One Developer Experience — Build, Deploy, and Experiment

By: Ravi Lachhman . Evangelist at Harness

Table of Contents

Summary

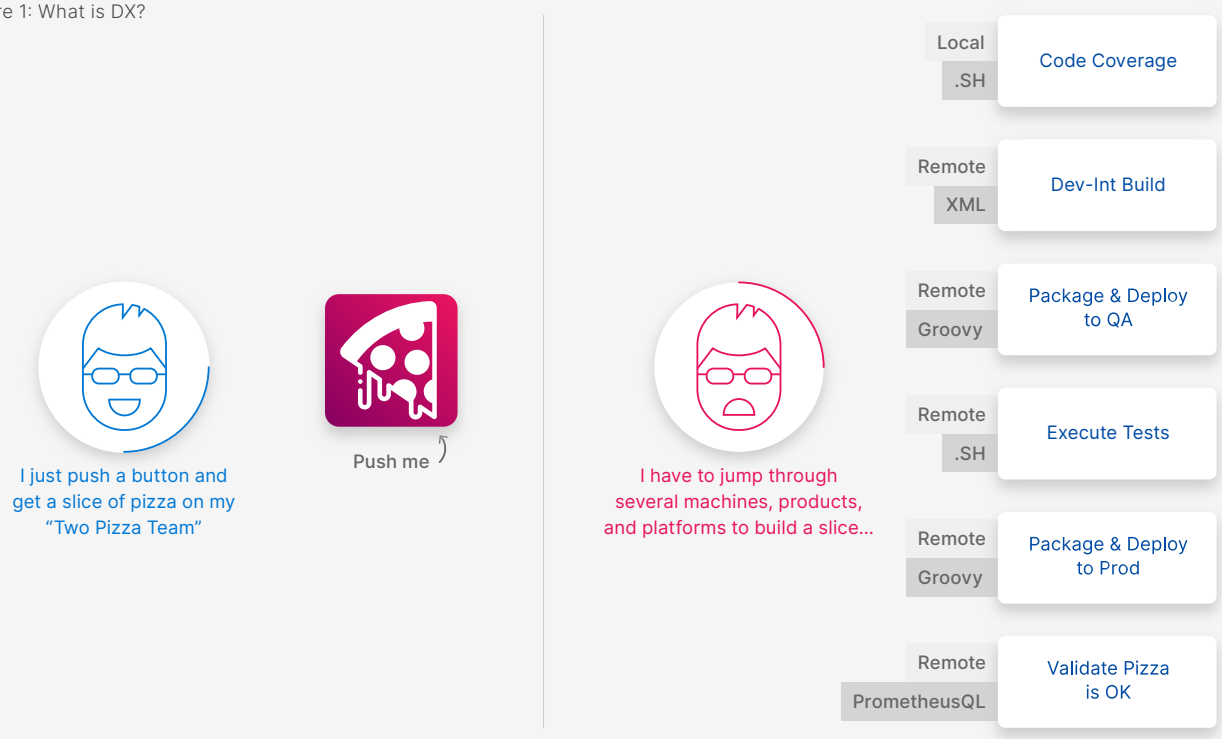
Taking a look at where major DevOps trends are headed, a common theme across many tools and practices is improving the Developer Experience. One paradigm of thinking is that if you improve your internal customer experience, your external customers will benefit. The Developer Experience has been quite siloed/segregated for a multitude of reasons, such as scaling or having best-of-breed technologies to support individual concerns. In this eBook, we will go through how you can improve your developer experience across the entire SDLC.

What Is Developer Experience/DX?

Developer experience is the overall interactions and feeling that the developer feels when working towards a goal. It is similar to the definition of User Experience (UX), but the primary user is a software engineer. The core definition of UX is how a user interacts and experiences a product, system, or service focusing on perceptions of utility, ease of use, and efficiency. User Experience focuses on improving customer satisfaction and loyalty. Developers, however, are seen as internal customers vs. external customers that traditional UX targets.

With the consumerization of enterprise IT, the expectation for users of enterprise systems continues to rise because of their experiences outside of the enterprise. Seen as highly technical when compared to an average business system user, the Developer Experience can sometimes take a back seat because of a stigma that "developers can just figure it out." Having poor DX is a detriment to overall engineering efficiency and the ability to innovate and iterate while not adding to technical debt.

Figure 1: What is DX?



How Do You Measure Developer Experience?

Similar to User Experience, there can be multiple objective and subjective measures of Developer Experience. Focusing on the three pillars of User Experience, usability, findability, and credibility, are excellent markers of measuring Developer Experience.

Usability

Usability is how easy your software or service is to use. It answers the question, "Would a first-time user of your platform be able to achieve their goal?" Also, usability is about removing barriers. For example, unnecessary steps. Measuring usability for internal customers might be a little more challenging if the frameworks that are in place for UX for external customers are not there. Surveys are a good way of gathering feedback and also, if possible, using remote usability tests such as heat-mapping/tracking tools to measure interaction and drop-off.

Findability

Findability is how easy or quickly a user of your software or service can find the functionality they are looking for. Having logical, contextual, and concise search/navigation features are important. Supporting the user journey across their changing tasks alludes to good findability. Similarly to measuring usability, asking users what they think or having the ability to record times/heatmaps of what is being clicked on or where frustration starts are all good steps.

