Automating safe, hands-off deployments

Clare Liguori

When I interviewed for my job at Amazon, I made sure to ask one of the interviewers, “How often do you deploy to production?” At the time, I was working on a product that rolled out a major release once or twice a year, but sometimes I needed to release a small fix in between big releases. For each fix that I released, I spent hours carefully rolling it out. Then I frantically checked logs and metrics to see if I had broken anything after the deployment and needed to roll it back.

I read that Amazon practiced continuous deployment, so when I interviewed, I wanted to know how much time I would have to spend managing and watching deployments as a developer at Amazon. The interviewer told me that changes were automatically deployed to production multiple times a day by continuous deployment pipelines. When I asked how much of his day was spent carefully shepherding each of those deployments and watching logs and metrics for any impact as I had been doing, he told me usually none. Because the pipelines did this work for his team, most deployments weren’t actively watched by anyone. “Whoa!” I said. After I joined Amazon, I was excited to find out exactly how these “hands-off” automated deployments worked.

How safe continuous deployment frees up developer time

Since then, I’ve seen first-hand the way that Amazon sets up continuous deployment pipelines to help us deploy quickly and safely. I came to appreciate how our continuous deployment safety practices free up developer time from work on deployments. When I push production code into the main branch of my service’s source code repository, usually I forget about it and go on to my next task, while my team’s pipeline takes over getting that change into production. The release of my code change to a production service is fully automated by the pipeline, which means that the last time I or any other developer touches or reviews a piece of code is when it is merged into the source code repository.

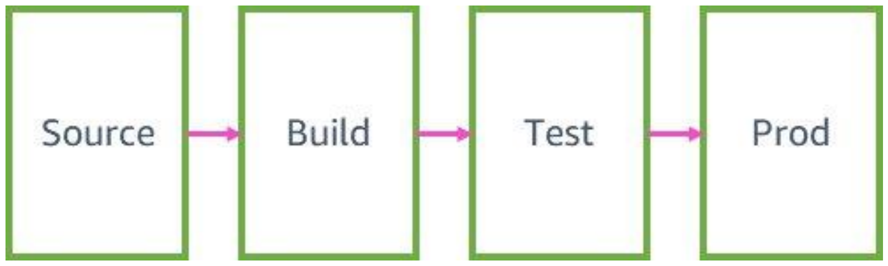
My team set up that pipeline with automated steps that deploy our changes safely to production so we don’t have to watch each deployment. The pipeline runs the latest changes through a set of tests and deployment safety checks. These automated steps prevent customer-impacting defects from reaching production and limit the impact of defects on customers if they do reach production. As a developer, I’m able to trust that the pipeline will cautiously and safely deploy my change to production for me, without the need for me to actively watch it.

The journey to continuous delivery

Amazon didn’t start out practicing continuous delivery, and developers here used to spend hours and days managing deployments of their code to production. We adopted continuous delivery across the company as a way to automate and standardize how we deployed software and to reduce the time it took for changes to reach production. Improvements to our release process built up incrementally over time. We identified deployment risks and found ways to mitigate those risks through new safety automation in pipelines. We continue to iterate on the release process by identifying new risks and new ways of improving deployment safety. To learn more about our journey to continuous delivery and how we continue to improve, see the Builders’ Library article Going faster with continuous delivery.

