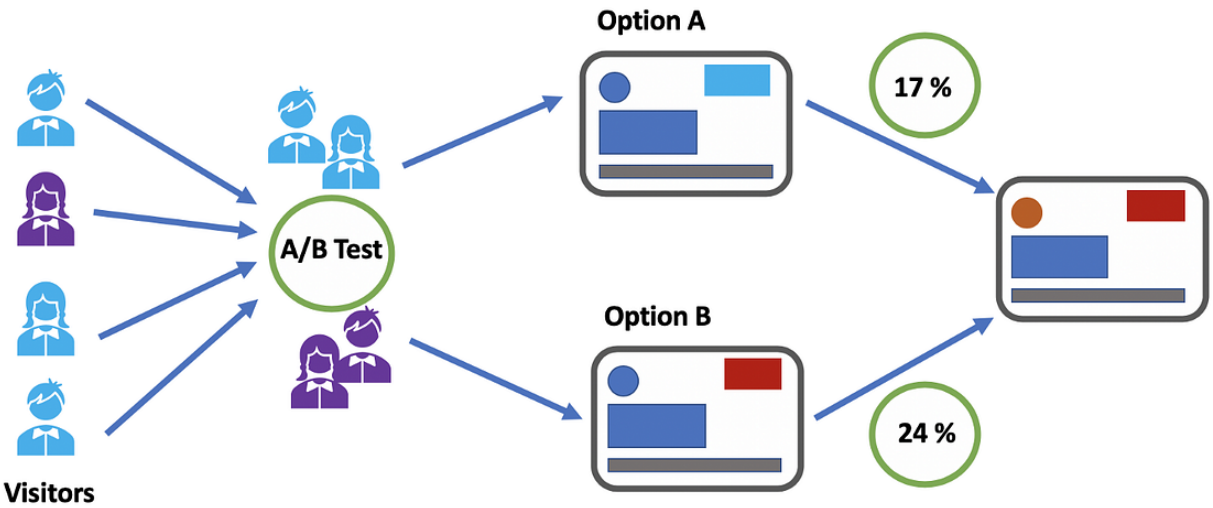
If there are any issues with the new version, the team can roll back to the previous version by directing all traffic back to the old instance. If the new version is stable and performing as expected, the team continues rolling out the new version to the rest of the user base until it is serving all users.

> One of the key benefits of the rolling deployment pattern is that it allows for real-time monitoring and testing of the new version in a live environment, enabling quick identification and resolution of any issues that may arise. It also allows for a smoother transition for users, as the rollout is gradual and not all users are affected at once. However, this method can be more time-consuming and requires careful planning and coordination to ensure a successful rollout.

By using rolling deployment, the team is able to minimize disruption to users during the deployment process and ensure that the new version of the platform is stable and ready for production. They are also able to easily roll back to the previous version if necessary, which helps to reduce risk and ensure the reliability of the platform.

A/B testing:

A/B testing is a deployment pattern that involves deploying two different versions of an application and comparing their performance to determine which one is better. This can be useful for testing new features or designs, or for comparing the performance of different algorithms or approaches.



To implement A/B testing, the application team can create two different versions of the application and deploy them to separate groups of users. They can then monitor the performance of each version and compare the results to determine which version is better.

Here is an example of how A/B testing could be used in a real project: Imagine that a company has developed a new landing page for its website and wants to test which version will be more effective at converting visitors into customers. The company can create two versions of the landing page, one as the control and one as the treatment. They can then randomly assign a percentage of visitors to each page and track their actions, such as whether they make a purchase or sign up for a newsletter.

After a certain amount of time, the company can compare the conversion rates of the two pages and determine which version is more effective. They can then use the results of the A/B test to make informed decisions about which version of the landing page to use going forward.

How to implement A/B testing ?

To implement A/B testing, you can follow these steps:

1. Identify the feature or element that you want to test. This could be anything from the layout of a particular screen to the placement of a call-to-action button.
2. Create two versions of the feature or element, one as the control and one as the treatment. Make sure to keep all other elements of the app consistent between the two versions.
3. Use a tool or service to randomly assign a percentage of users to each version of the feature or element. Some tools that can help with this include Optimizely, Google Optimize, and Apptimize.
4. Track and measure the results of the test. This could include metrics such as the number of users who complete a specific action (e.g., making a purchase or signing up for a newsletter), the amount of time users spend in the app, or the number of users who return to the app after their first visit.
5. Analyze the results and make a decision based on which version performed better. If the treatment version outperformed the control version, you may want to consider implementing it in the app. If the control version performed better, you may want to stick with that version or try testing a different variation.

> A/B testing is often used in situations where it is important to understand the impact of a specific change or feature on an application’s performance. It can also be useful for testing different approaches or designs to see which one performs better.

In-place deployment:

In-place deployment is a type of deployment pattern that involves updating a running application in place, without creating a new version or instance of the application. This is typically done by replacing or updating the code, configuration, or other resources of the application while it is running.

There are a few different approaches to in-place deployment, depending on the specific needs and constraints of the project. One common method is to use a rolling update, where the application is updated one server or instance at a time, allowing for a more controlled rollout and minimizing the risk of disrupting the entire production environment. Another approach is to use a blue-green deployment pattern, where the application is updated in a separate environment and then switched over to the active production environment when the update is complete.

In-place deployment has a number of advantages, including the ability to deploy updates quickly and with minimal downtime. It is particularly useful for applications that require continuous availability or have a high volume of users or transactions. However, it can also be risky, as there is a higher likelihood of disrupting the production environment if the update is not successful. In-place deployment is also not suitable for applications that require a complete rebuild or restructuring, as it is not possible to create a new version or instance of the application.

Here is an example of how in-place deployment could be used in a practical scenario: Imagine that a company has a web application that is used by thousands of users. The application is hosted on a cluster of servers and is designed to be continuously available. The company wants to release a new version of the application that includes several bug fixes and performance improvements.

To deploy the new version using the in-place deployment pattern, the company could use a rolling update approach. This would involve updating the application on one server at a time, and gradually rolling out the update to the rest of the servers in the cluster. The company could use a load balancer to route traffic to the updated servers, and monitor the performance of the new version to ensure that it is stable and does not disrupt the production environment.

Once the update has been fully deployed to all servers in the cluster, the company could switch over to the new version and retire the old version. This would allow for a quick and seamless deployment of the new version with minimal downtime for users.

Looking to the future, it is likely that deployment patterns will continue to evolve and become more automated and intelligent. For example, machine learning algorithms could be used to optimize deployment strategies by analyzing data on past deployments and identifying patterns that are associated with successful or unsuccessful deployments. Additionally, the use of containerization technologies such as Docker and Kubernetes will likely become more prevalent, as they allow for more efficient and flexible deployment of applications in cloud environments.

In conclusion, the blue-green, rolling, canary,In-place, and A/B testing deployment patterns are all effective methods for deploying new versions of applications in production environments. Each method has its own advantages and limitations, and the most appropriate pattern will depend on the specific needs and constraints of the project. Developers should carefully consider the trade-offs between the different deployment patterns and choose the one that is most appropriate for their specific use case.

Here is a table that evaluates the blue-green, rolling, canary, in-place, and A/B testing deployment patterns:

Deployment Pattern	Advantages	Limitations
Blue-green	Quick rollbacks in the event of a deployment issue	Resource-intensive; not suitable for applications with long-running processes or transactions
Rolling	More flexibility and control over the deployment process	Slower rollout of the new version
Canary	Early detection and resolution of issues with the new version	May not be representative of the entire user base for applications with a high volume of users or transactions
In-place	Quick and seamless deployment with minimal downtime	Higher risk of disrupting the production environment; not suitable for applications that require a complete rebuild or restructuring
A/B testing	Ability to compare the performance of different versions of an application	Limited to testing a small percentage of users or servers; may not be representative of the entire user base

Looking to the future, we can expect to see the continued evolution and automation of deployment patterns, as well as the increasing adoption of containerization technologies.

Best Practices for Deploying Your React Native App to the iOS App Store

medium.com/@tusharkumar27864/best-practices-for-deploying-your-react-native-app-to-the-ios-app-store-92b6a0ddcba6

TUSHAR KUMAR April 4, 2025

Hey there! I've deployed React Native apps to the App Store in the last two years. It's never been smooth sailing, but I've learned a ton. Let me share what actually works.

- An Apple Developer account ($99/year)
- A Mac (yes, you really do need one)
- Xcode installed and updated
- Your React Native app running well in development
- Patience (trust me on this one)

Step 1: Get Your App Ready for Production

Before touching the App Store, make sure your app isn't going to crash on users.

```
(__DEV__) {  .();}
.();
```

Check your app performance on real devices. The simulator is great, but it's not the real thing. Borrow iPhones from friends if needed to test on different models.

Step 2: App Store Connect Setup

This is where most developers hit their first wall.