Credibility

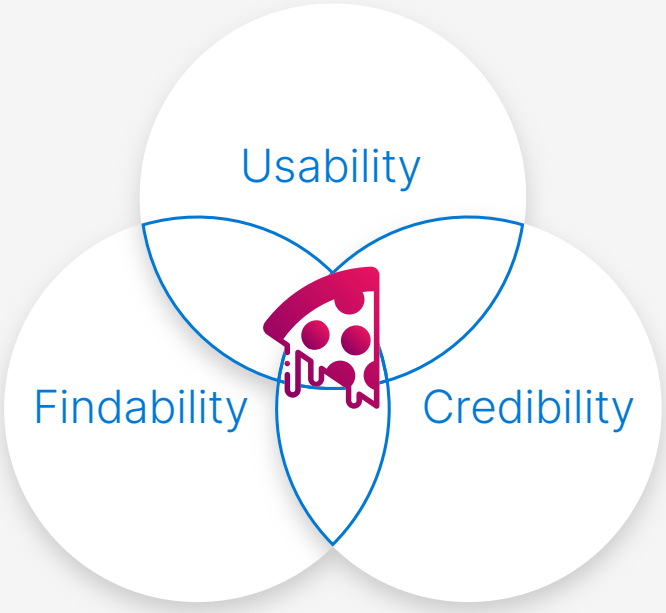
Credibility is that your users trust your software or service to solve their problems. Credibility is a long-term pillar, not only for today's needs but for tomorrow's needs also. The changing landscape of developer technology does make this hard to keep up. For Developer Experience, having a service that is reliable and performant is key to credibility. Meeting the scaling needs of the wide swath of technologies that internal customers can be challenging. Continued adoption of your software or service is a key measure of credibility.

How Do You Measure Developer Experience? (Cont.)

One pitfall to avoid is just focusing on the output first; for example, observability in the manufacturing world infers that if the output is good, the internal states are good. If your team or organization is producing a lot, then the DX must be good. The truth is, that's not necessarily the case: high production might be one indicator but there might also be high toil or high burn. The expertise, staff, and overhead costs are high to produce large amounts of artifacts.

Good Developer Experience will lead to more efficient, and thus more highly productive teams. Looking at the four key metrics from Accelerate (deployment frequency, lead time for changes, MTTR, and change failure rate), they can all improve.

Figure 2: UX/DX Pillars

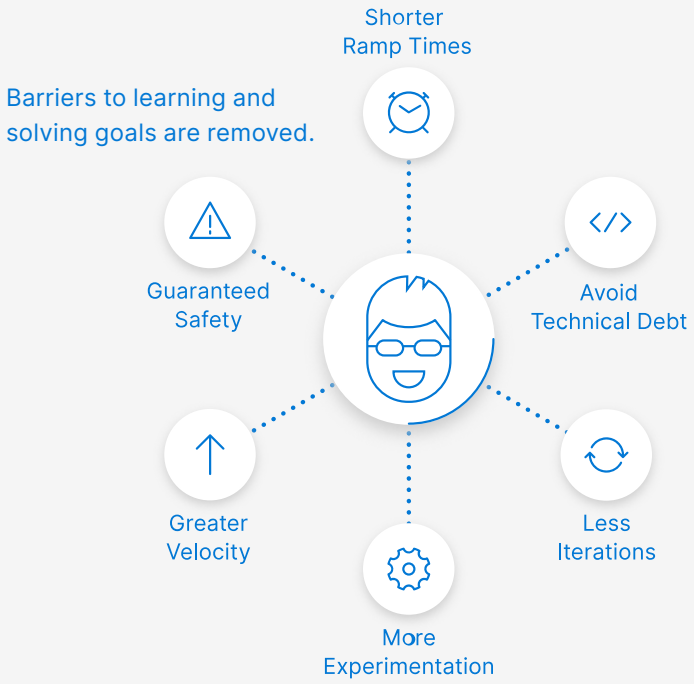


What Does Good Developer Experience Look Like?

Good Developer Experience supports innovation, iteration, safety, and velocity. Great Developer Experience takes those pillars by allowing developers to focus on and experiment with what is important and not pick up technical debt with non-functional or operational concerns along the journey. Software is a team sport. Allowing for quick integration and deployment of changes to get into the hands of the end users is the goal of software engineering.

Two of the biggest detriments of Developer Experience are context switching and decision fatigue. Imagine a customer service representative at an airline having a high volume of complex issues, switching between multiple systems and reaching out to different departments for answers/expertise - that can drive up resolution time (e.g. context switching). The need to constantly make decisions and negotiate can also lead to decision fatigue. Ironically, software engineers face similar dilemmas navigating the end of their workstreams, e.g. heading to production and supporting production.

Figure 3: Good Developer Experience



What Does Good Developer Experience Look Like? (Cont.)

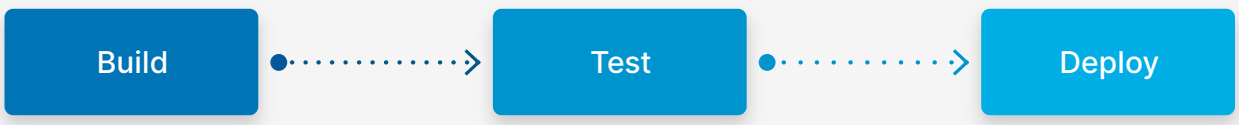
Because of competing priorities and deadlines, software engineers are great at finding the path of least resistance and technical debt can accrue if not careful. Reducing context switching and decision fatigue for developers by having highly usable and credible software and platforms alludes to a great DX.

Comparing StackOverflow Developer Survey results over the last few years, an interesting trend appears. In today's market, it is not normal for someone to work their entire career on one project or at one firm. The amount of time that developers are on projects is decreasing, and the amount of time that it takes developers to be productive members of the team (ramp up time) is increasing. Good DX will help reduce ramp up time and assist in the ultimate goal: getting ideas to production.

Typical Journey to Production

If your software is not available to someone or something else, is your software even a piece of software? As metaphoric as that statement is, as a software engineer, having what you built on your local machine does nobody any good. At its most simplistic form, you need to build, test, and deploy your changes.