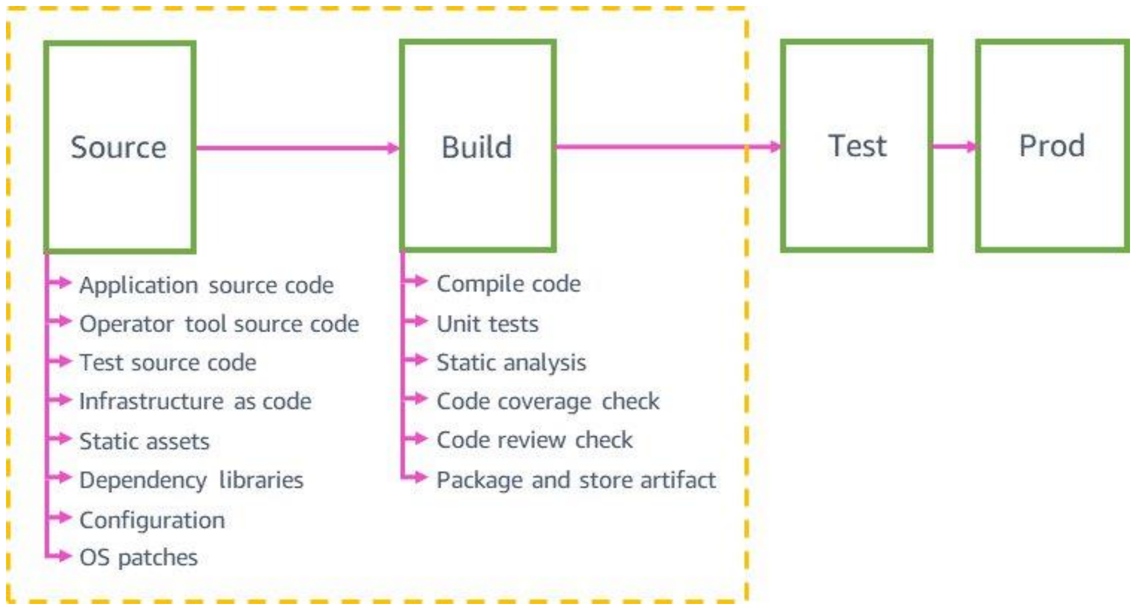
The four pipeline phases

In this article, we walk through the steps a code change goes through in a pipeline at Amazon on its way to production. A typical continuous delivery pipeline has four major phases—source, build, test, and production (prod). We'll dive into the details of what happens in each of these pipeline phases for a typical AWS service, and provide you with an example of how a typical AWS service team might set up one of their pipelines.



Source and build

The following diagram gives you an overview of the sources and build steps you might find in typical AWS service team pipelines.



Pipeline sources

Pipelines at Amazon automatically validate and safely deploy any type of source change to production, not only changes to application code. They can validate and deploy changes to sources such as website static assets, tools, tests, infrastructure, configuration, and the application's underlying operation system (OS). All of these changes are version controlled in individual source code repositories. The source code dependencies, such as libraries, programming languages, and parameters like AMI IDs, are automatically upgraded to the latest version at least weekly.

These sources are deployed in individual pipelines with the same safety mechanisms (like automatic rollback) that we use for deploying application code. For example, configuration values for a service that can change at runtime (like API rate limit increases and feature flags) are automatically deployed in a dedicated configuration pipeline. Source changes are automatically rolled back if they cause any issues in production for the service (such as failures to parse a configuration file).

A typical microservice might have an application code pipeline, an infrastructure pipeline, an OS patching pipeline, a configuration/feature flags pipeline, and an operator tools pipeline. Having multiple pipelines for the same microservice helps us deploy changes to production faster. Application code changes that fail integration tests and block the application pipeline don't affect other pipelines. For example, they don't block infrastructure code changes from reaching production in the infrastructure pipeline. All the pipelines for the same microservice tend to look very similar. For example, a feature flags pipeline uses the same safe deployment techniques as the application code pipeline, because a bad feature flag configuration change can have an impact on production just as a bad application code change can.

Code review

All changes going to production start with a code review and must be approved by a team member before merging into the *mainline* branch (our version of "master" or "trunk"), which automatically starts the pipeline. The pipeline enforces the requirement that all commits on the mainline branch must be code reviewed and approved by a member of the service team for that pipeline. The pipeline will block any unreviewed commits from being deployed.

With fully automated pipelines, the code review is the last manual review and approval that a code change receives from an engineer before being deployed to production, so this is a critical step. Code reviewers evaluate the code's correctness and also evaluate whether the change can be safely deployed to production. They evaluate whether the code has sufficient tests (unit tests, integration tests, and canary tests), whether it is sufficiently instrumented for deployment monitoring, and whether it can be safely rolled back. Some teams use a custom checklist like the one in the following sample, which is automatically added to each of the team's code reviews to explicitly check for deployment safety concerns.

Example code review checklist

~~~
## Testing
[ ] Did you write new unit tests for this change?
[ ] Did you write new integration tests for this change?

Include the test commands you ran locally to test this change:
```
mvn test && mvn verify
```
~~~

