Intro to deployment strategies: blue-green, canary, and more

Jason Skowronski Nov 21 '18 · 7 min read

#devops #webdev #monitoring

These days, the biggest change to software development is the frequency of deployments. Product teams deploy releases to production earlier (and more often). Months or years-long release cycles are becoming rare—especially among those building pure software products.

Today, using a service-oriented architecture and microservices approach, developers can design a code base to be modular. This allows them to write and deploy changes to different parts of the code base simultaneously.

The business benefits of shorter deployment cycles are clear:

- Time-to-market is reduced
- Customers get product value in less time
- Customer feedback also flows back into the product team faster, which means the team can iterate on features and fix problems faster
- Overall developer morale goes up

However, this shift also creates new challenges for the operations or DevOps team. With more frequent deployments, it's more likely that the deployed code could negatively affect site reliability or customer experience. That's why it's important to develop strategies for deploying code that minimize risk to the product and customers.

In this article, we'll talk about a few different deployment strategies, best practices, and tools that will allow your team to work faster *and* more reliably.

Challenges of Modern Applications

Modern applications are often distributed and cloud-based. They can scale elastically to meet demand, and are more resilient to failure thanks to highly-available architectures. They may utilize fully managed services like AWS Lambda or Elastic Container Service (ECS) where the platform handles some of the operational responsibility.

These applications almost always have frequent deployments. For example, a mobile application or an consumer web

application may undergo several changes within a month. Some are even deployed to production multiple times a day.

They often use microservice architectures in which several components work together to deliver full functionality. There can be different release cycles for different components, but they all have to work together seamlessly.

The increased number of moving parts mean more chances for something to go wrong. With multiple development teams making changes throughout the codebase, it can be difficult to determine the root cause of a problem when one inevitably occurs.

Another challenge: the abstraction of the infrastructure layer, which is now considered code. Deployment of a new application may require the deployment of new infrastructure code as well.

Popular Deployment Strategies

To meet these challenges, application and infrastructure teams should devise and adopt a deployment strategy suitable for their use case.

We will review several and discuss the pros and cons of several different deployment strategies so you can choose which one suits your organization.

"Big Bang" Deployment

As the name suggests, "big bang" deployments update whole or large parts of an application in one fell swoop. This strategy goes back to the days when software was released on physical media and installed by the customer.

Big bang deployments required the business to conduct extensive development and testing before release, often associated with the "waterfall model" of large sequential releases.

Modern applications have the advantage of updating regularly and automatically on either the client side or the server side. That makes the big bang approach slower and less agile for modern teams.

Characteristics of big bang deployment include:

- All major pieces packaged in one deployment;
- Largely or completely replacing an existing software version with a new one;
- Deployment usually resulting in long development and testing cycles;
- Assuming a minimal chance of failure as rollbacks may be impossible or impractical;
- Completion times are usually long and can take multiple teams' efforts;
- Requiring action from clients to update the client-side installation.

Big bang deployments aren't suitable for modern applications because the risks are unacceptable for public-facing or business-critical applications where outages mean huge financial loss. Rollbacks are often costly, time-consuming, or even impossible.

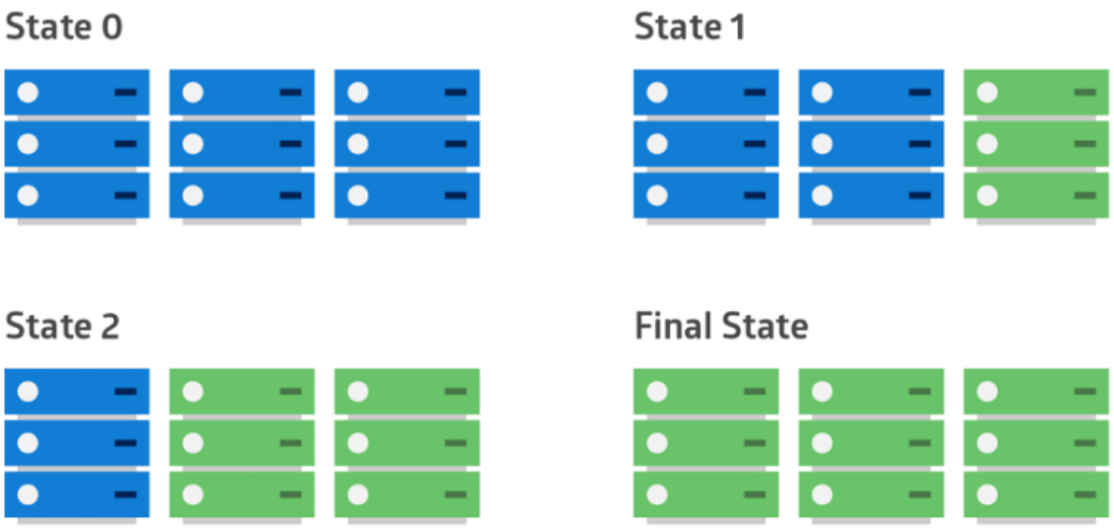
The big bang approach can be suitable for non-production systems (e.g., re-creating a development environment) or vendor-packaged solutions like desktop applications.