1. Go to
2. Create a “New App” in the “My Apps” section
3. Fill in basic details — name, description, pricing, etc.

Here’s what tripped me up: Apple needs unique bundle IDs. I once spent hours debugging because I reused an ID from an old project.

```
CFBundleIdentifiercom.yourcompany.uniqueappname
```

Step 3: Certificates and Provisioning Profiles

This part confused me for weeks when I started. Here’s what actually matters:

1. You need a (think of it as your digital signature)
2. You need a (it connects your app to your certificate)

Xcode can handle this automatically, but if you’re using Fastlane or CI/CD, you’ll need to manage them manually.

I wasted days trying manual certificate management before discovering this simple approach:

```
fastlane match developmentfastlane match appstore
```

Step 4: Configure Your App for Production

Make these changes to your project configuration:

1. Set a proper app icon (I forgot this once and Apple rejected my app)
2. Update your `Info.plist` with required permissions

```
NSLocationWhenInUseUsageDescriptionWe show nearby stores based on your location

NSCameraUsageDescriptionWe need camera access to scan QR codes
```

The hardest part? Managing environment variables. Here’s what worked for me:

```
   = __DEV__   ?    : ;   ;  = .;

 = ;
```

Step 5: Build and Archive Your App

Open your project in Xcode. Then:

1. Select “Any iOS Device” (not a simulator)
2. Select Product → Archive

3. Wait (seriously, go make coffee)

Common error I kept hitting: “Archive Failed”. The fix was usually:

```
iospod install ..npx react-native clean-ios-build
```

Step 6: TestFlight Testing (Don’t Skip This!)

I shipped a broken app once because I skipped TestFlight. Never again.

1. Upload your archive to App Store Connect
2. Add internal testers (your team)
3. Test on real devices before public release

TestFlight revealed subtle issues I missed in development:

- Push notifications weren’t working in production
- Deep links broke in the live environment
- The app crashed on older iOS versions

Step 7: App Store Submission

Now for the real thing:

1. Complete all App Store metadata

- Screenshots (make them look good!)
- Description
- Keywords
- Age rating

2. Submit for review

3. Wait for Apple’s review (usually 24–48 hours)

Real Problems I’ve Faced (And How I Fixed Them)

Problem 1: App Size Too Large

My first app was 85MB — way too big for mobile download.

```
  { format }  ;

 moment ;
```

I also enabled Hermes JavaScript engine, which reduced my app size by 30%:

```
project.. = [  : ]
```

Problem 2: App Rejected for Crash on Launch

Apple found a crash I didn't catch. The issue? Native module initialization.

```
{   videoModule = .;   (videoModule) {    videoModule.();  }}  (error) {    .();}

videoModule = .;videoModule.();
```

Problem 3: Missing Privacy Policy

Apple rejected my app because I didn't have a privacy policy. Even simple apps need one now.

Solution: I created a basic privacy policy using PrivacyPolicies.com and hosted it on GitHub Pages.

Automating Deployments

Manual deployment is tedious. Here's how I automated everything:

```
: {  : ,}
```

My Fastlane setup:

```
lane    increment_build_number( )  build_app( ,  )  upload_to_testflight
```

This reduced my deployment time from 45 minutes to a single command.

What I Wish I'd Known Earlier

1. : They're different! Version is like "1.2.3" (visible to users). Build number is sequential (for your reference).
2. : I spent days trying to make Bitcode work before learning Apple deprecated it.
3. : You need to build for "Any iOS Device" or a specific device.
4. : Export and backup your certificates or you'll regret it later.

Getting Started Today

If you're ready to ship your React Native app:

1. Sign up for an Apple Developer account now (it takes time to activate)
2. Set up App Store Connect with your app information
3. Create a basic deployment script to automate the process
4. Run a TestFlight beta with friends before going public

Let's Connect

Deploying apps is hard, but you've got this! I'm happy to help if you get stuck — reach out to me with your questions. I've been exactly where you are now.

Remember: every developer struggles with App Store deployment at first. It gets easier with practice!

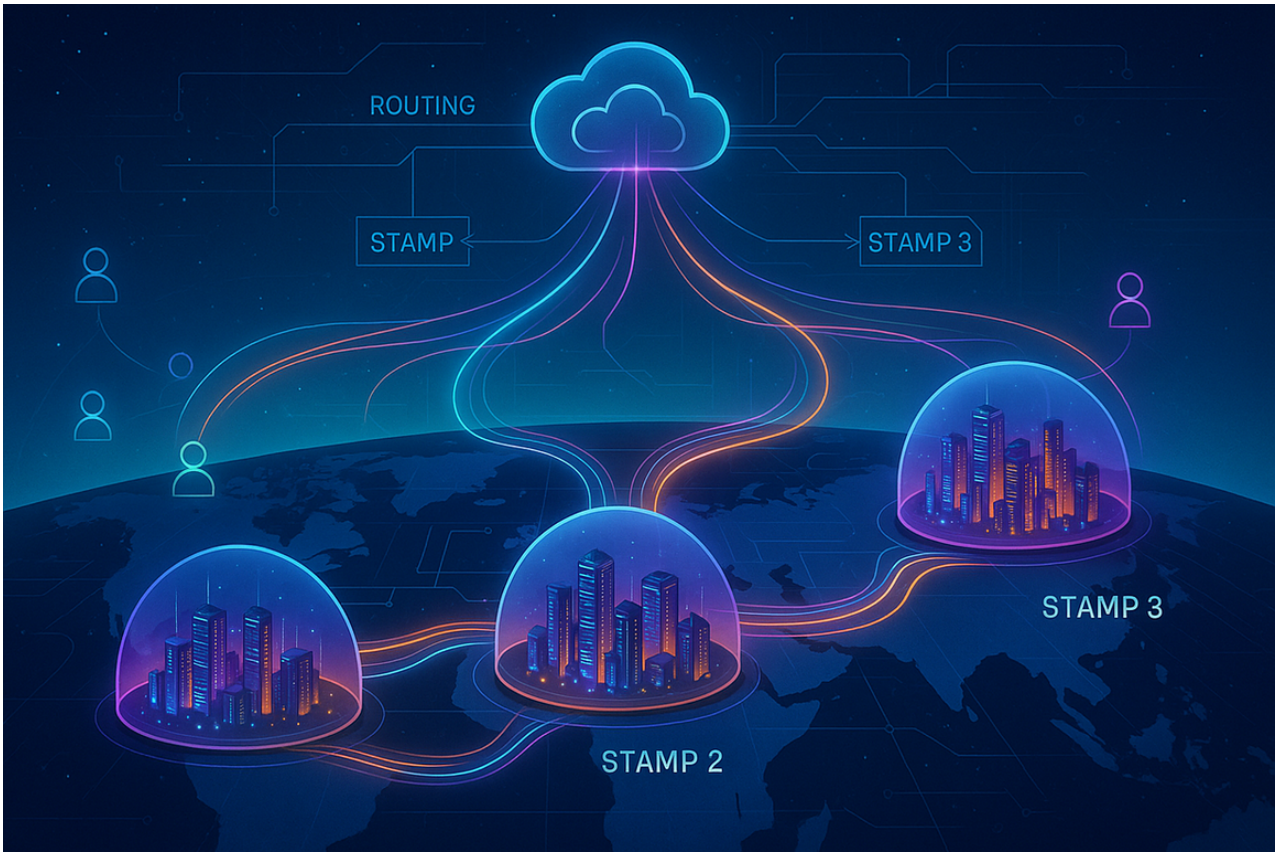
> If you found this helpful, follow me on Medium for more practical React Native tips.

The Stamp Deployment Pattern: Building Isolated, Scalable Cloud Ecosystems

medium.com/@vikpoca/the-stamp-deployment-pattern-building-isolated-scalable-cloud-ecosystems-6f6c976dc2f6

Viktor Ponamarev May 22, 2025

A brief overview of building resilient, compliant, and globally scalable cloud systems through the power of stamp deployments.



The **stamp deployment pattern** has emerged as a critical strategy for building hyperscale, resilient, and compliant cloud-native systems. This article explores its technical foundations, historical drivers, and real-world applications, supported by architectural diagrams and evidence from industry practices.