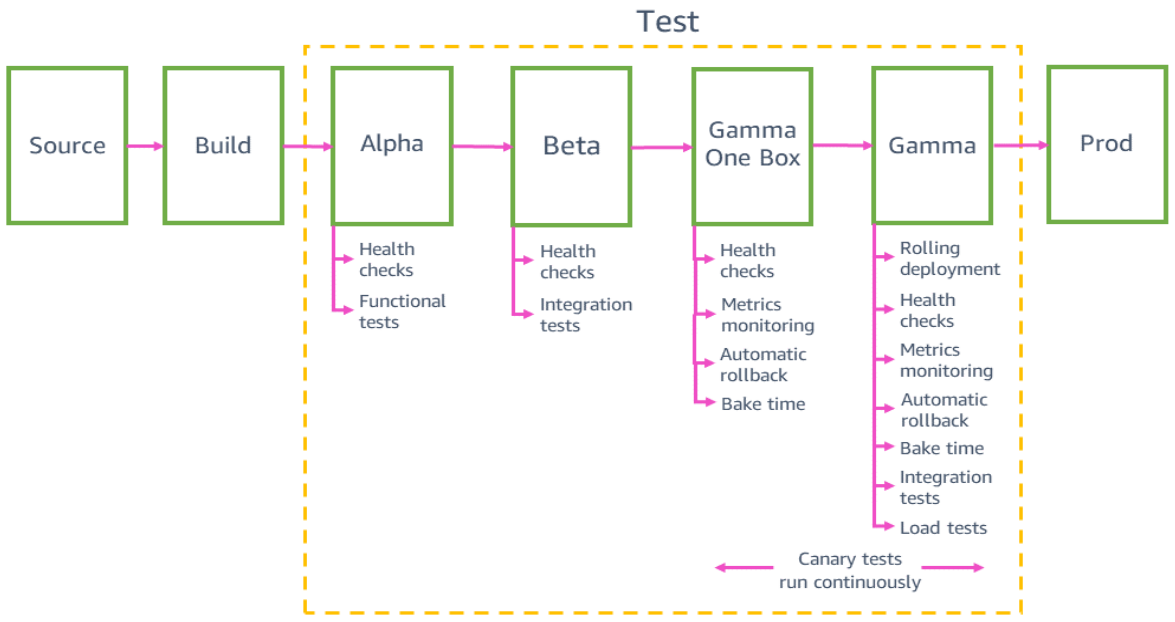
```
## Monitoring
[ ] Will this change be covered by our existing monitoring?
 (no new canaries/metrics/dashboards/alarms are required)
[ ] Will this change have no (or positive) effect on resources and/or
limits?
 (including CPU, memory, AWS resources, calls to other services)
[ ] Can this change be deployed to Prod without triggering any alarms?

## Rollout
[ ] Can this change be merged immediately into the pipeline upon approval?
[ ] Are all dependent changes already deployed to Prod?
[ ] Can this change be rolled back without any issues after deployment to
Prod?
```

Build and unit tests

In the build stage, the code is compiled and unit tested. The build tools and build logic can vary from language to language and even from team to team. For example, teams can choose the unit test frameworks, linters, and static analysis tools that work best for them. In addition, teams can choose the configuration of those tools, such as the minimum acceptable code coverage in their unit test framework. The tools and types of tests that run will also vary depending on the type of code that is deployed by the pipeline. For example, unit tests are used for application code and linters are used for infrastructure as code templates. All builds run without network access to isolate the builds and encourage build reproducibility. Typically, unit tests mock (simulate) all their API calls to dependencies, such as other AWS services. Interactions with “live” non-mocked dependencies are tested later in the pipeline in integration tests. Compared to integration tests, unit tests with mocked dependencies are able to exercise edge cases like unexpected errors returned from API calls and ensure graceful error handling in the code. When the build is complete, the compiled code is packaged and signed.

Test deployments in pre-production environments

Before deploying to production, the pipeline deploys and validates changes in multiple pre-production environments, for example, alpha, beta, and gamma. Alpha and beta validate that the latest code functions as expected by running functional API tests and end-to-end integration tests. Gamma validates that the code is both functional and that it can be safely deployed to production. Gamma is as production-like as possible, including the same deployment configuration, the same monitoring and alarms, and the same continuous canary testing as production. Gamma is also deployed in multiple AWS Regions to catch any potential impact from regional differences.



Integration tests

Integration tests help us to automatically use a service just like customers do as part of the pipeline. These tests exercise the full stack end-to-end by calling real APIs running on real infrastructure in each pre-production stage for all meaningful customer scenarios. The aim of integration testing is to catch any unexpected or incorrect behavior of the service before deploying to production.

