SAUCELABS

WHITE PAPER

Designing Infrastructure for Continuous Testing and CI/CD

Moving to an effective and efficient CI/CD pipeline requires significant effort from organizations. It involves process and policy changes across your entire organization. The payoff can be extraordinary: continual improvement as the organization moves to constantly deliver high-quality, well-tested value to their customers. This white paper will discuss the approaches, requirements and processes to consider when implementing a CI/CD workflow.

TABLE OF CONTENTS

DESIGNING INFRASTRUCTURE FOR CONTINUOUS TESTING AND CI/CD

Adding integration and end-to-end testing to a pipeline can enable the leap from Continuous Integration to Continuous Delivery (CD). Being effective, long term, with Continuous Delivery requires an entirely new set of features -- testability features -- built into the architecture itself, along with other changes to the way software is built. Without these changes, organizations typically struggle to see the benefits that Continuous Deployment and Delivery promise.

Organizations looking to move to Continuous Delivery would do well to consider improvements in the following categories, looking for "missing features", anticipating the cost of building those features, and the consequences of leaving them blank.

- Fast Management of Environments
- Careful Feature Management
- Adapting Processes
- Embracing Codebase Improvements

FAST MANAGEMENT OF ENVIRONMENTS

Fast-moving teams can't wait weeks or months to get environments for builds and testing in place, nor can they wait for already overloaded DBAs to manually inject data or schema changes. This is simply untenable for projects where builds and testing are ongoing many, many times each day. CD requires environments be created, provisioned, and ready for test in just a few minutes--more than ten or 15 minutes makes it near impossible to have a smooth-functioning automated pipeline.

This need for speed requires significant changes to how infrastructure and environments are managed. Cloud-based hosting offerings like Microsoft's Azure or Amazon's Elastic Computing Cloud (EC2) allow teams to offload their build and server management. Testing on demand in parallel can be easily added to a pipeline through offerings like the Sauce Labs cloud-based testing platform. Services like Sauce Labs also offer physical mobile device farms, enabling teams to avoid management of potentially huge matrices of devices, operating system versions, and features.

Self-Service for Teams

Luckily, technology has advanced to the point where there are affordable, industry standard solutions. In addition to the cloud-based solutions

mentioned above, on-premises solutions range from commercial products like VMWare to open source containers like Docker.

All of the tools, both on- and off- premises, make it simple to wrap in a CI/CD toolset. Scripts, plugins, adapters, etc. to manage infrastructure are available for every popular solution listed above—it's simple to tie provisioning and deployment into a job on Jenkins or other toolsets. Moreover, many cloud vendors provide easy secure tunneling to backend (on-premise) infrastructure to support a mix of on- and off-premises systems. Sauce Labs' Sauce Connect Proxy is just one example of this.

Infrastructure management isn't limited to provisioning and deploying. Fast-moving teams need *self-service*. They need to create an environment on-demand for any given build, and, obviously, their tools need that capability as well. Organizations need to move system management and access rights down from centralized administration to individual teams. This doesn't mean ignoring or bypassing regulatory compliance and auditing issues emplaced by concerns such as HIPAA or Sarbanes-Oxley, it means understanding the power of automation via a delivery pipeline to handle security, artifact storage, configuration, and auditing.

Teams also need to quickly access logs and performance/environmental data from production environments. Traditional, slow-moving organizations often use ticketing systems with long SLA times to handle these situations. That's untenable in a CI/CD environment where teams need to have constant monitoring of their software in production.

Logging, monitoring, and advanced log search tools can also have benefits for testability and debugging. Building this capability into dev and test environments mean it will be "free" in production and enable programmers to perform production support, removing a barrier between development and operations.

Scriptable Databases

Continuous Deployment "but manual database changes" puts Continuous Delivery in quotation marks and quality in jeopardy. Scriptable database changes mean the schema and data can be stored in source control right alongside the appropriate version of the system code. If the database versioning and the production code are separate, then changes need to be coordinated so they don't break production - and they certainly will break

self-service dev and test environments. Even if the changes are carefully coordinated in dev and test, anyone trying to patch an old version of production with new database code will find problems.

Many toolsets offer this scripting for major databases, regardless whether they're traditional RDBMSs or "NoSQL." Tools range from commercial products to open source tools such as LiquiBase and Ruby Migrations.