Figure 4: Overly Simple



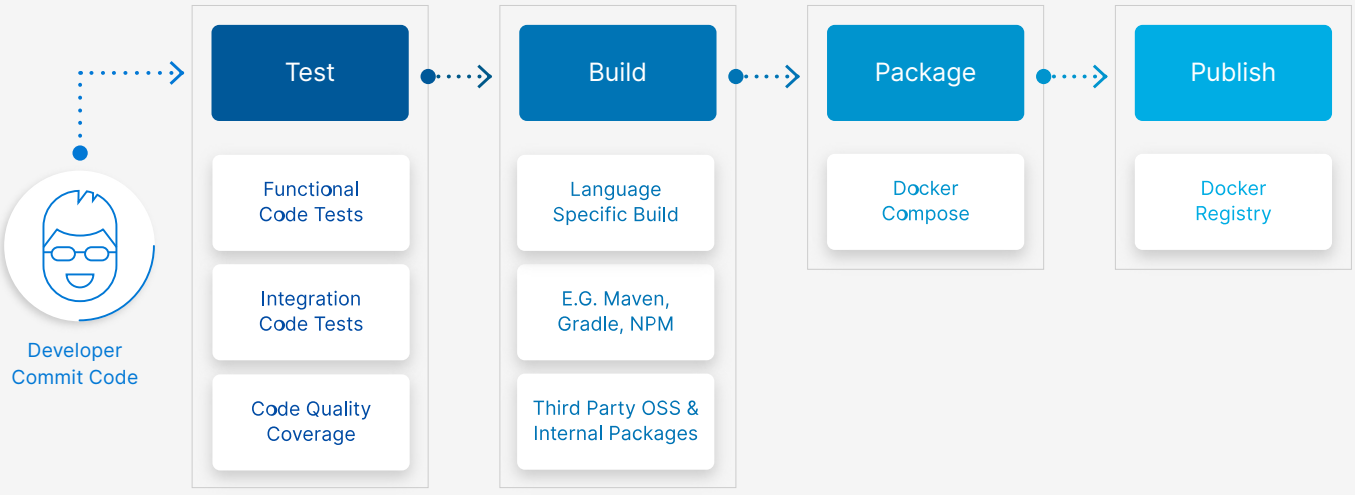
Typical Journey to Production (Cont.)

Your idea can transverse many environments and pieces of infrastructure before being in the hands of the users. Even taking safety into consideration, you might have to navigate release strategies, such as a canary release, to incrementally update running systems. As more items shift left towards the developer, the amount of broadened knowledge, such as infrastructure automation and DevSecOps practices, can be challenging to learn.

Thanks to advancements in Continuous Integration and Continuous Delivery, your changes can be built, packaged, tested, and deployed safely into new or existing infrastructure. Automation and expertise can be placed into CI/CD pipelines and beyond to help further the journey and achieve DX goals. A more complete journey to production can look like the journey below; getting a deployable artifact and deploying that artifact.

Going from code to artifact might look like the below diagram:

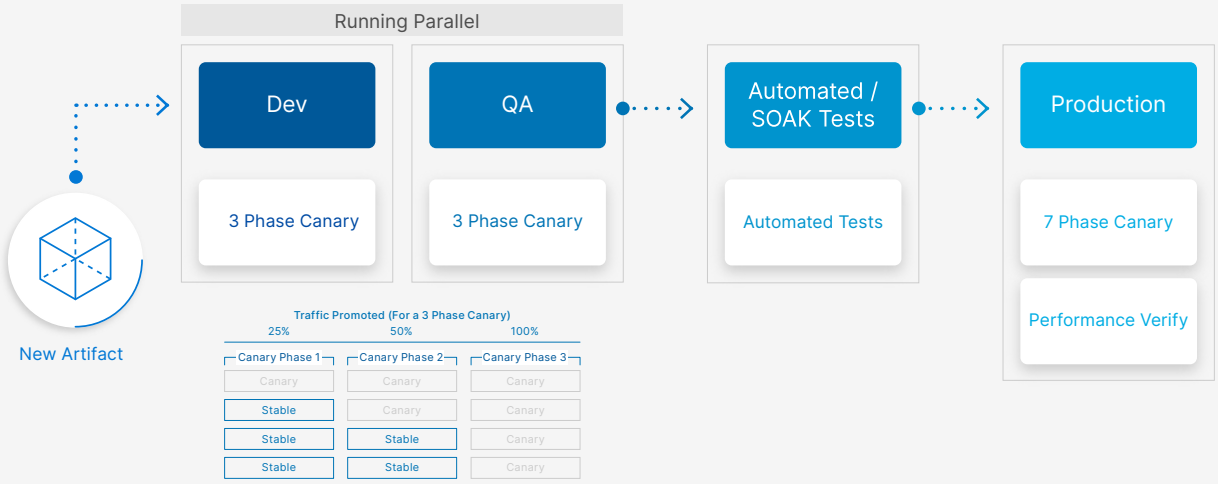
Figure 5: Deployable Artifact



Typical Journey to Production (Cont.)

Though having an artifact is only part of the journey. In modern software delivery there is a need to orchestrate confidence building steps and safety. Automation is key for consistency and a good developer experience. A highly-automated deployment to production might look like the below (leveraging a canary release):

Figure 6: Automated Deployment



Expectation On and of Modern Software Engineers

Engineers, in general, are natural optimizers. The ability to work more efficiently is very important. Because of the rise in incremental development practices such as Agile, the velocity of work and demands for features can be infinite. The only limitation to software is really time and resources.

Expectation On and of Modern Software Engineers (Cont.)