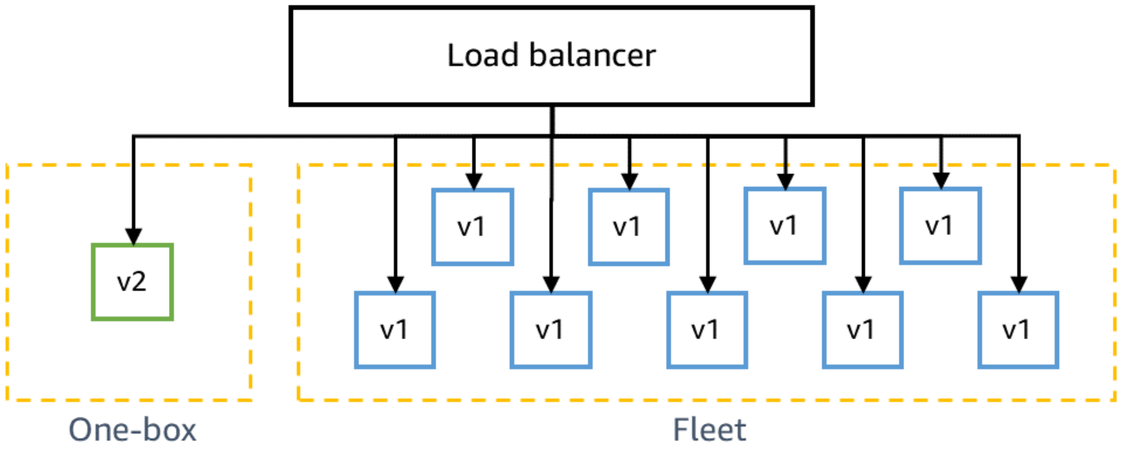
While unit tests run against mocked dependencies, integration tests run against a pre-production system that calls real dependencies, validating the assumptions of the mocks about how those dependencies behave. Integration tests validate the behavior of individual APIs across different inputs. In addition, they validate full workflows that join multiple APIs like creating a new resource, describing the new resource until it is ready, and then using the resource.

Integration tests run both positive and negative test cases, such as providing invalid input to an API and checking that an “invalid input” error is returned as expected. Some pipelines run a fuzz test to generate many possible API inputs and validate that they don’t cause any internal failures in the service. Some pipelines also run a short load test in a pre-production stage to ensure that the latest changes don’t cause any latency or throughput regressions at real load levels.

Backward compatibility and one-box testing

Before deploying to production, we need to ensure that the latest code is backward-compatible and can be safely deployed alongside the current code. For example, we need to detect whether the latest code writes data in a format that the current code can’t parse. The *one-box* stage in gamma deploys the latest code to the smallest unit of deployment, such as to a single virtual machine or single container, or to a small percentage of AWS Lambda function invocations. This one-box deployment leaves the rest of the gamma environment deployed with the current code for some period of time, such as 30 minutes or one hour. Traffic doesn't have to be specially driven to the one box. It can be added to the same load balancer or poll the same queue as the rest of the gamma environment. For example, in a gamma environment of ten containers behind a load balancer, the one box receives ten percent of the gamma traffic generated by continuous canary tests. The one-box deployment monitors canary test success rates and service metrics to detect any impact from the deployment or from having a "mixed" fleet deployed side by side.

The following diagram shows the state of a gamma environment after new code has been deployed to the one-box stage but has not yet been deployed to the rest of the gamma fleet:



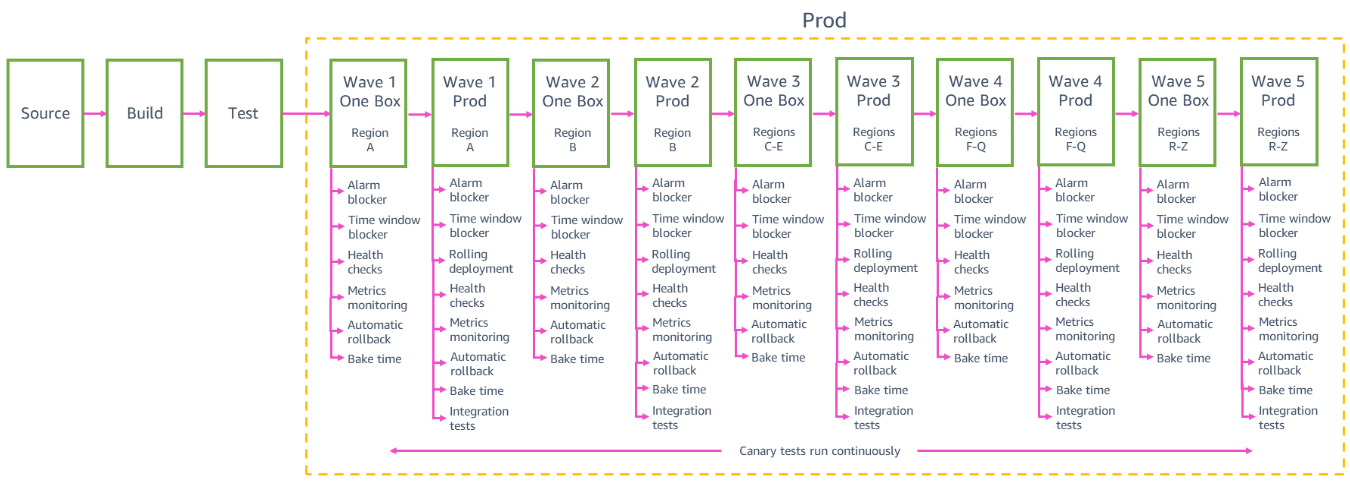
We also need to ensure that the latest code is backward-compatible with our dependencies, for example if a change needs to be made across microservices in a specific order. Microservices in pre-production environments typically call the production endpoint of any services owned by another team, like Amazon Simple Storage Service (S3) or Amazon DynamoDB, but they call the pre-production endpoint of the service team's other microservices in the same stage. For example, a team's microservice A in gamma calls the same team's microservice B in gamma, but it calls the production endpoint for Amazon S3.