Scriptable Data

Some applications have an API to create a user, an account, or a product; wrapping that in a command-line application is trivial. Adding the capability to batch-export, clear or delete data in the database and import data in the same way, by logical groups, adds a feature to the product that is incredibly valuable for testability. With this feature in hand, programmers can create test data sets that are imported automatically for any test. The test can then login as those users, perform transactions, check the results - and know when the test ends the system will clear the data and re-import it for the next test. Advanced features to save time include allowing a test to be 'read only' (no need for teardown) or to perform its own, unique setup. Once the team has achieved the trifecta of fast self-service scriptable environments, scriptable databases, and scriptable data, the continuous deployment system can truly run end-to-end tests atomically, without introducing errors and challenges to the deployment pipeline.

FEATURE MANAGEMENT

Too often organizations ignore the impact of business implementation: What features are selected, how they're conceived, and how they're coded up.

Smaller Features

Learning to decompose large monolithic "features" into smaller ones takes hard work on the part of organizations. These changes don't just hit technical issues, more importantly *business* concerns. Stakeholders, product owners, and users all need to understand the advantages of smaller work.

Smaller features are generally less complex, thereby making them easier to build as part of a CI/CD pipeline. It's far easier to build, integrate, test, and deploy a smaller component with a small set of APIs, database changes, and UI versus a huge footprint of massive schema updates, interwoven service dependencies, and numerous components to be installed for the end user.

Small features will be easier to turn off (make switchable). To be successful with larger features under CI/CD, organizations will need to understand and use feature flags to make features switchable -- our next topic.

Switchable Features

Skilled teams are often asked "How do I test CAPTCHA in my user registration form?" as those teams struggle with automating a tool (CAPTCHA) that is specifically designed to prevent automation.

Those same teams will often answer "Don't. Cheat instead."

What's meant in these situations is design your system where you can turn off wholesale features to make the system more testable. In this case, you don't need to test CAPTCHA itself. It's a well-tested, high-quality third-party component. You obviously need to test its integration and functionality, but that's likely a one-time thing.

Automating your tests for your user registration process should first disable CAPTCHA, run the test, then turn CAPTCHA back on. This requires coding and engineering effort to make the feature switchable, but it's an extraordinary help when trying to wrap testing into your CI/CD pipeline.

Moreover, feature switching can be used for much larger chunks of functionality--a technique sometimes called "canary releases." You could deploy features to your entire production farm, but enable them only on a carefully monitored small number of servers. This allows you to gradually roll out your features to a small population, ensure they're working properly, and gradually turn on those features to more and more users.

PROCESS

Moving to a pipeline that constantly delivers high-quality, well-tested software is far more a process change than a technical one. Of course significant changes will be required to an organization's technical practices; however, organizations will need to adjust their delivery processes. This can often be more challenging than technical changes, as it requires non-technical roles such as stakeholders, product owners, and users to adjust their work habits and mindsets. Getting past the "I just want to do my job!" mindset to "How can I improve my job?" isn't always easy.

Technical roles also have to adapt how they're doing their work in order to successfully move to a CI/CD model.

Moving to Smaller Branches

Branching strategy is something often as hotly debated as tabs versus spaces or Emacs versus Vim. The branching strategy a team chooses will have a significant impact on the team's ability to work as smoothly as possible in a CI/CD environment.

Small, short-lived branches (feature or smaller-sized) allow teams or even small groups (pairs!) of team members to do their own work isolated from others' churn in the codebase. Development and testing can be accomplished without interruption from other work in the database. Small branches aren't something new: Martin Fowler wrote about them back in 2009 based on years of his experience. Dependency management across branches is still an issue, but it's lessened by good communication around APIs and automated tests to guard against regressions.

Normally each branch will have its own build job in the pipeline. Each branch has a job to monitor that branch in source control, then pull, build, deploy, and test as required. This requires more setup time up front; however, job/task templates make this much easier. (Every popular CI/CD toolset offers some form of job/task templating just for these situations.).

Tests in Source Alongside Code

Keeping automated tests in sync with the application code they cover is critical to automated delivery pipelines. The entire team needs to have complete confidence in the checks guarding against regressions and confirming high-value business features.

The smoothest approach to handling this is simply keeping automated test code in the same repository as the system code. Having tests right alongside the system makes it simple to pull the current branch from source, build, deploy, and test. There's no mess about determining versions from other repositories, passing of messy variables, etc.

Organizing test and system code in a repository is something each project team needs to work out for their own requirements. Larger organizations might have a few guidelines (not "best practices!"), but teams will need to evaluate approaches that meet their environments and pipelines. Moreover, different toolsets prefer different organization of tests. A common approach is to have unit tests very close in layout to the code they're testing, as shown in the following figure depicting a very simple Java project in Eclipse. Source code is under `/src/main` with test code under `/src/test`.