Software engineers are also naturally inclined to better the craft and in a field that requires constant learning. Several paradigms continue to shift left towards the developer, as brought up in the previous section, such as security with the DevSecOps movement and application infrastructure automation thanks to Kubernetes. The engineering burden continues to increase.

Modern software engineers expect a good DX. Needing to buck the trend of the ever-increasing ramp up times to be productive, software engineers can feel fulfillment seeing the fruits of their labor quicker by building and deploying at the pace of innovation.

Best Experience When Building Software/ Continuous Integration

Extending the positive local build experience outside of a developer's development environment does take thought. Continuous Integration is build automation and focuses on externalizing the build. Though, more than just the compiled source code can go into a build; the end product for Continuous Integration is a release candidate.

A core tenet of engineering efficiency is meeting your internal customers where they are. For software engineers, this is being as close to their tools and projects as possible. Like many modern pieces of application infrastructure, shifting left to the developer means being included in the project structure in source code management (SCM).

As the velocity of builds increases to match the mantra that "every commit should trigger a build," development teams could potentially be generating several builds per day per team member, if not more. The firepower required to produce a modern containerized build has increased over the years, versus traditional application packaging.

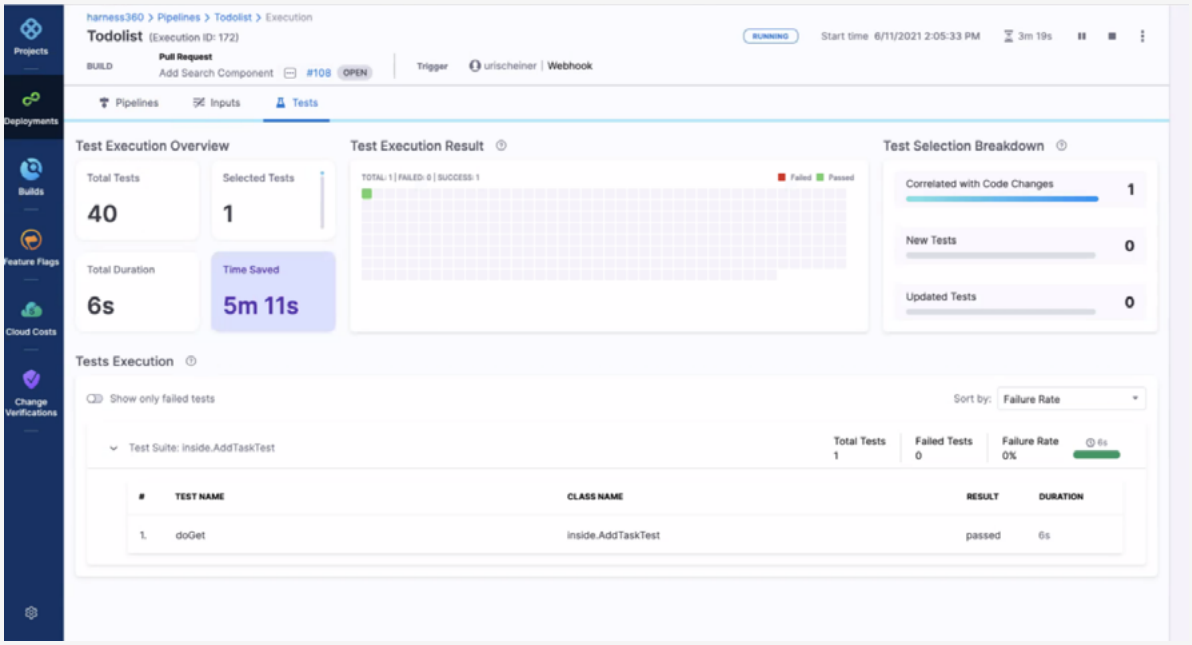
Best Experience When Building Software/ Continuous Integration (Cont.)

Typically, an organization's first forays in running automated tests in a repeatable and consistent fashion end up in their Continuous Integration pipelines. Usually, this is an easy lift; the same code/test coverage that a developer is subject to in their local build makes its way into the build pipeline since those steps should have been executed before the commit.

A common distributed system fallacy is that one person understands the entire end-to-end of the system. This is not true. When adding new features or expanding test coverage, we are prone to a Big Ball of Mud pattern, both in development and test-wise. Execution times and complexity of test suites potentially increase with every new change. By running tests in an intelligent manner and only executing tests that are prudent to the new changes, this significantly combats test coverage complexity.

Showing time saved by executing test coverage in an intelligent manner:

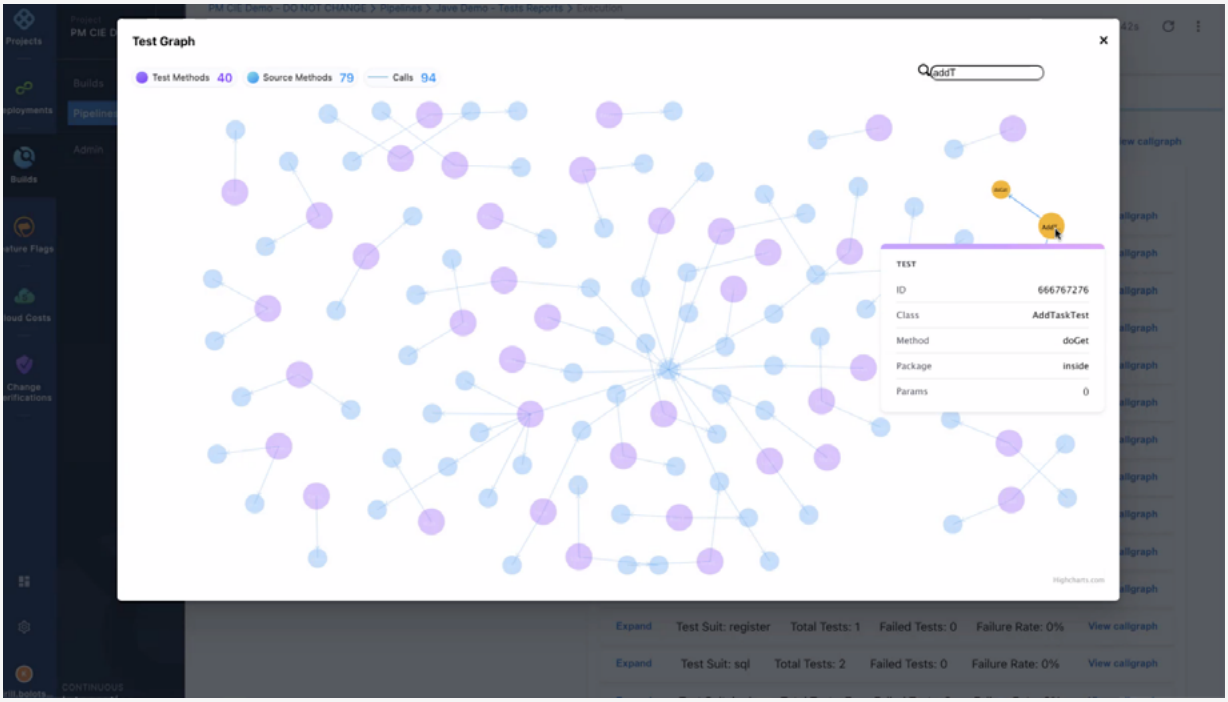
Test Optimization — Time Saved



Best Experience When Building Software/ Continuous Integration (Cont.)

Helping determine appropriate test coverage by modeling coverage and changes:

Automatically Building Test Coverage Graph and Model

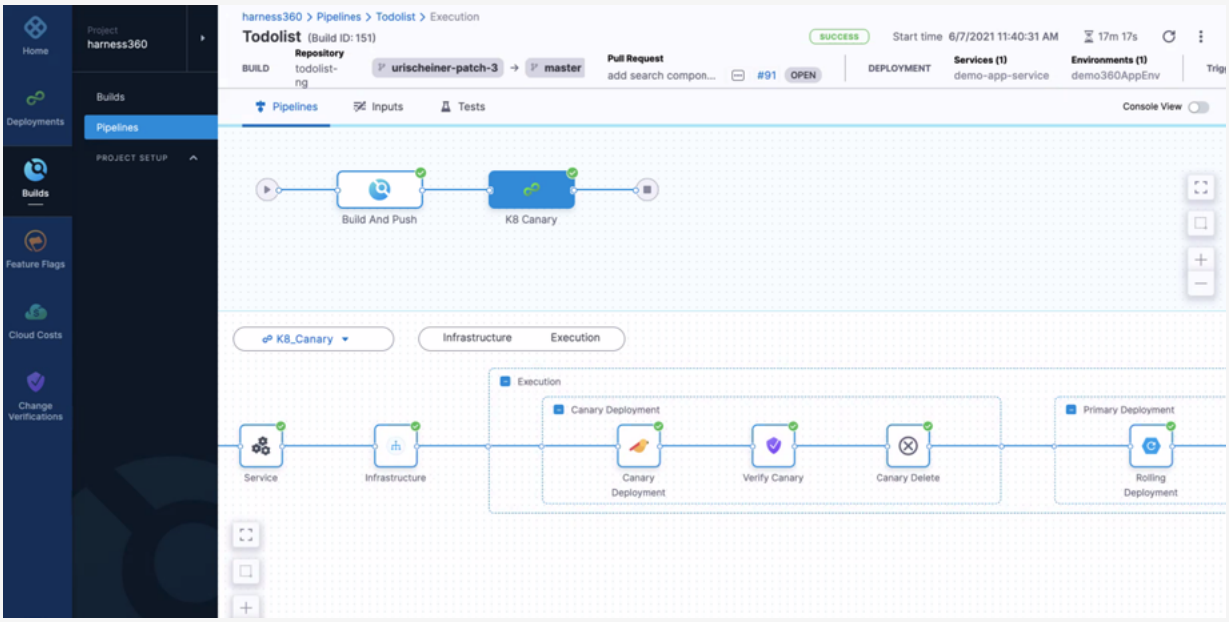


Best Experience When Deploying Software/ Continuous Delivery

Building software is typically done by convention, but deploying software can be very bespoke. Software is the culmination of decisions before, during, and potentially after your time on a project or product team. Navigating all of the application and infrastructure choices, especially on a live system with user traffic, can be complex and unique to each team.

Deploying incremental changes into a development-integration environment usually is geared towards DX as developers have full control over the environment. Though as the march towards production continues, certain organizations might be under business controls that developers are not allowed in production. This is where Continuous Delivery steps in.

Kubernetes Canary Deployment



Best Experience When Deploying Software/ Continuous Delivery (Cont.)

The definition of Continuous Delivery from Jez Humble:

Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes, and experiments—into production, or into the hands of users, safely and quickly in a sustainable way.

Safely means pipelines enforce quality, testing, and a mechanism to deal with failure. Quickly implies pipelines promote code without human intervention, in an automated fashion, and finally, sustainable means pipelines can consistently achieve all this with low effort and resources.

Good delivery is fast and repeatable, great pipelines are fast, safe, and repeatable.The book Accelerate states that elite performers have a lead time of less than 1 hour, and a change failure rate of less than 15% for production deployments. Therefore, a great pipeline will complete in under an hour, and catch 95% of anomalies and regressions, before code reaches an end user.