Some pipelines also run integration tests again in a separate backward-compatibility stage we call *zeta*, which is a separate environment where each microservice calls only production endpoints, testing that changes going to production are compatible with the code currently deployed in production across multiple microservices. For example, microservice A in zeta calls microservice B's prod endpoint and the production endpoint for Amazon S3.

For a description of strategies for writing and deploying backward-compatible changes, see the Builders' Library article [Ensuring rollback safety during deployments](#).

Production deployments

Our #1 objective for production deployments at AWS is to prevent negative impact to multiple Regions at the same time and to multiple Availability Zones in the same Region. Limiting the scope of each individual deployment limits the potential impact on customers from failed production deployments and prevents a multi-Availability-Zone or multi-Region impact. To limit the scope of automatic deployments, we split the production phase of the pipeline into many stages and many deployments to individual Regions. Teams split regional deployments into even smaller-scoped deployments by deploying to individual Availability Zones or to their service's individual internal shards (called *cells*) in their pipeline, to further limit the scope of potential impact from a failed production deployment.



Staggered deployments

Each team needs to balance the safety of small-scoped deployments with the speed at which we can deliver changes to customers in all Regions. Deploying changes to 24 Regions or 76 Availability Zones through the pipeline one at a time has the lowest risk of causing broad impact, but it could take weeks for the pipeline to deliver a change to customers globally. We have found that grouping deployments into "waves" of increasing size, as seen in the previous sample prod pipeline, helps us achieve a good balance between deployment risk and speed. Each wave's stage in the pipeline orchestrates deployments to a group of Regions, with changes being promoted from wave to wave. New changes can enter the production phase of the pipeline at any time. After a set of changes is promoted from the first step to the second step in wave 1, the next set of changes from gamma is promoted into the first step of wave 1, so we don't end up with large bundles of changes waiting to be deployed to production.

The first two waves in the pipeline build the most confidence in the change: The first wave deploys to a Region with a low number of requests to limit the possible impact of the first production deployment of the new change. The wave deploys to only one Availability Zone (or cell) at a time within that Region to cautiously deploy the change across the Region. The second wave then deploys to one Availability Zone (or cell) at a time in a Region with a high number of requests where it is highly likely that customers will exercise all the new code paths and where we get good validation of the changes.

After we have higher confidence in the safety of the change from the initial pipeline waves' deployments, we can deploy to more and more Regions in parallel in the same wave. For example, the previous sample prod pipeline deploys to three Regions in wave 3, then to up to 12 Regions in wave 4, then to the remaining Regions in wave 5. The exact number and choice of Regions in each of these waves and the number of waves in a service team's pipeline depend on the individual service's usage patterns and scale. The later waves in the pipeline still help us achieve our objective to prevent negative impact to multiple Availability Zones in the same Region. When a wave deploys to multiple Regions in parallel, it follows the same cautious rollout behavior for each Region that was used in the initial waves. Each step in the wave only deploys to a single Availability Zone or cell from each Region in the wave.