Rolling Deployment

Rolling, phased, or step deployments are better than big bang deployments because they minimize many of the associated risks, including user-facing downtime without easy rollbacks.

In a rolling deployment, an application's new version gradually replaces the old one. The actual deployment happens over a period of time. During that time, new and old versions will coexist without affecting functionality or user experience. This process makes it easier to roll back any new component incompatible with the old components.

The following diagram shows the deployment pattern: the old version is shown in blue and the new version is shown in green across each server in the cluster.



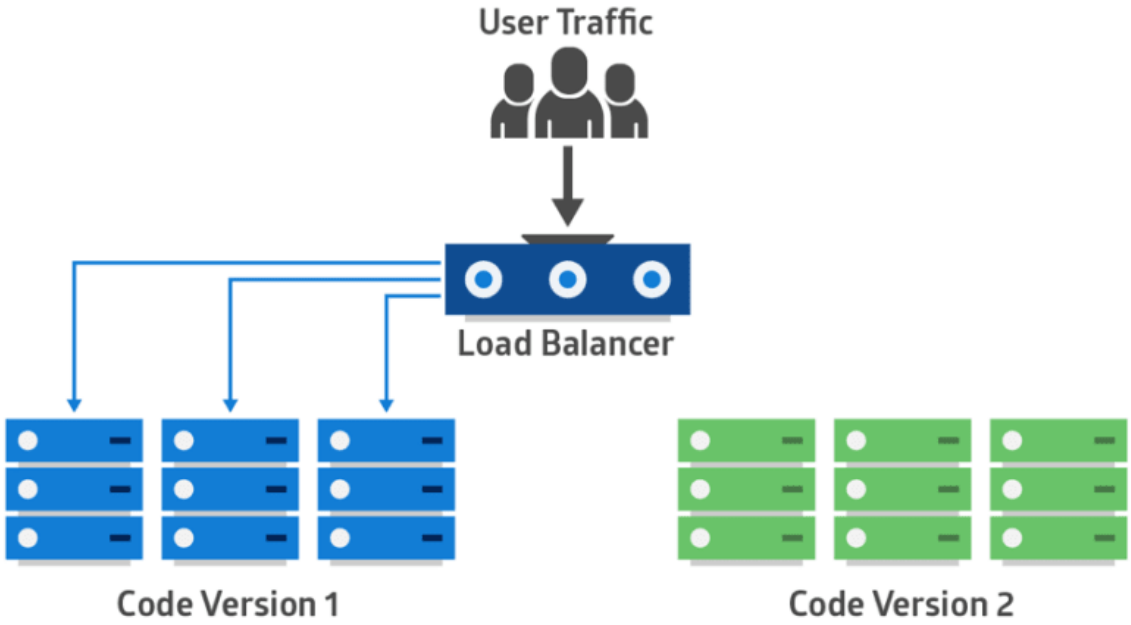
An application suite upgrade is an example of a rolling deployment. If the original applications were deployed in containers, the upgrade can tackle one container at a time. Each container is modified to download the latest image from the app vendor's site. If there is a compatibility issue for one of the apps, the older image can recreate the container. In this case, the new and old versions of the suite's applications coexist until every app is upgraded.