Further organization of integration, functional, security, performance, and other test types generally has those in separate projects. Again, approaches vary greatly across different organizations, teams, and toolsets.

CODE

Lastly, but far from least, are the technical impacts in a codebase to ensure good testability in a fast delivery pipeline. These issues aren't trivial, but they certainly come after the enabling concepts laid out earlier in this paper.

Leveraging APIs for Testability

Public APIs are a crucial piece of any well-architected system, regardless of whether it's a monolithic system or one composed of numerous service-based components. Those APIs provide a terrific way to dramatically improve automated testing of a system as it's moving through a delivery pipeline.

This article earlier mentioned dual-purpose features providing test data or feature switching. Automating calls to these features is best done through an API, versus re-writing configuration files or injecting changes into a database. Using APIs for this approach ensures appropriate business rules for the general feature are followed.

Such an approach is critical when setting up an environment for automated and/or exploratory testing. This might include steps such as validating that parts are in stock and available when pulling part data from a data warehouse. It might ensure only active customers are exported from a database. It could also appropriate pre-requisite steps when creating unique data for automated testing.

Imagine a test that checks if a customer can search for a particular item and place it in a shopping cart. All good automated tests avoid sharing state and data between themselves due to the extreme fragility of such an approach.

This requires all data for a test to be randomly or uniquely generated. Pseudo code for setting up prerequisites might look similar to this:

```
Create_test_customer
        Randomly generate customer name, address, etc.
        Call system APIs to create a customer with random data
        Create_test_store_item
Generate an item with random name, description, etc.
        Call system APIs to create a store item with random data
        Create_test_customer_cart(test_customer)
Call system APIs to create a cart for test customer from above
```

Each pseudo method uses something like the Faker library to randomly generate appropriate data and in turn calls the true system APIs to create real objects with the randomly generated data. Again, the system APIs do all the proper validation (is the new customer's phone correct? Is the store item's price correct? etc.), relieving the team of having to rewrite and possibly inject bugs in their own prerequisite or validation code.

Managing Code Dependencies With Software Craftsmanship

Perhaps the most fundamental concept for fast-moving, CI/CD environments is managing dependencies at the lowest level of the system. Using sound Software Craftsmanship principles ensures external services and dependencies can be properly mocked or substituted in various environments through the delivery pipeline.

Software Craftsmanship is a complex, varied school of practice for software construction. While there are many tenants to it, including a Manifesto, one of its basic principles is ensuring software is flexible and adaptable. Part of that concept is handled by ensuring dependency management is carefully thought out and implemented. Using one form or another of dependency injection means no component is responsible for creating dependencies it relies on. Instead, those components have their dependencies injected or passed into them.

Injecting dependencies on external services is one way teams can stand up systems in lower environments without being reliant on those external systems. As an example, imagine a payroll system. Editing an employee's hourly rates or annual salary should require a security check to ensure the user doing the editing is indeed authorized to do it. That security check likely relies on some system outside the payroll system—a larger human resources system, for example.

With a properly architected system, a test of the employee wage edit feature could simply swap out a call to the "real" security system for a fake call that simply approves a test user for the edit. This cuts the dependency to that external system, ensuring tests could run in lower environments, or within unit tests themselves.

CLOSING

Moving to a well-tested CI/CD pipeline requires significant effort from organizations. It's not only a technical issue, it involves process and policy changes across your entire organization. You'll need to bring stakeholders and your business users into the fold as they'll need to adapt their own mindsets and cultures. You'll need to change, sometimes dramatically, your processes for handling infrastructure. Finally, you'll need to raise up the skills of your entire delivery team as part of the effort.

While this can involve years of effort, the payoff can be extraordinary: continual improvement as the organization moves to constantly deliver high-quality, well-tested value to their customers.

SAUCELABS

ABOUT SAUCE LABS

Sauce Labs is the leading provider of continuous testing solutions that deliver digital confidence. The Sauce Labs Continuous Testing Cloud delivers a 360-degree view of a customer's application experience, ensuring that web and mobile applications look, function, and perform exactly as they should on every browser, OS, and device, every single time. Sauce Labs is a privately held company funded by Toba Capital, Salesforce Ventures, Centerview Capital Technology, IVP, Adams Street Partners and Riverwood Capital. For more information, please visit saucelabs.com.

saucelabs.com/signup/trial

FREE TRIAL

SAUCE LABS INC. - HQ 116 New Montgomery Street, 3rd Fl San Francisco, CA 94105 USA