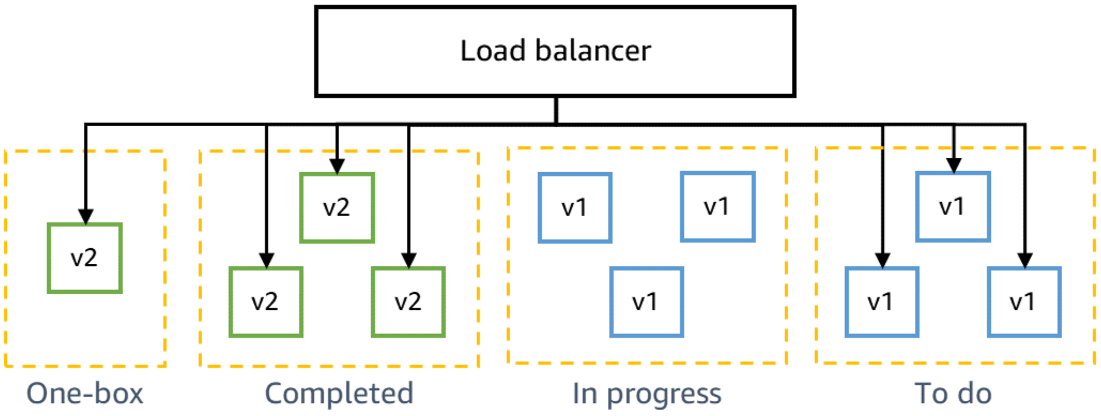
One-box and rolling deployments

Deployments to each production wave start with a one-box stage. As in the gamma one-box stage, each prod one-box stage deploys the latest code to one *box* (a single virtual machine, single container, or a small percentage of Lambda function invocations) in each of the wave's Regions or Availability Zones. The prod one-box deployment minimizes the potential impact of changes on the wave by initially limiting the requests that are served by the new code in that wave. Typically, the one box serves at most ten percent of overall requests for the Region or Availability Zone. If the change causes a negative impact in the one box, the pipeline automatically rolls back the change and doesn't promote it to the rest of the prod stages.

After the one-box stage, most teams use rolling deployments to deploy to the wave's main production fleet. A rolling deployment ensures that the service has enough capacity to serve the production load throughout the deployment. It controls the rate at which the new code is *put into service* (that is, when it starts serving production traffic) to limit the impact of changes. In a typical rolling deployment to a Region, at most 33 percent of the service's boxes in that Region (containers, Lambda invocations, or software running on virtual machines) are replaced with the new code.

During a deployment, the deployment system first chooses an initial batch of up to 33 percent of boxes to replace with the new code. During the replacement, at least 66 percent of the overall capacity is healthy and serving requests. All services are scaled to withstand losing an Availability Zone in the Region, so we know that the service can still serve production load at this capacity. After the deployment system determines that a box in the initial batch of boxes is passing health checks, a box from the remaining fleet can be replaced with the new code, and so on. Meanwhile, we still maintain a minimum of 66 percent of capacity to serve requests at all times. To further limit the impact of changes, some teams' pipelines deploy as little as five percent of their boxes at a time. However, then they do *fast rollbacks*, where the system replaces 33 percent of the boxes at a time with the previous code to speed up rollback.

The following diagram shows the state of a production environment in the middle of a rolling deployment. The new code has been deployed to the one-box stage and to the first batch of the main prod fleet. Another batch has been removed from the load balancer and is being shut down for replacement.



Metrics monitoring and auto-rollback

Automated deployments in the pipeline typically don't have a developer who actively watches each deployment to prod, checks the metrics, and manually rolls back if they see issues. These deployments are completely hands-off. The deployment system actively monitors an alarm to determine if it needs to automatically roll back a deployment. A rollback will switch the environment back to the container image, AWS Lambda function deployment package, or internal deployment package that was previously deployed. Our internal deployment packages are similar to container images, because the packages are immutable and use a checksum to verify their integrity.

Each microservice in each Region typically has a high-severity alarm that triggers on thresholds for the metrics that impact the service's customers (like fault rates and high latency) and on system health metrics (like CPU utilization), as illustrated in the following example. This high-severity alarm is used to page the oncall engineer and to automatically roll back the service if a deployment is in progress. Often, the rollback is already in progress by the time the oncall engineer has been paged and starts engaging.