If your code takes longer than an hour to reach production, or if more than 2 out of 10 deployments fail, you might want to reconsider your pipeline design and strategy. Because experimentation takes iteration, having an all-out deployment might be too lengthy to make targeted changes or experiments if it needs to happen throughout the day. Experimentation is important in the next generation of DX.

Importance of Experimentation for Developer Experience

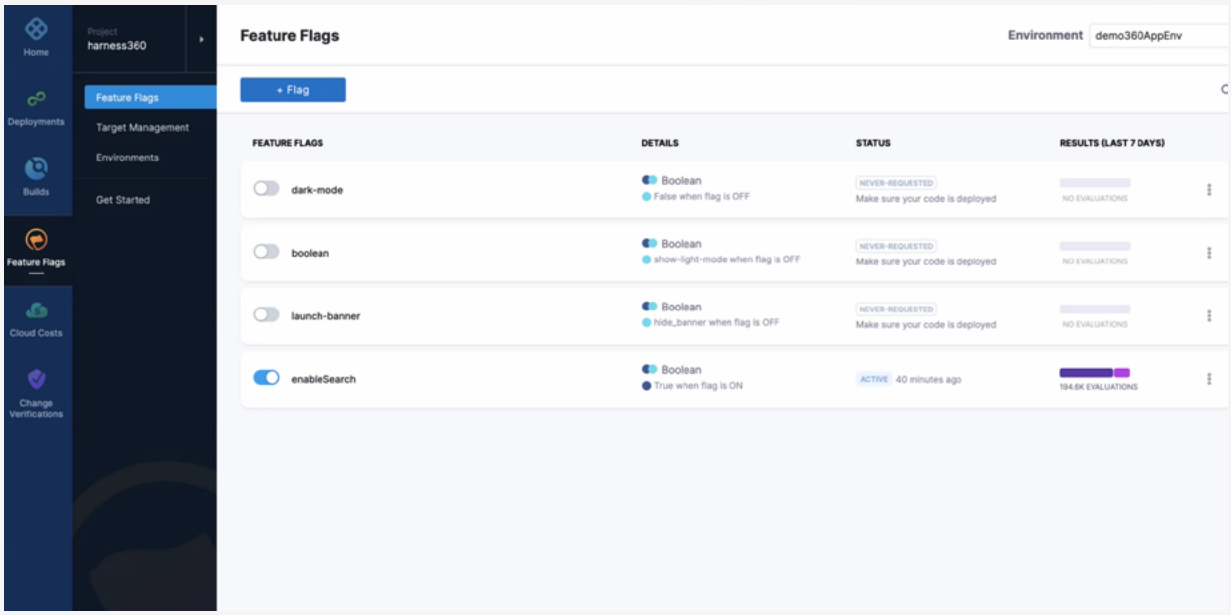
Experimentation has been around in the business world for a long time. From credit card vendors changing credit rules for certain cards to the placement of goods in retail stores, experimentation in business is to be expected. However, because of previous pain or fear of deploying and potentially having an unbound blast radius (e.g. impacting all users), experimentation with production traffic/users might take fairly long cycles in the development world—and sometimes altogether be avoided.

Importance of Experimentation for Developer Experience (Cont.)

The ability to experiment goes hand in hand with the ability to quickly iterate. Core to innovation work is to iterate, adjust, and optimize. Without having data from different renditions or experiments, improvements can turn into guesswork. As a developer, the experience around experimenting in a local or development environment is great because of the speed of implementation and limited blast radius of the changes/ experiments. Though, to get actual production data and provide important feedback loops, feature flags are key to supporting DX in experimentation. The ability to experiment might not fit with the UX mappings to DX, but more of an intrinsic experience with more ability to help shape and mold product direction, for example.

Like the scientific method taught us, we need to be able to capture data and have a baseline/control when we run experiments. Having an idea of what is normal vs. regression for an application can actually be another challenge; going back, usually no one person works their entire career on one application.

Managing Feature Flags



Understanding Normality in Applications

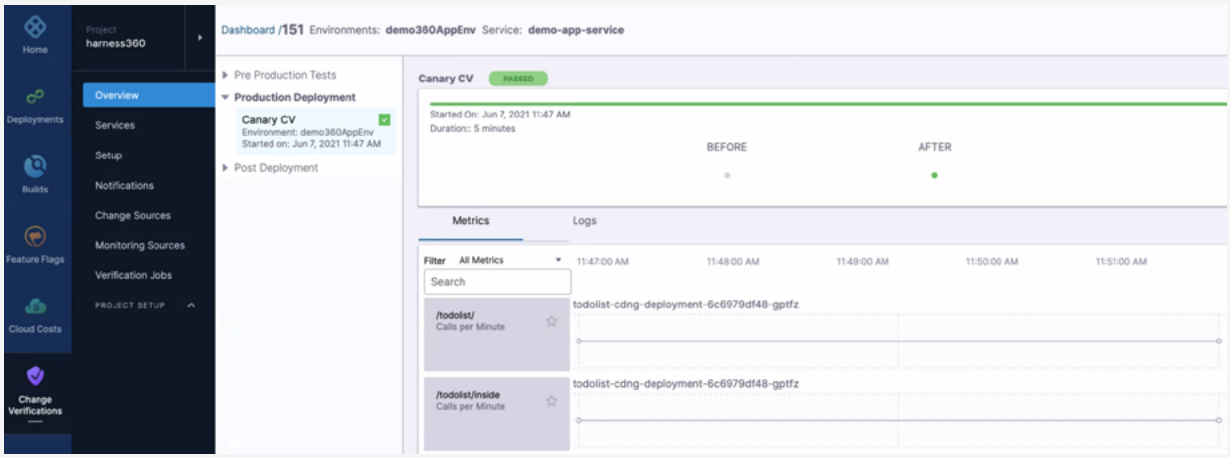
A common challenge for developers is having a clear picture of what is normal performance behavior for a distributed application. Going back to engineers being natural optimizers, when starting a project, a generic overarching goal of improving performance or stability is usually in the backlog in some shape or form.