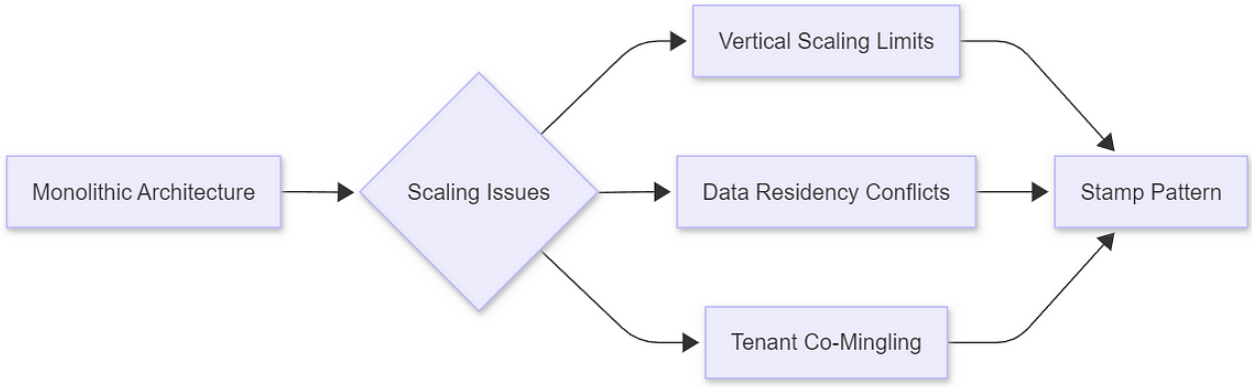
Pre-Cloud Scaling Challenges

Before cloud adoption, monolithic architectures dominated enterprise software. These systems faced three key limitations:

1. : Single database instances or application servers hit hardware limits, forcing costly upgrades.
2. : Data sovereignty laws (e.g.,) made single-region deployments non-compliant for international businesses.

3. : Multi-tenant SaaS platforms risked cross-customer data leaks in shared infrastructure.

Cloud providers like Microsoft Azure addressed these through **scale units**-preconfigured hardware stacks allowing horizontal scaling. However, manual scale unit management proved operationally burdensome, leading to the formalization of the stamp pattern.

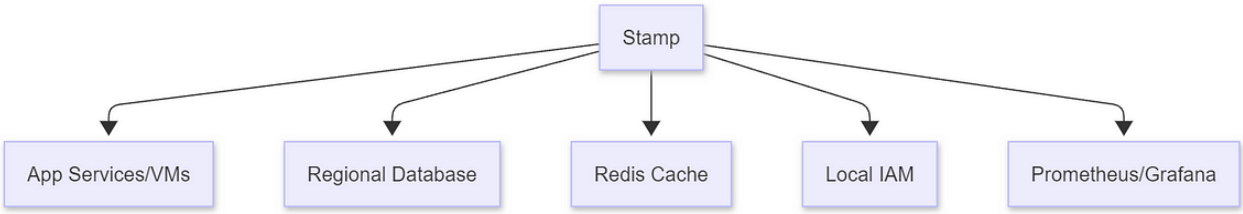


Architectural Anatomy of a Stamp

Core Components

A **stamp** (or *scale unit*) is a self-contained deployment unit containing:

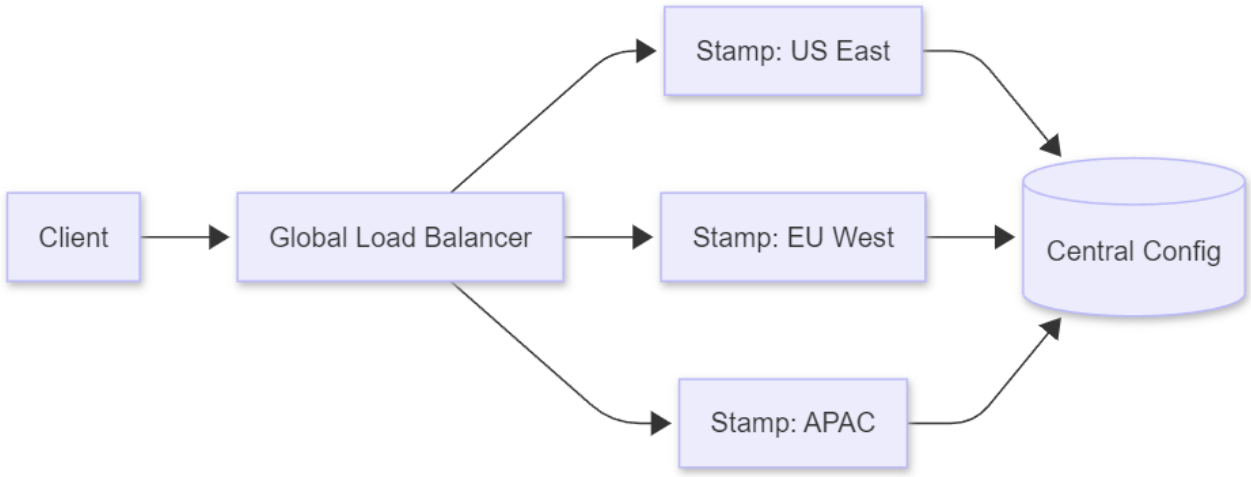
- Compute clusters (e.g., nodes)
- Dedicated databases (SQL/NoSQL)
- Regional caching layers
- Monitoring/telemetry agents



Global Orchestration Layer

Stamps coordinate through:

- : Routes traffic based on geo-location or tenant affinity, e.g.,
- : Centralized Helm charts or modules for stamp consistency
- : Event streaming via or for data synchronization



Real-World Implementation Cases

Case 1: Multi-Region SaaS Platform

A global HR software provider uses stamps to:

- Isolate EU customer data in Frankfurt/Germany stamps for
- Deploy dedicated high-CPU stamps for analytics clients with ML workloads
- Achieve 99.999% uptime via automatic failover between Sydney and Melbourne stamps during regional outages

Case 2: Financial Trading System

A stock exchange platform leverages stamps for:

- Ultra-low latency by placing stamps in co-location facilities near exchange servers
- Circuit-breaker isolation-errant trading algorithms in one stamp don’t impact others
- Regulatory audits using per-stamp activity logs

Implementation Guide

Step 1: Template Creation

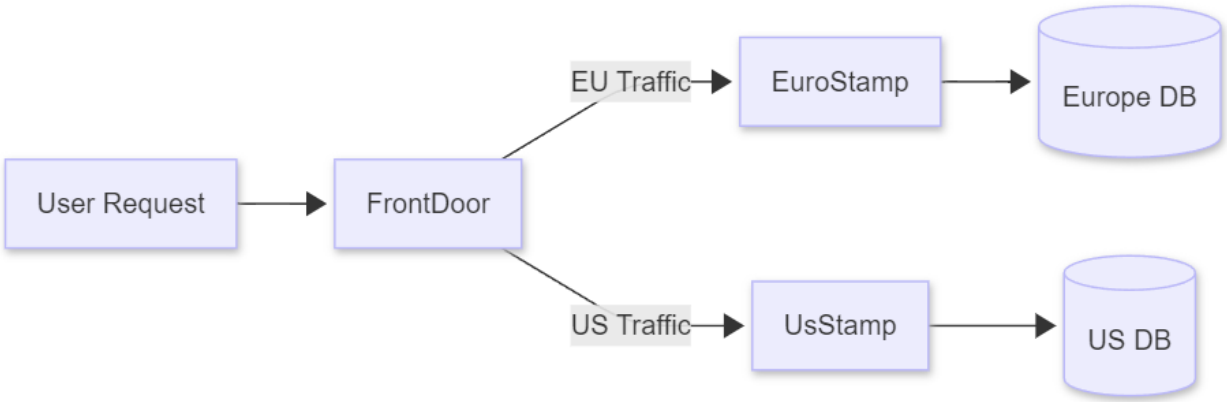
Define infrastructure-as-code (IaC) templates for reproducible stamp creation:

```
# Azure Stamp Module  module "eu_stamp" {    source = "./stamp-template"    region = "westeurope"    db_tier = "BusinessCritical"    tenant_capacity = 1000  }
```

Source: practices

Step 2: Traffic Management

Configure Azure Front Door with geo-routing rules:



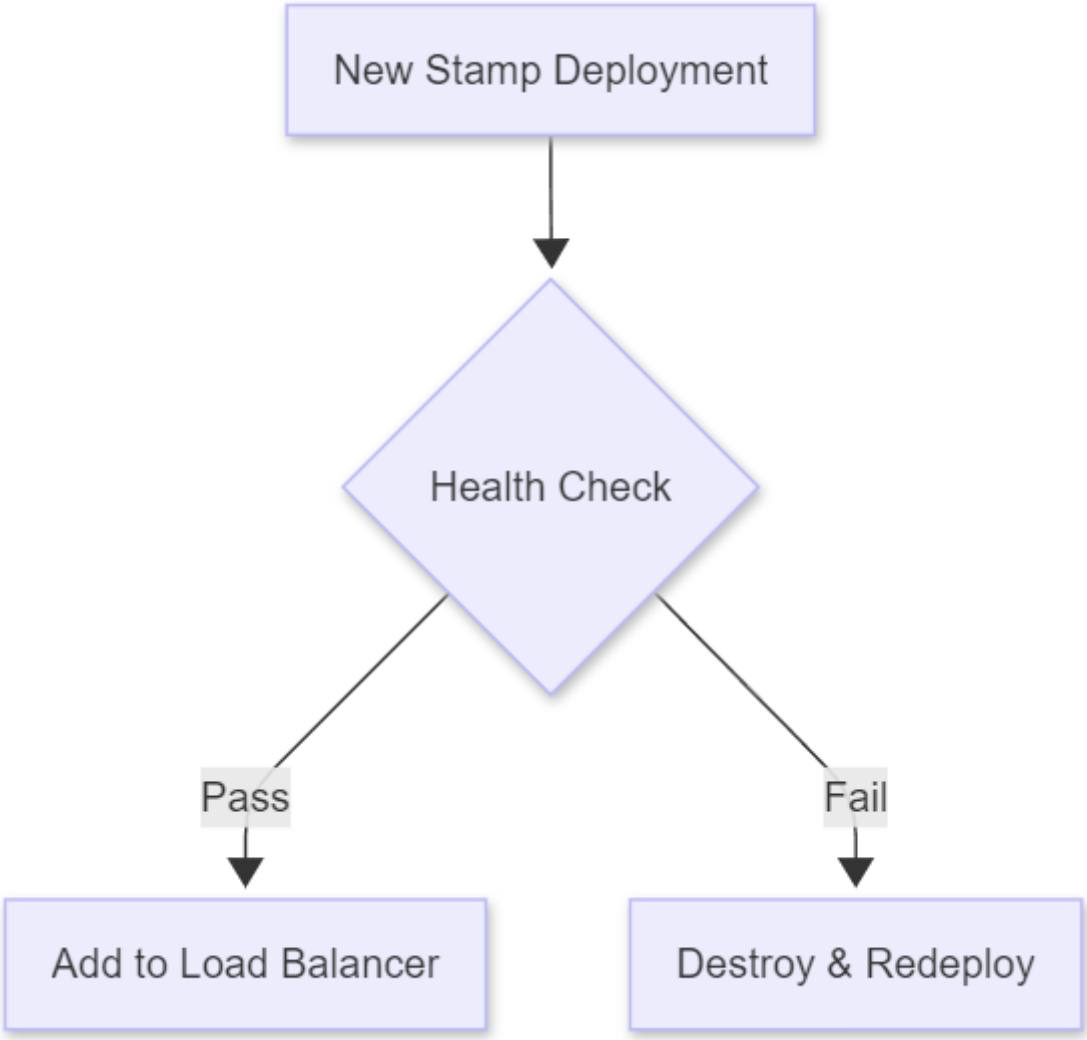
Step 3: Data Synchronization

For cross-stamp consistency:

- Use eventual consistency models for non-critical data
- Implement for real-time collaboration features
- Schedule daily pipelines for reporting database consolidation

Challenges and Mitigations

Challenge	Solution
Data Replication Lag	Hybrid logical clocks for timestamp ordering (Google Spanner paper)
Operational Overhead	Automated healing via Azure Monitor Autoscale
Cost Optimization	Spot instances for non-primary stamps + reserved capacity for core stamps



Comparative Analysis

vs. Blue-Green Deployment

FactorStampBlue-GreenScopeInfrastructure unitApplication versionRollbackStamp decommissioningTraffic re-routingCostHigher (full env copy)Lower (2 app versions)

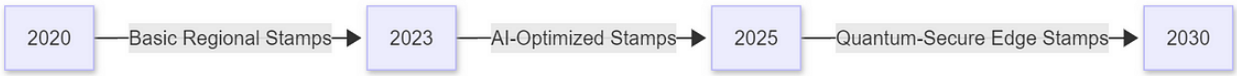
vs. Geode Pattern

While Geode focuses on data replication across nodes, stamps enforce full environment isolation-making them preferable for regulatory compliance over pure performance scenarios.

Future Evolution

Emerging trends reshaping stamp deployments:

1. : Machine learning predicts optimal stamp locations based on traffic patterns
2. : 5G-enabled micro-stamps in cell towers for IoT applications
3. : Immutable ledger tracking cross-stamp transactions for financial audits



The stamp deployment pattern represents a paradigm shift from “scale up” to “scale out” architectures, directly addressing cloud-era challenges in compliance, resiliency, and multi-tenant isolation. As demonstrated by Azure’s large-scale implementations, it provides a template for building systems that are as geographically adaptable as they are technically robust. While complexity demands mature DevOps practices, the tradeoff enables enterprises to truly harness cloud-native potential.

Quick Overview of Deployment Strategies

medium.com/@elouadinouhaila566/quick-overview-of-deployment-strategies-3f6b06231013

Nouhaila El Ouadi October 9, 2024

Image Credit:

When deploying a new feature, the primary objectives are to **maintain continuous operations**, **ensure safety and zero downtime**, and **achieve controlled, low-risk rollouts**. This means that the application should remain fully operational and responsive to users throughout the deployment process, without any interruptions or outages. Additionally, the rollout should be gradual and monitored, allowing for quick identification and resolution of potential issues while minimizing user impact.

Strategies such as Blue/Green Deployment, Canary Deployment, Rolling Deployment, Feature Toggles, and the A/B Testing can be utilized to meet these goals effectively, ensuring a smooth transition to the new feature while maintaining a positive user experience.

Blue/Green Deployment

A deployment strategy that uses two identical environments (Blue and Green). One environment (Blue) is live and serving users, while the other (Green) is updated with the new version. After testing, traffic is switched to the Green environment, allowing for easy rollbacks if needed.

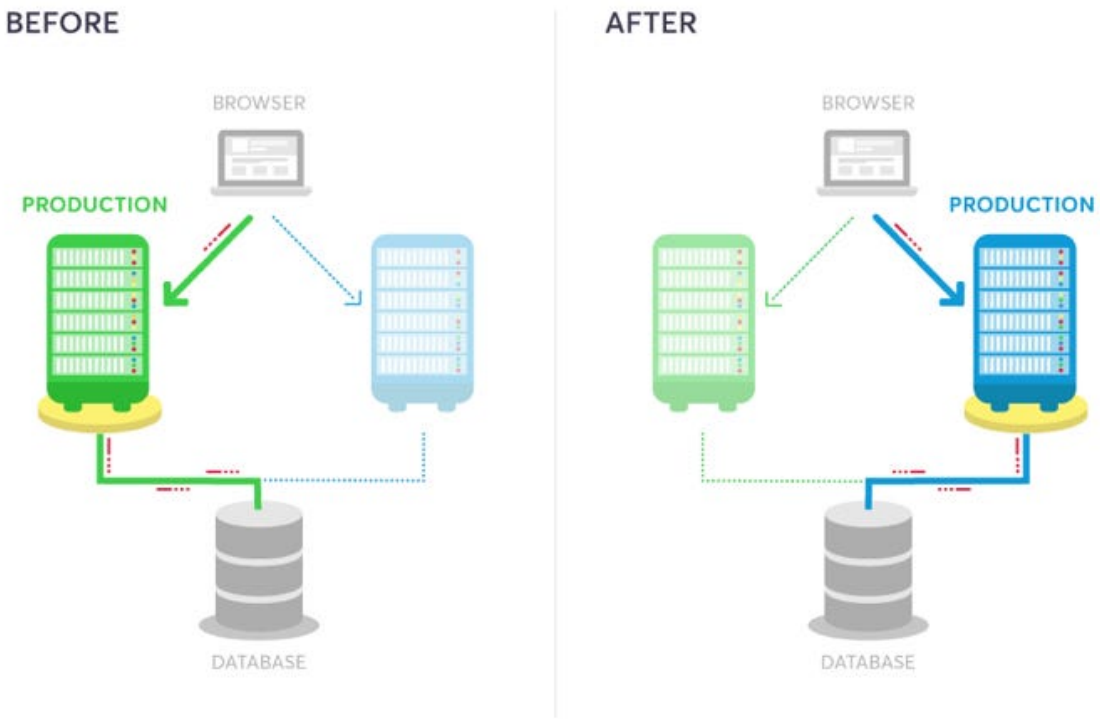


Image Credit:

How It Works:

> Two identical environments (Blue and Green) are maintained. The Blue environment runs the current version, while the Green environment is updated with the new version. After testing, traffic is switched to the Green environment.

When to Use:

> For major releases or significant changes where immediate rollback capability is essential.

Drawbacks:

- Requires twice the infrastructure, which can be costly.

- Complexity in managing and synchronizing two environments.

Objectives Ensured:

> Continuous operations, safety, zero downtime, and low-risk rollouts.

Canary Deployment

A strategy where the new version of an application is deployed to a small subset of users first. This allows for monitoring the new version's performance and gradually rolling it out to all users, minimizing risk.

