Going faster with continuous delivery

Mark Mansour

Going faster with continuous delivery

Continuous improvement and software automation

Over 10 years ago, we undertook a project at Amazon to understand how quickly our teams were turning ideas into high-quality production systems. This led us to measure our software throughput so that we could improve the speed of our execution. We discovered that it was taking, on average, 16 days from code check-in to production. At Amazon, teams started with an idea, and then typically took a day and a half to write code to bring the idea to life. We spent less than an hour to build and deploy the new code. The remainder of the time, almost 14 days, we spent waiting for team members to start a build, to perform deployments, and run tests. At the end of the project, we recommended automating our post check-in processes to improve our speed of execution. The goal was to eliminate delays while maintaining or even improving quality.

At the heart of this recommendation was a continuous improvement program to increase the speed of our execution. We based our decision to improve our execution speed on to the leadership principle of Insistence on the Highest Standards. This principle is about having relentlessly high standards, continually raising the bar, and delivering high quality products, services, and processes. Our Leadership Principles describe how Amazon does business, how leaders lead, and how we keep the customer at the center of our decisions.

Amazon had already built software development tools to make our software engineers more productive. We created our own centralized and hosted build system, Brazil, that executes a series of commands on a server with the goal of generating an artifact that could be deployed. At this point Brazil didn't listen to source code changes. A person had to initiate a build. We also had our own deployment system, Apollo, which required a build artifact to be uploaded to it as part of initiating a deployment. Inspired by the interest in continuous delivery occurring in the industry, we built our own system, Pipelines, to automate the software delivery process between Brazil and Apollo.

Pipelines: Our continuous deployment tool

We started a pilot program to automate the software delivery process for a small number of teams. By the time we were done, the headline pilot team had achieved a 90 percent reduction in the overall time it took to go from check-in to production.

The project validated the concept of a pipeline as a way for our teams to define all the steps needed to release software to customers. The first step in a pipeline is to build an artifact. The pipeline then runs that build artifact through a series of steps until the artifact is released to all customers. We use pipelines to reduce the risk that a new code change could have a negative impact on our customers. Each step in the pipeline should increase our confidence that the build artifact doesn't contain defects. If a defect does reach production, we want to get production back to a healthy state as quickly as possible.

When we launched Pipelines, it could only model a single release process per application. This limitation drove consistency, standardization, and simplification of a team's release processes. This resulted in fewer defects. Before we moved to using pipelines, it was common for teams to have different release processes for bug fixes and major feature releases. As other teams saw the success of the teams that had piloted automated delivery, they started to migrate their hand-managed release processes into pipelines so that they could also improve consistency. Teams that used to have a variety of release processes now had one standardized process that everyone used. In addition, when they moved their release processes into a tool, team members often revisited their approach and found ways to simplify their process.

The Pipelines team had yearly goals to increase usage by using “seductive adoption.” In other words, they needed to make the product so good that people will demand to use it. We measured the number of teams using a pipeline to deploy their software to production, and we classified pipelines by their level of automation. We saw teams taking goals to use a pipeline to release software and to move toward fully automated releases. However, we noticed that in some organizations, the way we were measuring quality could lead teams to automate their release process without providing for any testing.

The answer to the question, “how much testing is enough?” is a judgement call. It requires a team to understand the context in which they operate. To deal with this situation we used another leadership principle, Ownership. This principle is about thinking long term and not sacrificing long-term value for short-term results. Software teams at Amazon have a high bar for testing, and spend a lot of effort on it, because owning a product means also owning the consequences of any defects in that product. If a problem were to have an impact on customers, it is members of the small, single-threaded software team who handle that issue and fix it in real time. The tension between increasing execution speed and responding to issues in production means that teams are motivated to test adequately. However, if we over invest in testing, then we might not succeed because others have moved faster than us. We’re always looking to improve our software release processes without becoming a blocker to the business.

Another problem we faced is that teams were not learning software release best practices from each other. Single-threaded teams are encouraged to work autonomously, and that meant that engineers were solving their deployment problems independently. When they found a solution that addressed their software release needs they’d promote the technique to other engineers through mailing lists, operations meetings, and other communication channels. There were two problems with this style of communication. First, these channels are a best-effort communication channel, meaning that not everyone learned about the new techniques. Second, leaders encouraging their teams to adopt the new best practice had no way to understand if their teams had done the work necessary to actually adopt the best practice. We realized that we needed to help all engineers gain access to the best practices we’d learned, and give leaders the ability to identify pipelines that needed attention.