Example high-severity microservice alarm

```
ALARM("FrontEndApiService_High_Fault_Rate") OR
ALARM("FrontEndApiService_High_P50_Latency") OR
ALARM("FrontEndApiService_High_P90_Latency") OR
ALARM("FrontEndApiService_High_P99_Latency") OR
ALARM("FrontEndApiService_High_Cpu_Usage") OR
ALARM("FrontEndApiService_High_Memory_Usage") OR
ALARM("FrontEndApiService_High_Disk_Usage") OR
ALARM("FrontEndApiService_High_Errors_In_Logs") OR
ALARM("FrontEndApiService_High_Failing_Health_Checks")
```

Changes introduced by a deployment can have an impact on upstream and downstream microservices, so the deployment system needs to monitor the high-severity alarm for the microservice under deployment *and* monitor the high-severity alarms for the team's other microservices to determine when to roll back. Deployed changes can also affect the metrics of continuous canary testing, so the deployment system additionally needs to monitor for failing canary tests. To automatically roll back on all of these possible areas of impact, teams create high-severity aggregate alarms for the deployment system to monitor. High-severity aggregate alarms roll up the state of all of the team's individual microservice high-severity alarms and the state of the canary alarms into a single aggregate state, as in the following sample. If any of the high-severity alarms for the team's microservices go into the alarm state, all of the team's ongoing deployments across all of their microservices in that Region automatically roll back.

Example high-severity aggregate rollback alarm

```
ALARM("FrontEndApiService_High_Severity") OR
ALARM("BackendApiService_High_Severity") OR
ALARM("BackendWorkflows_High_Severity") OR
ALARM("Canaries_High_Severity")
```

A one-box stage serves a small percentage of overall traffic, so issues introduced by a one-box deployment might not trigger the service's high-severity rollback alarm. To catch and roll back changes that cause issues in the one-box stage before they reach the rest of the prod stages, one-box stages additionally roll back on metrics that are scoped to only the one box. For example, they roll back on the fault rate of the requests that were served specifically by the one box, which makes up a small percentage of overall number of requests.

Example one-box rollback alarm

```
ALARM("High_Severity_Aggregate_Rollback_Alarm") OR
ALARM("FrontEndApiService_OneBox_High_Fault_Rate") OR
ALARM("FrontEndApiService_OneBox_High_P50_Latency") OR
ALARM("FrontEndApiService_OneBox_High_P90_Latency") OR
ALARM("FrontEndApiService_OneBox_High_P99_Latency") OR
ALARM("FrontEndApiService_OneBox_High_Cpu_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Memory_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Disk_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Errors_In_Logs") OR
ALARM("FrontEndApiService_OneBox_Failing_Health_Checks")
```

In addition to rolling back on alarms defined by the service team, our deployment system can also detect and automatically roll back on anomalies in common metrics emitted by our internal web service framework. Most of our microservices emit metrics such as request count, request latency, and fault count in a standard format. Using these standard metrics, the deployment system can roll back automatically if there are anomalies in the metrics during a deployment. Examples of this are if the request count suddenly drops to zero, or if the latency or number of faults becomes much higher than normal.

Bake time

Sometimes a negative impact caused by a deployment is not readily apparent. It's *slow burning*. That is, it doesn't show up immediately during the deployment, especially if the service is under low load at the time. Promoting the change to the next pipeline stage immediately after the deployment is complete can end up having an impact in multiple Regions by the time the impact surfaces in the first Region. Before promoting a change to the next production stage, each prod stage in the pipeline has *bake time*, which is when the pipeline continues to monitor the team's high-severity aggregate alarm for any slow burning impact after a deployment is completed and before moving on to the next stage.

To calculate the amount of time we spend baking a deployment, we need to balance the risk of causing a broader impact if we promote changes to multiple Regions too quickly versus the speed at which we can deliver changes to customers globally. We have found that a good way to balance these risks is for earlier waves in the pipeline to have a longer bake time while we build confidence in the safety of the change, and then for later waves to have a shorter bake time. Our goal is to minimize the risk of an impact that affects multiple Regions. Because most deployments are not actively watched by a team member, the typical pipeline's default bake times are conservative and will deploy a change to all Regions in about four or five business days. Services that are larger or highly critical have even more conservative bake times and times for their pipelines to deploy a change globally.

A typical pipeline waits at least one hour after each one-box stage, at least 12 hours after the first regional wave, and at least two to four hours after each of the rest of the regional waves, with additional bake time for individual Regions, Availability Zones, and cells within each wave. The bake time includes requirements to wait for a specific number of data points in the team's metrics (for example, "wait for at least 100 requests to the Create API") to ensure that enough requests have occurred to make it likely that the new code has been fully exercised. During the entire bake time, the deployment is automatically rolled back if the team's high-severity aggregate alarm goes into the alarm state.