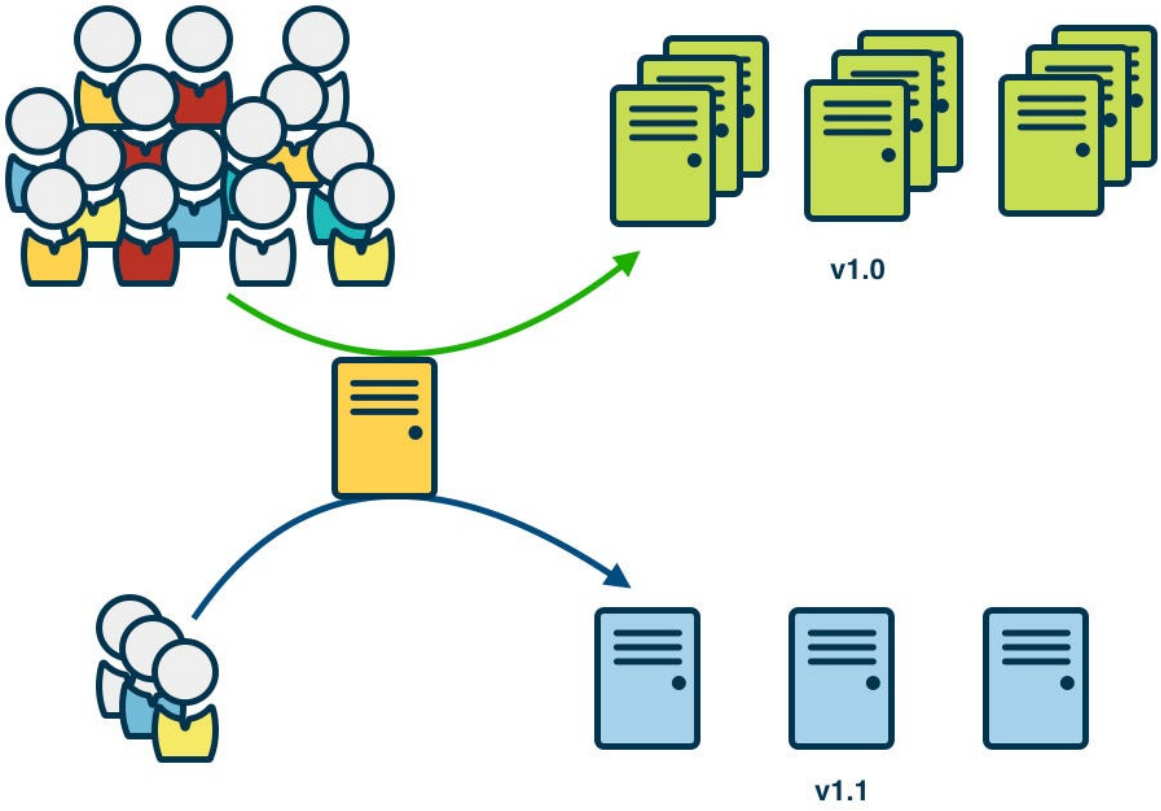


Image Credit:

How It Works:

> The new version is deployed to a small subset of users first. Monitoring is performed to observe the new version's performance before gradually rolling it out to all users.

When to Use:

> When introducing new features or updates that require real-world testing without affecting all users.

Drawbacks:

- Potentially exposes a small number of users to issues if the new version has defects.
- Requires effective monitoring and metrics to evaluate performance.

Objectives Ensured:

Continuous operations, controlled low-risk rollouts, and safety.

A Rolling Deployment is a way to update an application gradually, where a few servers are updated at a time while the rest keep running the old version. This helps avoid downtime.

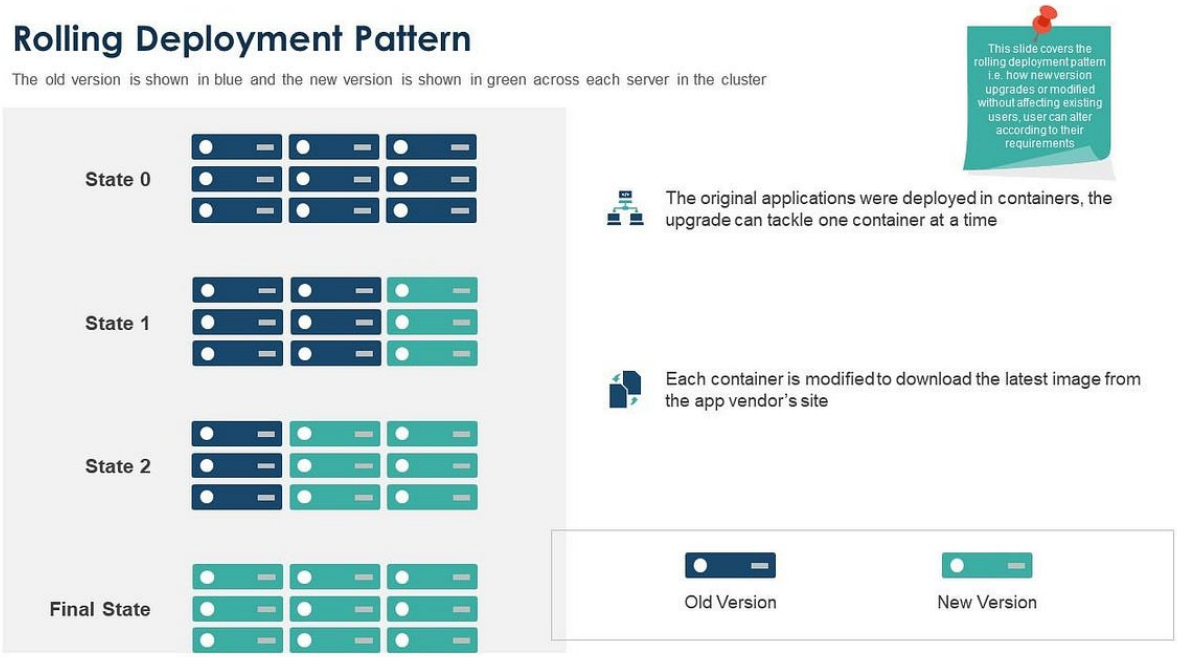


Image Credit:

How It Works:

Updates are applied gradually across multiple servers. A few servers are updated at a time while the rest continue to serve users.

When to Use:

For incremental updates or less critical changes where temporary inconsistencies are acceptable.

Drawbacks:

- May lead to mixed versions in production, causing potential confusion or issues.
- Slightly increased complexity in managing multiple versions.

Objectives Ensured:

Continuous operations and low-risk rollouts, though may not guarantee zero downtime.

Feature Toggles (Feature Flags)

A technique that allows developers to deploy new features in the code but keep them inactive until they are explicitly turned on. This enables testing in production without exposing new features to all users right away.

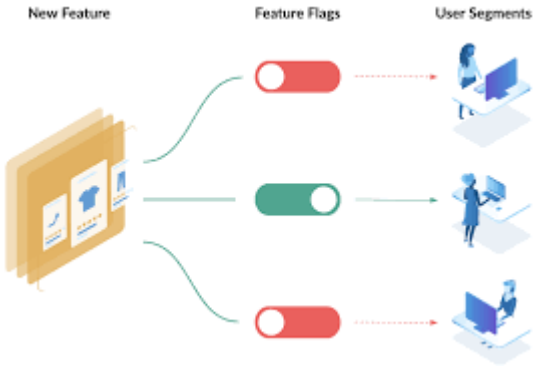


Image Credit:

How It Works:

> New features are deployed in the code but remain inactive until they are explicitly turned on. This allows for testing in production without exposing features to all users.

When to Use:

> For scenarios requiring rapid iterations or controlled releases of new features.

Drawbacks:

- Increases code complexity and maintenance overhead.
- Risk of technical debt if feature flags are not managed properly.

Objectives Ensured:

> Continuous operations, safety, and controlled low-risk rollouts.

A/B Testing

A method of comparing two different versions of an application (Version A and Version B) by serving them to different user groups. This helps determine which version performs better based on user feedback and interaction.