Our solution was to mechanize our learning by adding checks for best practices into the tools we use to build and release software. We were sensitive to the fact that a best practice for one organization might not be a good practice for another, so we allowed these checks to be configured on a per-organization basis. Organizational-level best practice checks gave leaders the ability to tailor their release processes to meet the needs of their business. Leaders wanting to encourage or enforce a new best practice were able to start by provide a warning from within the tools that engineers used on a daily basis. By placing messages in the tools, it almost guaranteed that team members would learn about the best practice and when that best practice would take effect. We found that by giving teams time to learn about and debate a new best practice an organization had a chance to iterate and improve their best practice checks. This ultimately led to improving the quality of the best practice and better buy-in from the engineering community.

We systematically identified the best practices to apply. A group of our most senior engineers catalogued common reasons for a release not working. They identified the steps that would have made the release work. Then we used that list to build our set of best practice checks. Through this process, we came to realize that although we wanted new software revisions to reach customers instantly, without effort, and without degrading availability, we prioritize availability first, then speed, and then making it easier for our engineers.

Reducing the risk that a defect will reach customers

Our release processes, including our pipelines and deployment systems, must be designed to identify those defects as quickly as possible and prevent them from having an impact on our customers. We need to make sure that our release processes are configured correctly and make sure that our build artifact works as intended.

Deployment hygiene: The most basic form of deployment testing ensures that the newly deployed artifact can be started and can respond to work. As part of the post-deployment workflow, we run quick checks that ensure the newly deployed artifact has started and is serving traffic. For example, we use lifecycle event hooks in the AWS CodeDeploy AppSpec file to trigger simple scripts to stop, start, and validate the deployment. We also check that we have enough capacity to serve customer traffic. We've built techniques like minimum healthy hosts in CodeDeploy to validate that we always have enough capacity to serve our customers. Finally, if the deployment engine can detect a failure, it should roll back the change to minimize the time that customers see a defect.

Testing prior to production: One of Amazon's best practices is to automate unit, integration, and pre-production testing, and add these tests into our pipeline. We insist that we perform load and security testing, with a bias to adding these tests into our pipelines. When we say unit tests, we mean all tests that you might want to perform on your build machine, including style checks, code coverage, code complexity, and more. We think of integration tests as including all off-box testing such as failure injection, automated browser testing, and the like. There are many great articles on unit and integration tests, so I won't go into more details here.

Our unit and integration testing aim to verify that our build artifact's behavior is functionally correct. The more validation we perform, the lower the risk of a defect being exposed to our customers. To reduce the time it takes to get a product into our customer's hands, we try to detect a defect as early in the release process as possible. In general, this means that if your tests are smaller and faster, you'll receive feedback quicker on any problems with your changes.

At Amazon, we also use a technique we call pre-production testing. A pre-production environment is the last place where testing occurs before we deploy our changes into production. A pre-production environment test uses the system's production configuration so that it acts exactly like a production system. This approach has two benefits. First, pre-production environments test the production configuration to make sure that the service can correctly connect to all production resources, including production data stores. Second, it ensures that the system interacts correctly with the APIs of the production services it depends on. Pre-production environments are only used by the team that owns that service, and they are never sent traffic from customers. Running pre-production tests increases our confidence that the same code and configuration will work in production.

Validating in production: When we release code to our customers, we don't do it all at once. The scope of the impact of releasing a defect to all customers at once is too large. Instead, we deploy to cells—a completely independent instance of a service. When we deploy changes to our first set of customers in our first cell, we are extremely cautious. We'll only let a small number of customers see the new change, and we gather feedback on whether the new code is working. We monitor the amount of errors that our services emit after a canary deployment. If the error rate goes up, we will automatically roll the change back. For example, we might wait for 3,000 positive data points, with no negative data points, before continuing a deployment.

A complication can arise if your automated tests miss a use case. We strive to catch all errors with our structured and repeatable tests, whether they are automated or manual. However, even when we try our hardest, a defect can slip through. To test our tests, we leave the new change in production for a fixed period of time to see if a non-team member finds an issue. We have spent a lot of time debating whether we should let changes just sit in production or how long we should wait after a canary deployment before deploying to the rest of the deployment group. Many of our teams have decided to wait for a fixed period of time in addition to gathering positive data points before progressing through our deployment routine. The amount of time a pipeline waits is highly dependent on the team. Some teams wait for hours and others wait for minutes. The higher the impact and the longer the time to remediate the issue, the slower the release process.