The question of “who owns the performance” of an application can be like playing a game of hot potato. Ranging from operations engineers to performance engineers down to the developers who wrote the feature, everyone has expertise to offer. From a developer’s perspective, unless the developer has production-facing or on-call responsibilities, disseminating this information can be tricky.

Even in lower environments, when introducing a change, performance impact or regression can slip through the cracks until getting closer to production. The adage “slowness is the new down” is very true for internal and external customers alike. Disseminating information that was not easily unlocked before is core to the findability and credibility portions of Developer Experience.

One of the last chasms to cross around Developer Experience supporting both the findability and credibility portions is solving for cost and density optimization.

Continuous Verification Making Automated Judgment Calls



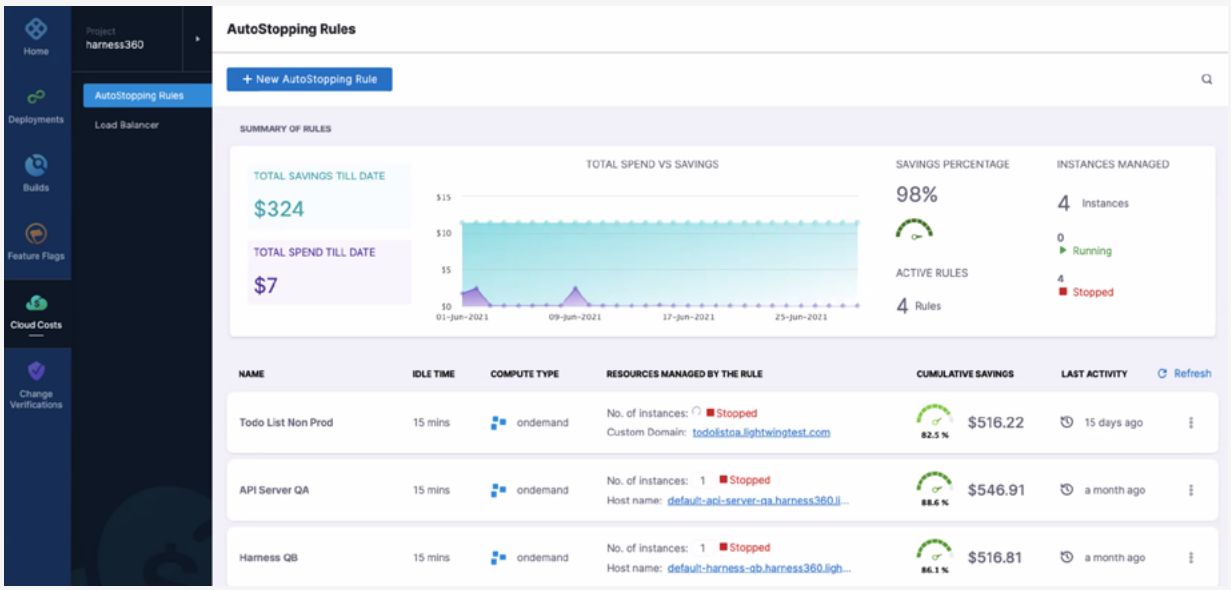
Continuing On the Optimization Journey — Cost and Density

As systems grow more distributed, so do the lower environments that support distributed development and the increased flurry of iteration. It is a two-part problem to solve for. How does one optimize lower environments, and how does one optimize production environments? Providing functionality to assist in the optimization and dissemination of information fosters a better DX.

Lower Environments

One of the quintessential engineering jokes is that turning off is easy but turning back on is hard. Supporting development infrastructure and distributed environments can be quite costly. Shutting down resources on the flip side brings one more delay: eventually, developers will need to spin back up the resources. Because of advancements in compute and container technology, this could be slightly faster than years gone by. Development environments can consistently criss-cross distributed infrastructure to start back. The ability to autostop and also autostart workloads can help support the “what if I need it again” point of view.