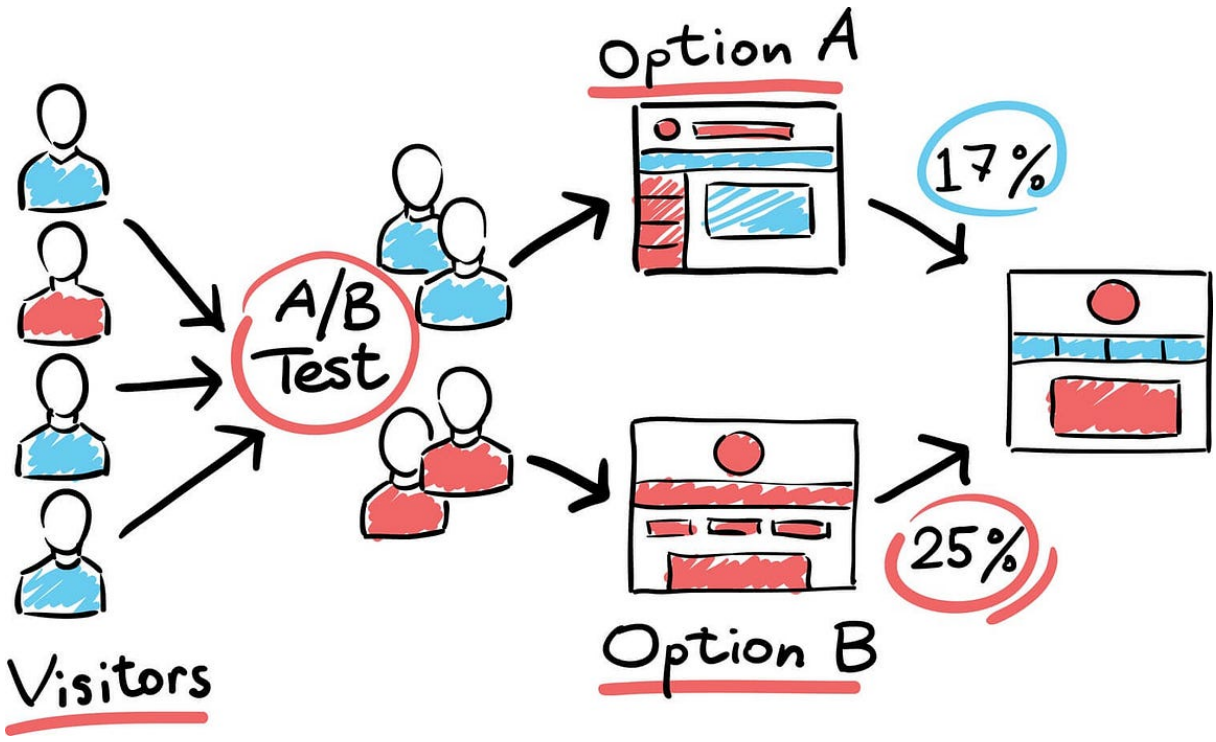


Image Credit:

How It Works:

> Different versions of an application (A and B) are served to different user segments to compare performance, user interaction, and effectiveness, and to see which one performs better.

When to Use:

> When needing to validate new features or changes based on user feedback and performance metrics.

Drawbacks:

- Requires significant user traffic for valid statistical results.
- Potentially introduces inconsistency in user experience.

Objectives Ensured:

> Continuous operations and low-risk rollouts, although not directly focused on zero downtime.

One common drawback across various deployment strategies, such as Blue/Green, Canary, Rolling, and Dark Read, is the challenge of managing **shared services**, particularly databases. This issue can manifest in several ways:

1. : Changes to the database schema may be required when introducing new features, leading to compatibility issues between old and new application versions.

2. : Introducing breaking changes can complicate rollbacks or transitions, as new versions may expect database structures that older versions do not recognize, resulting in errors.
3. : Migrating existing data to align with a new schema can be complex and error-prone, particularly when the application is still running.
4. : Multiple versions accessing the same database can lead to race conditions or data inconsistencies if they make simultaneous changes.
5. : Running different application versions that share a database can increase load and potentially degrade performance.

Mitigation Strategies

To address these challenges, organizations can implement strategies such as ensuring backward compatibility, utilizing feature toggles, employing database versioning, executing shadow writes, and preparing well-tested migration scripts. By effectively managing shared services, teams can enhance the safety and reliability of their deployment processes while minimizing risks associated with updates and new feature releases.

Each deployment strategy serves different purposes and has unique advantages and drawbacks. The choice of strategy depends on the specific needs of the organization, the criticality of the changes being deployed, and the objectives that need to be met during the deployment process.

I’d love to hear your thoughts! Share your experiences or insights on deployment strategies in the comments below, and let’s create a community of learning together!

Here are some valuable resources and further readings:

What are the most popular deployment patterns? | Nikki Siapno posted on the topic | LinkedIn

www.linkedin.com

Yan Cui on LinkedIn: Everyone knows Canary Deployments, but do you know the Dark Read pattern?... |…

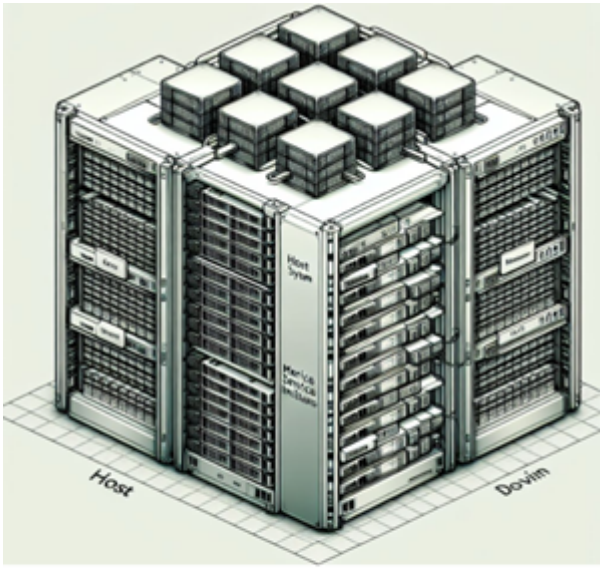
www.linkedin.com

Microservice & RESTFul API deployment strategies

medium.com/@mayank_sh/microservice-restful-api-deployment-strategies-e92d9a36e6ef

Mayank Shrivastava June 2, 2024

Multiple service instances per host



Multiple Service Instances per Host

This pattern involves deploying multiple instances of different microservices on a single host. This can be done using virtual machines or containers.

Pros: Efficient resource utilization and simplified deployment.

Cons: Tighter coupling between services and potential for conflicts.

Single service instance per host

Single Service Instance per Host

Each microservice instance runs on its own host, which can be a virtual machine or a container.

Pros: Strong isolation and easier to scale individual services.

Cons: Can be more expensive due to underutilized resources.

Service instance per container

Service Instance per Container

Each microservice instance runs in its own container, providing a lightweight form of virtualization.

Pros: High scalability, isolation, and consistent environments across development, testing, and production.

Cons: Requires container orchestration and management tools.

Serverless deployment pattern

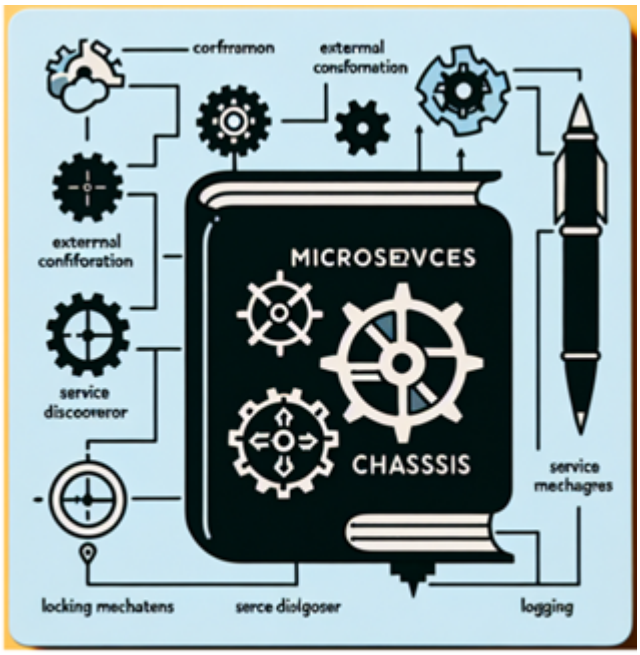
Service Instance per Container

Microservices are deployed as functions that are triggered by events, and the cloud provider dynamically manages the allocation of machine resources.

Pros: No need to manage servers, potentially lower costs, and high scalability.

Cons: Vendor lock-in, potential for increased latency, and cold start issues.

Microservice chassis pattern



Microservice Chassis Pattern

This pattern involves using a framework that provides the necessary infrastructure and cross-cutting concerns for microservices, such as logging, monitoring, and security.

Pros: Reduces the development effort required for each microservice.

Cons: Can lead to dependency on the framework and reduced flexibility.

Externalized configuration pattern

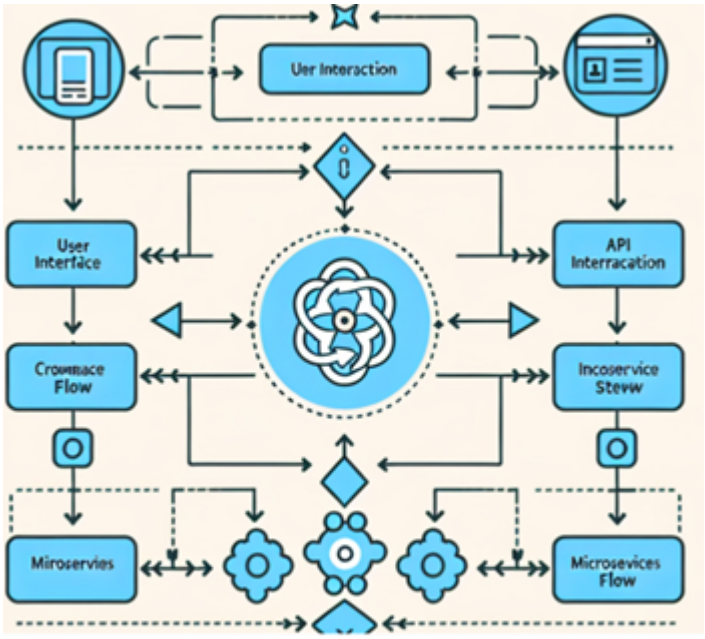
Externalized Configuration Pattern

Configuration settings are stored outside the microservice package, often in a centralized configuration server.