Blue-Green, Red-Black or A/B Deployment

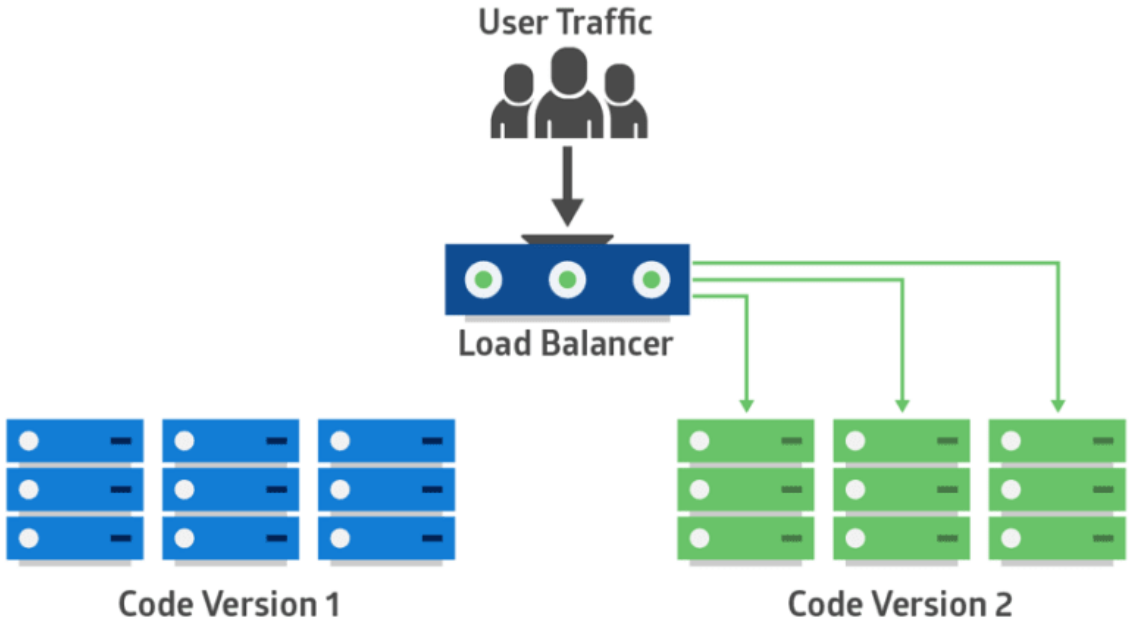
This is another fail-safe process. In this method, two identical production environments work in parallel.

One is the currently-running production environment receiving all user traffic (depicted as Blue). The other is a clone of it, but

idle (Green). Both use the same database back-end and app configuration:



The new version of the application is deployed in the green environment and tested for functionality and performance. Once the testing results are successful, application traffic is routed from blue to green. Green then becomes the new production.



If there is an issue after green becomes live, traffic can be routed back to blue.

In a blue-green deployment, both systems use the same persistence layer or database back end. It's essential to keep the application data in sync, but a mirrored database can help achieve that.

You can use the primary database by blue for write operations and use the secondary by green for read operations. During switchover from blue to green, the database is failed over from primary to secondary. If green also needs to write data during testing, the databases can be in bidirectional replication.

Once green becomes live, you can shut down or recycle the old blue instances. You might deploy a newer version on those instances and make them the new green for the next release.

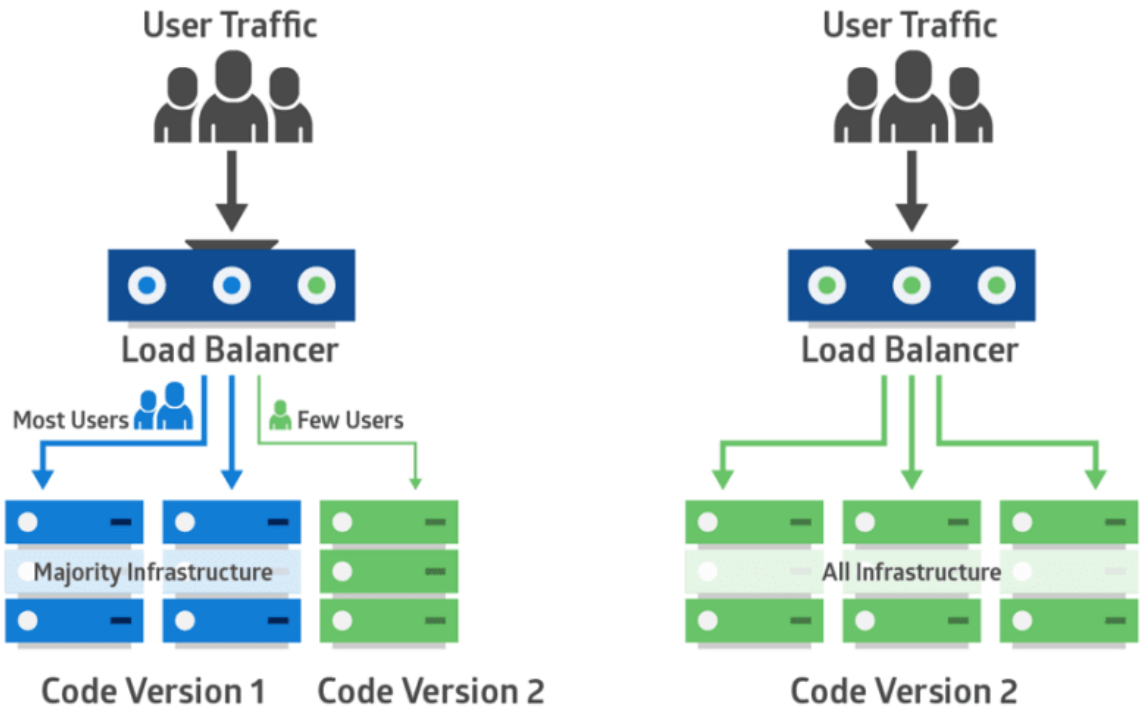
Blue-green deployments rely on traffic routing. This can be done by updating DNS CNAMES for hosts. However, long TTL values can delay these changes. Alternatively, you can change the load balancer settings so the changes take effect immediately. Features like connection draining in ELB can be used to serve in-flight connections.