AutoStopping/Starting Rules

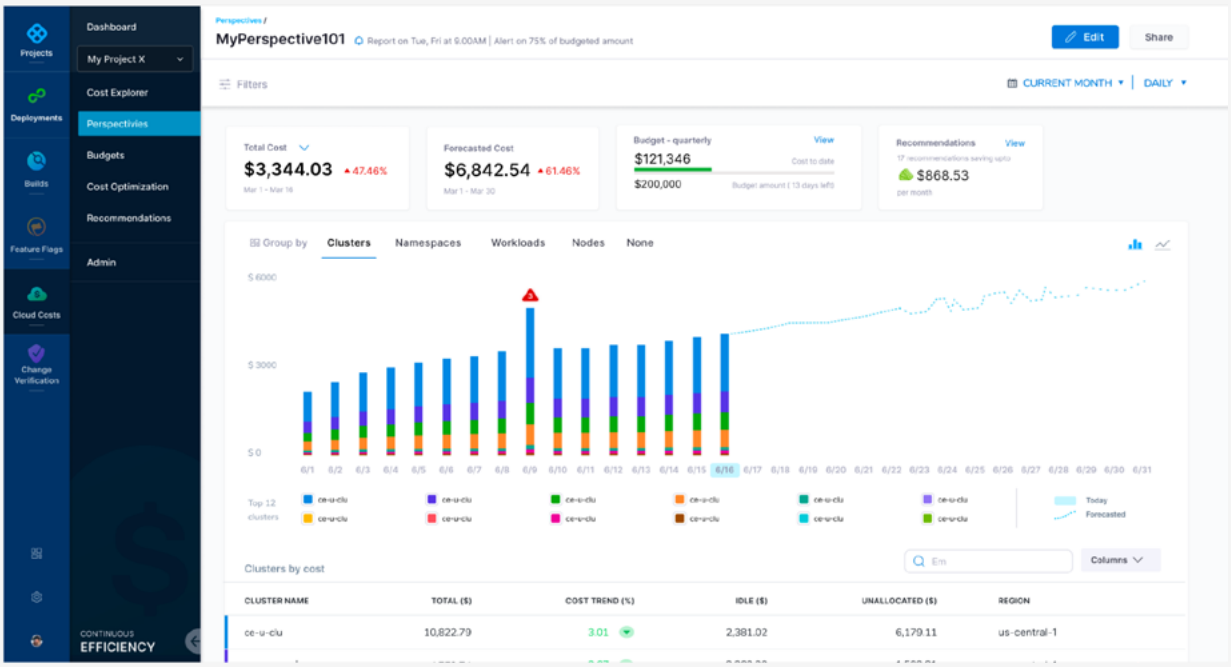


Continuing On the Optimization Journey — Cost and Density (Cont.)

All Environments

Having clear and concise information on public cloud spend in how it pertains to specific applications and services can be an exercise in data aggregation and mathematical calculations. The ability to unlock and disseminate usage, density, and cost information pertaining to public cloud workloads allows for all engineers to make optimizations. Like modern paradigms around shifting left, developers will make the best decisions based on the data they have available to them. Typically, cost information would be locked away by finance orgs — but with these new FinOps methodologies, information is shared and optimizations are applied.

Cost Optimization and Visibility

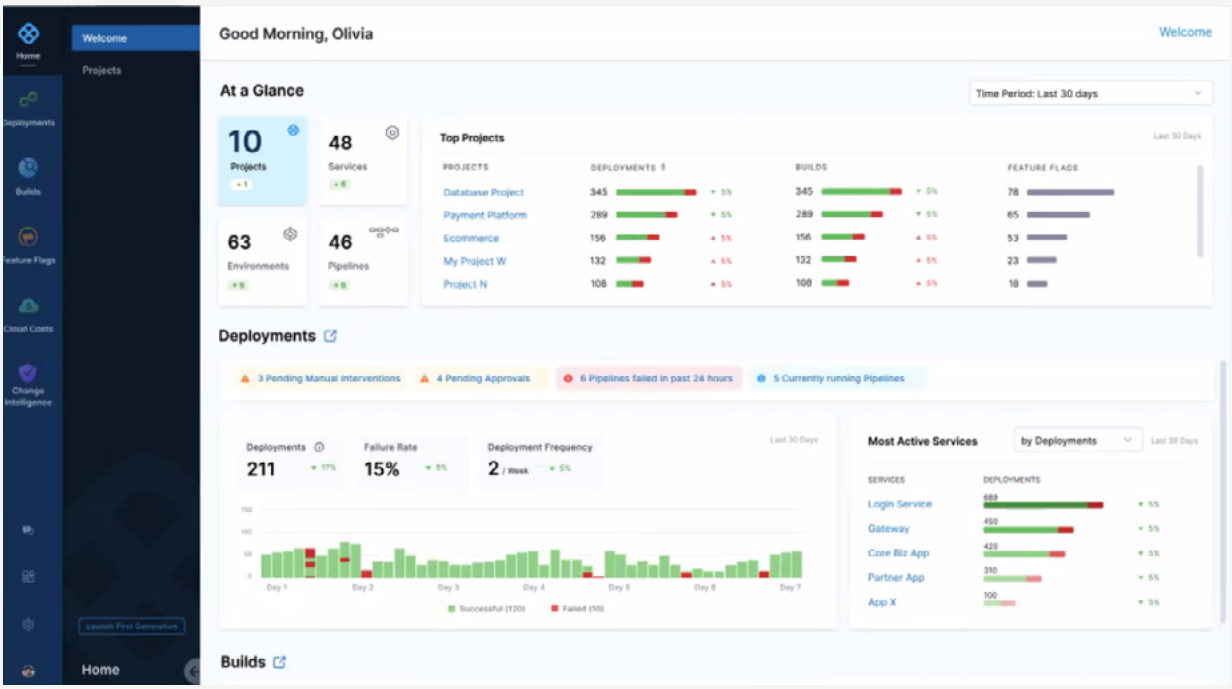


The Harness Platform: One DX From Idea to Production and Back

The Harness Platform offers all of these pillars — and more — to help drive a positive Developer Experience from idea to production, and support as many of those iterative cycles to match the speed of development.

Supporting great developer experience, your four key metrics from Accelerate (deployment frequency, lead time for changes, MTTR, and change failure rate) can all improve.

Deployment Metrics



In Conclusion

Developer Experience is key to building and maintaining the next generation of software and platforms. As technology proliferates our lives, your external customers do not care how features reach production. However, your internal customers are the source of those features and improving their experience will improve and nurture the innovation pipeline. As engineers in industry continue to move around to increase their skill and craft, having engineers become more productive in a shorter period of time fosters a culture of learning and innovation.

Author Appendix

Written By:

Ravi Lachhman
Evangelist
Harness

Prior to Harness, Ravi was an evangelist at AppDynamics. He has held various sales and engineering roles at Mesosphere, Red Hat, and IBM helping commercial and federal clients build the next generation of distributed systems. Ravi enjoys traveling the world with his stomach and is obsessed with Korean BBQ.