Pros: Simplifies the management of environment-specific configurations and supports dynamic configuration updates.

Cons: Adds complexity and a potential single point of failure.

API Gateway pattern



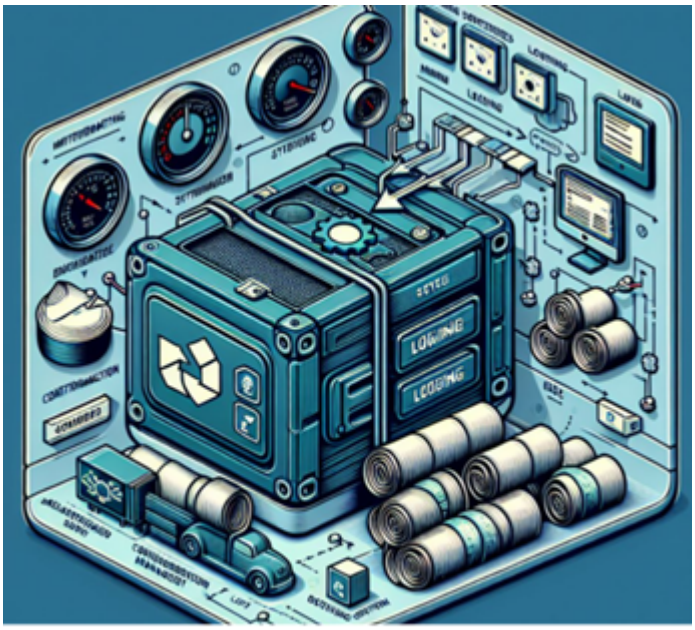
API Gateway Pattern

An API gateway is used to manage requests to microservices, providing a single-entry point for clients.

Pros: Simplifies client interactions with microservices, provides security and monitoring.

Cons: Can become a bottleneck if not properly managed and adds an additional layer to maintain.

Sidecar pattern



Sidecar Pattern

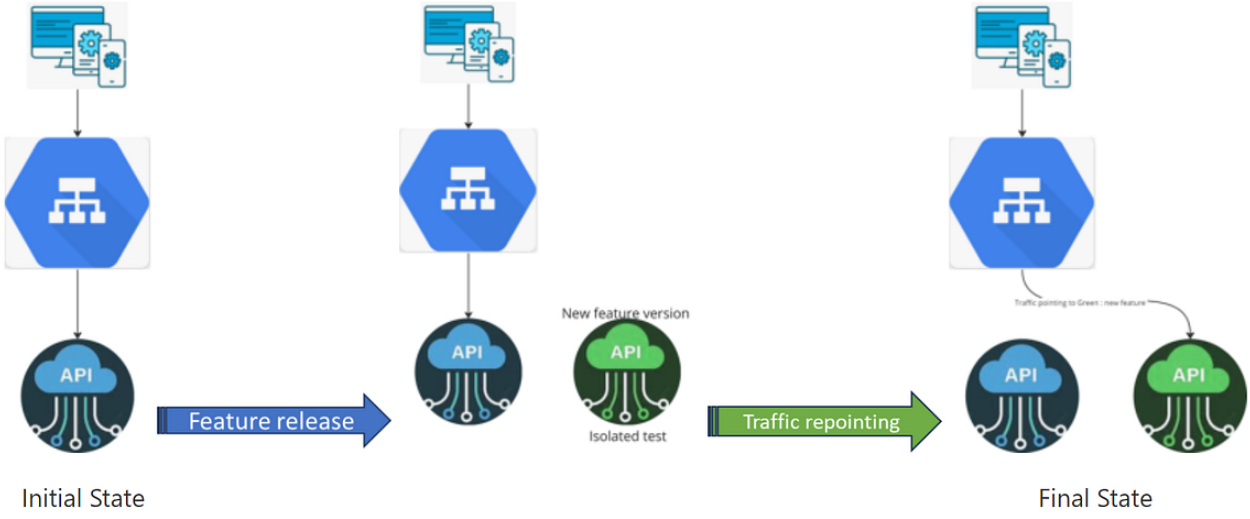
Each microservice is deployed with a sidecar — a separate process or container that provides additional functionality, such as monitoring, logging, or configuration management.

Pros: Allows for separation of concerns and reusable components.

Cons: Additional overhead and complexity.

Restful API deployment pattern

Blue Green deployment



Steps typically involved in a blue-green deployment:

> Initial Setup:

Blue Environment: Your current live production environment where the API is actively serving users.

Green Environment: An identical but separate environment which is not yet serving users.

> Preparation:

Deploy the new version of the API to the Green environment.

Test the Green environment to ensure the new API version functions correctly.

> Routing:

Configure your router or load balancer to switch traffic from the Blue environment to the Green environment.

Perform the switch, which should be a fast and easily reversible operation.

> Monitoring:

After the switch, closely monitor the Green environment for any issues.

If problems arise, you can quickly route traffic back to the Blue environment.

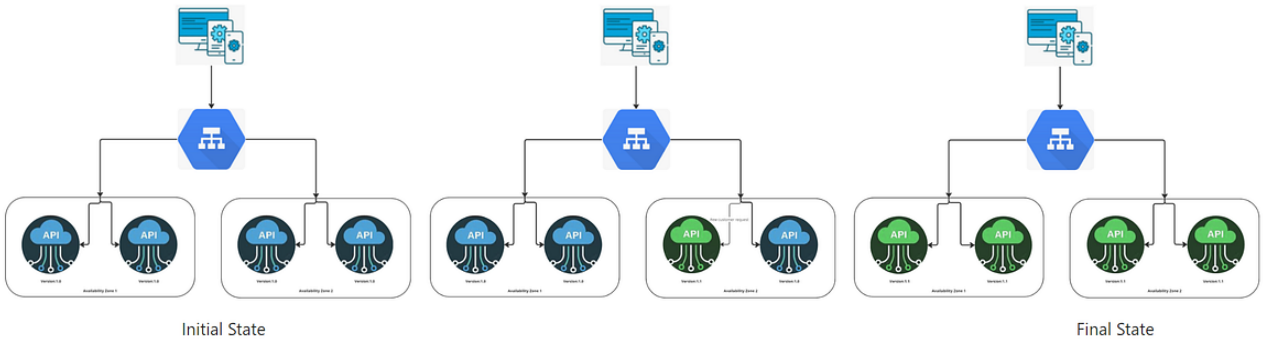
> Decommission:

If the Green environment is stable, the Blue environment can be decommissioned or repurposed as the new staging area for the next release.

> Cleanup:

Perform any necessary cleanup tasks such as clearing old logs, releasing resources, etc.

Canary deployment



Steps involved in a canary deployment for an API:

> Preparation:

Ensure that your infrastructure supports canary deployments, with the ability to divert a subset of traffic to different service versions.

Deploy the new version of your API to a subset of your infrastructure, i.e., the “canary” servers, while the old version remains running on the main servers.

> Routing a Subset of Traffic:

Configure your load balancer or router to divert a small percentage of live traffic to the new API version on the canary servers.

This percentage can be adjusted based on your confidence in the new release and your traffic patterns.

> Monitoring and Testing:

Monitor the performance and error rates of the canary deployment closely.

Use real-time monitoring and observability tools to detect any issues.

You may also want to run automated tests against the canary deployment.

> Feedback Evaluation:

Evaluate user feedback and system metrics to determine the success of the new release.

If any issues are detected, roll back by routing all traffic back to the old API version.

> Incremental Rollout:

If the canary is performing well, gradually increase the percentage of traffic that it handles.

Continue monitoring and testing, and if no issues arise, keep increasing the traffic until the canary is handling 100%.

> Full Rollout:

Once the canary version is handling all the traffic successfully, and you are confident that it is stable, you can roll out the new version to all servers.

Replace the old API version with the new one across your entire infrastructure.

> Decommissioning:

After the new API version is fully rolled out and the old version is no longer receiving traffic, decommission the old servers.

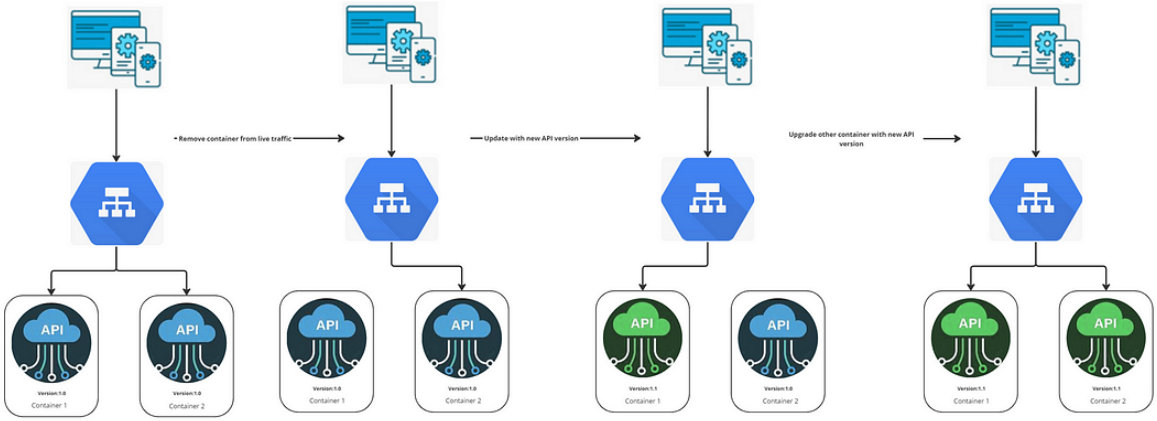
Clean up any resources that are no longer needed.