Canary Deployment

Canary deployment is like blue-green, except it's more risk-averse. Instead of switching from blue to green in one step, you use a phased approach.

With canary deployment, you deploy a new application code in a small part of the production infrastructure. Once the application is signed off for release, only a few users are routed to it. This minimizes any impact.

With no errors reported, the new version can gradually roll out to the rest of the infrastructure. The image below demonstrates canary deployment:



The main challenge of canary deployment is to devise a way to route some users to the new application. Also, some applications may always need the same group of users for testing, while others may require a different group every time.

Consider a way to route new users by exploring several techniques:

- Exposing internal users to the canary deployment before allowing external user access;
- Basing routing on the source IP range;
- Releasing the application in specific geographic regions;
- Using an application logic to unlock new features to specific users and groups. This logic is removed when the application goes live for the rest of the users.

Deployment Best Practices

Modern application teams can follow a number of best practices to keep deployment risks to a minimum:

- **Use a deployment checklist.** For example, an item on the checklist may be to "backup all databases only after app services have been stopped" to prevent data corruption.
- **Adopt Continuous Integration (CI).** CI ensures code checked into the feature branch of a code repository merges with its main branch only *after* it has gone through a series of dependency checks, unit and integration tests, and a successful build. If there are errors along the path, the build fails and the app team is notified. Using CI therefore means every change to the application is tested before it can be deployed. Examples of CI tools include: CircleCI, Jenkins.
- **Adopt Continuous Delivery (CD).** With CD, the CI-built code artifact is packaged and always ready to be deployed in one or more environments. Read more in our Low-Risk Continuous Delivery eBook.
- **Use standard operating environments (SOEs)** to ensure environment consistency. You can use tools like Vagrant and Packer for development workstations and servers.
- **Use Build Automation tools to automate environment builds.** With these tools, it's often simple to click a button to tear down an entire infrastructure stack and rebuild from scratch. An example of such tools is CloudFormation.

- **Use configuration management tools** like Puppet, Chef, or Ansible in target servers to automatically apply OS settings, apply patches, or install software
- **Use communication channels** like Slack for automated notifications of unsuccessful builds and application failures
- **Create a process for alerting the responsible team on deployments that fail.** Ideally you'll catch these in the CI environment, but if the changes are deployed you'll need a way to notify the responsible team
- **Enable automated rollbacks for deployments** that fail health checks, whether due to availability or error rate issues.

Post-Deployment Monitoring

Even after you adopt all of those best practices, things may still fall through the crack. Because of that, monitoring for issues occurring immediately after a deployment is as important as planning and executing a perfect deployment.

An application performance monitoring (APM) tool can help your team monitor critical performance metrics including server response times after deployments. Changes in application or system architecture can dramatically affect application performance.

An error-monitoring solution like Rollbar is equally essential. It will quickly notify your team of new or reactivated errors from

a deployment that could uncover important bugs requiring immediate attention.

Without an error monitoring tool, the bugs may never have been discovered. While a few users who encounter the bugs will take the time to report them, most others don't. The negative customer experience can lead to satisfaction issues over time, or worse, prevent business transactions from taking place now.

An error monitoring tool also creates a shared visibility of all the post-deployment issues among Operations / DevOps teams and developers. This shared understanding allows the teams to be more collaborative and responsive.

Originally published on rollbar.com

Jason Skowronski + FOLLOW
I'm the founder of Dev Spotlight. We focus on creating great content for Devs and DevOps.
@mostlyjason mostlyjason www.devspotlight.com