After we've gained confidence in the first cell, we'll then expose the new code change to more and more customers until it is completely released. Just as we did with the canary deployment, we wait to gain confidence in the deployment to the first new cell before progressing to the next cell. As we gain more confidence in the build artifact, we reduce the time we spend verifying the code change. This results in a pattern where we are aiming to get from check-in to our first production customer as quickly as possible. However, after we're in production we slowly release the new code to customers, working to gain additional confidence as we incrementally speed up the remainder of our deployments.

To make sure that our production systems are continuing to serve customer requests, we generate synthetic traffic on our systems. We want fast feedback if our service isn't working correctly, so we run our synthetic tests at least every minute. We design synthetic tests to make sure that our running processes are healthy and that all dependencies are tested, which often involves testing all public-facing APIs.

Controlling when software is released: To control the safety of our software releases, we've built mechanisms that allow us to control the speed at which changes move through our pipeline. We use metrics, time windows, and safety checks to control when our software is released.

Pipelines can be configured to prevent a deployment when an alarm is fired based on a change in metrics. We use metrics pervasively, and we have alarms on the health of our systems, the health of our cells, Availability Zones and Regions, and almost anything else you can think of. We configure our pipelines to halt the deployment of code when an important metric has triggered an alarm. However, sometimes a team needs to deploy a fix so that the system's alarm is addressed. For this scenario, we let teams override the alarms preventing changes from moving through a pipeline.

Our pipelines can specify a time window in which changes are allowed to progress through a pipeline. Teams can find their own time windows to restrict when changes reach customers. AWS teams prefer to release software when there are plenty of people who can quickly respond and mitigate an issue caused by a deployment. To make this a reality, teams generally set their time windows so that they deploy during business hours only. Other teams at Amazon prefer to release software when there is low customer traffic. These time windows can be overridden if needed.

We also have the ability to halt a pipeline based on the contents of the build artifact. For instance, we can block a build artifact that contains a known bad package or a specific Git reference. We have used this feature when we've discovered that a change to a package contained a performance regression. If we were to only remove the package from our package repository, then pipelines already containing the defective package would still deploy that bad change to customers.

How we've approached the speed of our execution

We've found that teams are eager to embrace automation. We're all highly motivated to build and release features to customers that improve their lives, and continuous delivery makes that sustainable. We've seen automation give engineers back time by removing frustrating, error prone, and laborious manual work. We've shown that continuous deployment has a positive impact on quality. We've seen that automation allows teams to release frequently, a change at a time, making it easier to identify regressions.

When systems are new, the surface area to test is usually understandable by most team members, which makes some manual testing tractable. However, as systems become more complex, and the team members change, then the value of automation increases. We like automating our systems so we can focus on adding customer value rather than hand managing the process of getting those changes out to customers.

For many years, Amazon has run continuous improvement programs focused on the speed with which we release software to customers and the safety of those releases. We didn't start with all of the risk checks and tests I've written about in this article. Over time, we've invented ways to identify and mitigate risk.

The continuous improvement programs are run by business leaders at different levels of the organization. This allows each business leader to adjust their software release process to match the risks to and impact on their business. Some of our continuous improvement programs are run across large sections of Amazon, and sometimes leaders of smaller organizations run their own programs. We know there are always exceptions to the rule. Our systems have opt-out mechanisms so we don't slow down teams that need a permanent or temporary exemption. Ultimately, our teams own the behavior of their software, and they are responsible for investing appropriately in their software release process.

We started by measuring where our pain was, addressing it, and iterating. To make this work sustainable, we needed to do it incrementally, and celebrate the improvements over time. When Amazon first started using pipelines, many teams were not sure that continuous deployment would work. To get teams started we encouraged teams to encode their current release process, manual steps and all, in a pipeline. For many teams their pipeline acted as a visual interface to their release process without automatically promoting build artifacts through a release process. As their confidence grew, they gradually turned on automation at different stages of their pipeline until they no longer needed to manually trigger any step of their pipeline.

Fast forward to today. Amazon is now at the point where teams aim for full automation as they write new code. For us, automation is the only way we could have continued to grow our business.