> Post-Deployment:

Document any lessons learned and update your deployment procedures accordingly.

Use the insights gained from monitoring to improve the API for future releases.

Rolling Deployment (or Rolling Update)



Steps involved in executing a rolling deployment:

> Preparation:

Ensure the new API version is fully tested and ready for deployment.

Confirm that the deployment environment supports rolling updates, typically through orchestration tools like Kubernetes, Docker Swarm, or cloud service providers.

> Initial Deployment:

Deploy the new version to a single instance or a small batch of instances, depending on the total number of instances available.

> Traffic Management:

Configure your load balancer or traffic manager to stop routing new requests to the instances being updated.

> Monitoring:

Monitor the new version for stability, performance, and errors. Ensure that the service continues to operate seamlessly for users.

> Gradual Rollout:

If the new instances are stable, continue the rolling update by incrementally replacing additional instances with the new version.

Continue to divert traffic away from instances that are about to be updated.

> Validation:

After each batch of instances is updated, perform health checks and validations to ensure that the new version is functioning as expected.

> Full Rollout:

Proceed with the rolling update until all instances are running the new version of the API.

At each step, ensure that enough capacity is available to handle the load.

> Rollback Plan:

Have a rollback plan in place in case any issues arise during the deployment.

If an issue is detected, you can stop the rollout and revert to the previous version by deploying it to the instances that were already updated.

> Decommissioning:

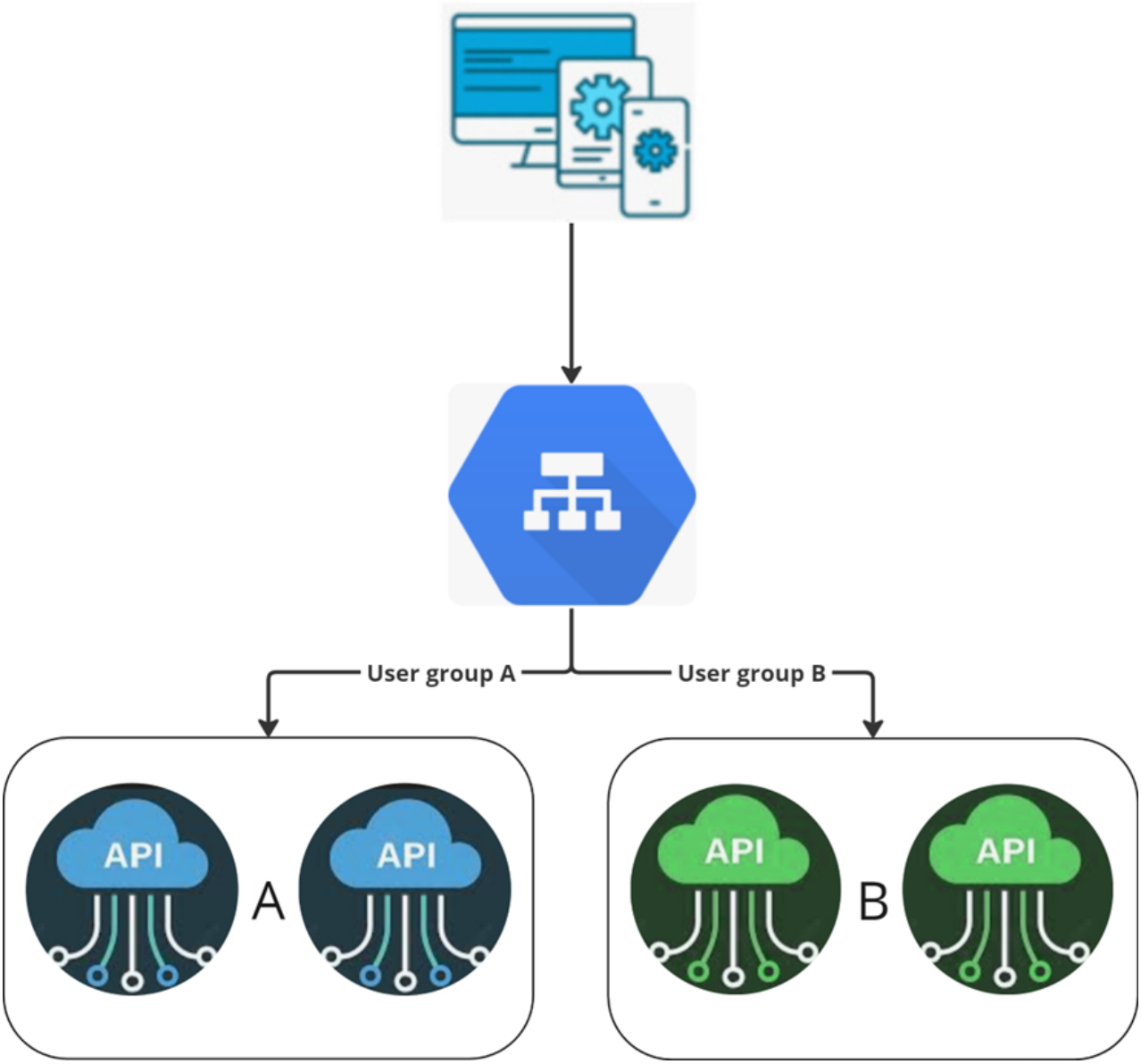
Once the new version is fully deployed and stable, decommission any remaining instances of the old version.

> Post-Deployment:

Conduct a post-deployment review to document any issues or improvements for future deployments.

Update any deployment scripts or configurations as needed based on the lessons learned.

A/B Testing Deployment



Steps for conducting an A/B testing deployment for an API:

> Preparation:

Ensure both versions of the API are ready for deployment.

Version A is your current production version, and version B is the new version you want to test.

Set up a mechanism to segment and route traffic to different API versions.

This can be done using a load balancer, API gateway, or a service mesh.

> Define Metrics:

Determine the success metrics for the test.

These could include response times, error rates, resource utilization, user engagement, or businessspecific KPIs.

> Segment Users:

Define criteria for how users will be segmented (e.g., geographically, by user type, randomly, etc.).

Ensure that the user experience for each segment will remain consistent throughout the test.

> Traffic Distribution:

Decide on the percentage of traffic that will be routed to each version of the API.

Typically, a smaller percentage is routed to the new version to minimize impact.

> Deployment:

Deploy version B of the API to the production environment without disrupting version A.

> Routing:

Configure the routing mechanism to direct the specified traffic to each API version according to the A/B test plan.

> Monitoring and Data Collection:

Collect data on the defined metrics for both versions of the API.

Monitor the performance and stability of both versions during the test.

> Analysis:

Analyze the collected data to evaluate the performance of each version.

Determine if there is a statistically significant difference in the metrics.

> Evaluation:

Based on the analysis, decide if version B is a clear winner, if the test is inconclusive, or if version A remains the best option.

Consider user feedback if applicable.

> Implementation or Rollback:

If version B is superior, gradually route more traffic to it until it becomes the new production version (and version A is decommissioned).

If version A is still the best option, route all traffic back to version A and decommission version B.

> Review:

Document the results of the A/B test, including any lessons learned or insights gained.

Use this information to inform future development and deployment strategies.

Select your strategy

> Blue-Green Deployment

Pros:

Minimal downtime during deployments.

Easy rollback to the previous version if issues arise.

Simplifies the switch between versions by just changing the router’s direction.

Cons:

Requires double the production environment, increasing costs.

Not suitable for database schema changes that are not backward compatible.

> Canary Deployment

Pros:

Allows gradual rollout to a subset of users.

Minimizes the impact of errors on the entire user base.

Useful for testing in a production environment with real traffic.

Cons:

More complex to implement and manage.

Rollback can be difficult if errors are detected late in the process.

> Rolling Deployment (or Rolling Update)

Pros:

Updates application instances incrementally, which keeps the service available during deployment.

Does not require double the infrastructure.

Cons:

If not enough instances, the capacity might be temporarily reduced.

Rollbacks can be tricky, especially if the deployment has progressed significantly.

> A/B Testing Deployment

Pros:

Good for testing new features with actual users before a full rollout.

Data-driven decision-making about the new features based on user feedback.

Cons:

Requires sophisticated monitoring and analytics tools.

Can be complex to route different users to different service versions.