Though it's extremely rare, in some cases an urgent change (like a security fix or a mitigation for a large-scale event affecting service availability) might need to be delivered to customers more quickly than the time the pipeline usually takes to bake changes and deploy. In these cases, we can dial down the pipeline's bake time to accelerate the deployment, but we require a high level of scrutiny on the change to do this. For these cases we require the scrutiny of the organization's Principal Engineers. The team must review the code change, as well as its urgency and risk of impact, with very experienced developers who are experts at operational safety. The change still goes through the same steps in the pipeline as usual, but gets promoted to the next stage more quickly. We manage the risk of a faster deployment by limiting the changes in flight in the pipeline during this time to allow only the most minimal code changes needed to address the current issue and by actively watching the deployments.

Alarm and time window blockers

The pipeline prevents automatic deployments to production when there is a higher risk of causing a negative impact. The pipeline uses a set of "blockers" that evaluate deployment risk. For example, automatically deploying a new change to prod when an issue is currently ongoing in the environment could make the impact worse or more prolonged. Before starting a new deployment to any prod stage, the pipeline checks the team's high-severity aggregate alarm to determine whether there are any active issues. If the alarm is currently in the alarm state, the pipeline prevents the change from moving forward. Pipelines can also check organization-wide alarms, like a large-scale event alarm that indicates whether there is a broad impact in another team's systems, and prevents starting a new deployment that could add to the overall impact. These deployment blockers can be overridden by developers when a change needs to be deployed to prod to recover from a high-severity issue.

The pipeline is also configured with a set of time windows that define when a deployment is allowed to start. When we configure time windows, we need to balance two causes of deployment risk. On the one hand, very small time windows can cause changes to pile up in the pipeline while the time window is closed, increasing the likelihood that any one of those changes in the next deployment will have an impact when the time window opens. On the other hand, very large time windows that go beyond regular business hours increase the risk of prolonging the impact from a failed deployment. During off-hours, it takes longer to engage the oncall engineer than during the day, when the oncall engineer and other team members are working. During regular business hours the team can be more quickly engaged after a failed deployment in case any manual recovery steps are needed.

Most deployments aren't actively watched by a team member, so we optimize the timing of deployments to minimize the time it takes to engage an oncall engineer, in case there is manual action required for recovery after an automatic rollback. Oncall engineers typically take longer to engage at night, on office holidays, and on weekends, so these times are excluded from the time windows. Depending on the usage patterns of the service, some issues might not surface for hours after the deployment, so many teams also exclude Fridays and late-afternoon deployments from their time windows to reduce the risk of needing to engage the oncall engineer at night or during the weekend after a deployment. We have found that this set of time windows enables a fast recovery even when manual action is needed, ensures less engagement with oncall engineers outside of regular working hours, and makes sure that a small number of changes is bundled together while the time windows are closed.

Pipelines as code

The typical AWS service team owns many pipelines to deploy the team's multiple microservices and source types (application code, infrastructure code, OS patches, etc.). Each pipeline has many deployment stages for an ever-increasing number of Regions and Availability Zones. This translates into a lot of configuration for the team to manage in the pipeline system, in the deployment system, and in the alarm system, and a lot of effort to keep up to date with the latest best practices and with new Regions and Availability Zones. In the last few years, we have embraced practicing "pipelines as code" as a way to more easily and consistently configure safe, up-to-date pipelines by modeling this configuration in code. Our internal pipelines as code tool pulls from a centralized list of Regions and Availability Zones to easily add new Regions and Availability Zones to pipelines across AWS. The tool also allows teams to model pipelines using inheritance, defining configuration that is common across a team's pipelines in a parent class (like which Regions go in each wave and how long bake time should be for each wave) and defining all microservice pipeline configuration as a subclass that inherits all the common configuration.

Conclusion

At Amazon, we have built our automated deployment practices over time based on what helps us balance deployment safety against deployment speed. At the same time, we want to minimize the amount of time developers need to spend worrying about deployments. Building automated deployment safety into the release process by using extensive pre-production testing, automatic rollbacks, and staggered production deployments lets us minimize the potential impact on production caused by deployments. This means that developers don’t need to actively watch deployments to production.

With fully automated pipelines, developers use code reviews to check their code and also to approve that the change is ready to go to production. After the change is merged into the source code repository, the developer can move on to the next task and forget about the deployment, trusting the pipeline to get their change to production safely and cautiously. The automated pipeline takes care of deploying continuously to production multiple times a day, while balancing safety and speed. By modeling our continuous delivery practice in code, it’s easier than ever for AWS service teams to set up their pipelines to deploy their code changes automatically and safely.