Automated Testing for Continuous Delivery Pipelines

Justin Wolf, Scott Yoon

justinwo@cisco.com, scoyoon@cisco.com

Abstract

Today's continuous delivery (CD) pipeline is a collection of many technologies that ensures the highest quality product reaches production. As a new feature moves from concept to production, it goes through several phases of testing. However, for the CD pipeline to remain efficient, the bulk of this testing must be automated. Unit tests, load tests, integration tests, and user interface tests all require automation infrastructure. In our product, we use Gerrit, Jenkins, Gulp, Protractor, Jasmine, Docker, and Kubernetes among other supporting technologies to perform automated testing on every release, execute long running soak tests, and support rapid iterations on deep performance tuning activities with complex cluster configurations. This paper illustrates how to weave together these testing technologies and integrate them into the CD pipeline so that every release maintains high quality while continuously delivering value to our customers.

Biography

Justin Wolf has worked in software engineering since 1988 as developer, architect, product manager, and engineering manager. His employers have ranged from game companies, to government contractors, to computer networking giants. At his current employer, he manages a team of server and embedded device software engineers to provide home network management solutions to large Internet service providers. These SaaS products help service providers support their millions of customers to reduce cost and improve retention. He enjoys aviation and sailing in his spare time and lives in southwest Washington with his wife, daughter, and pets.

Scott Yoon has experience in international quality assurance, quality assurance, and quality engineering throughout his career at Adobe Systems, KIXEYE, and Cisco Systems. His works have covered many aspects of software quality using both manual testing and developing automated test solutions. He likes to travel, hike, and swim whenever time permits. He recently moved to Oregon to unfold the next chapter of his life.

1 Introduction

These days, just about every service and application across every platform is either already continuously delivering updates to users or seeking ways to shorten update cycle time. Until recent years, many products were hindered in this pursuit by keeping development and testing resources in separate silos and by a general lack of high fidelity automated testing. These two problems meant that overall test cycles were usually lengthy and frequently incomplete. A better way is to bring testing resources into the development team and integrate development and testing into the same continuous delivery flow.

Our team began working on a new product approximately 18 months ago, which allowed us to make a clean break from the previous development and test environment of our prior 8 year old product. However, even legacy environments can be retrofitted with the techniques used in our new pipeline and converted from heavy-lift upgrades to continuous delivery, regardless of deployment style.

We [*throughout this paper, "we" refers to the authors and/or their product development team, --ed.*] invest heavily in automated testing in the form of unit tests, user interface (UI) tests, and end-to-end (E2E) tests. Though creating and maintaining the automated tests and associated infrastructure is nontrivial, the return on investment is enormous because we can confidently release updates to our users many times a day without worrying that some seemingly unrelated piece of the product is now broken.

In this paper, we start by discussing our product development methodology and why automated testing is so critical to our continuous delivery (CD) pipeline. This is followed by detailed discussions of the technologies and techniques we use to automatically test each build, even as a part of the code review process. Finally, we'll lay out our plans for continuing to improve our test infrastructure and coverage as our product becomes more established, complicated, and critical to our business.

2 Lean Continuous Delivery

Before talking about how we test our product, it's important to understand how our team builds products because this helps explain what kind of outcome we're hoping to achieve and why we test our product the way we do. Our product development philosophy combines Lean and Agile Development, Design Thinking (Cross 1982, Wikipedia 2016), and other best practices for two main reasons. First, we operate as a commercial endeavor that must be profit optimized. While there are many indirect aspects that affect profit, one place we can directly influence it is avoiding unnecessary effort (e.g., avoid building things people won't use and avoid releasing buggy code). Second, we want to build features that customers enjoy using. Delightful products generate more revenue and the sooner we achieve this level of refinement, the higher our overall profit will be.

We blend together many aspects of our philosophy when building products (Figure 1). We break these into *principles*, *skillsets*, and *facets*. *Principles* are:

- **Skill sharing:** We believe that *skill sharing* enables us to grow as a team. If each person on the team is the only person who knows about something, our team is weak. Weak teams have lower overall velocity and the ability to intuitively understand the best trade-offs as we implement new functionality is limited. Additionally, teaching a skill increases personal mastery.
- **Lean:** We believe that the best way to rapidly deliver value to our market is through relentlessly eliminating waste. By eliminating waste, we will be able to improve our velocity, our market accuracy (i.e., delivering the right thing at the right time), and customer satisfaction.
- **Design thinking:** We believe that the best solutions come from empathizing with the consumers of our product, whether they be users, requestors, benefactors, or anyone else our product affects, and that the best approach to this is *design thinking*. For us, design thinking leads directly to our internal value flow (see Figure 2). Design thinking provides us a view from the consumers' perspective that directly leads to a higher quality product that people enjoy working with.
- **"Us":** "Us" is a catch-all for the entire universe; everything influences our team, product, market, etc. *Everything* includes everything knowable and unknowable. Everything is everything.

However, four things in particular have the greatest impact on our team: Our people and their experiences, the history of Agile development, our company culture, and our market.

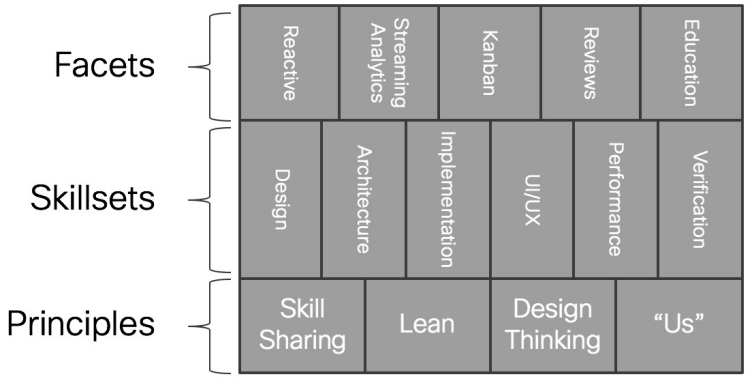

Figure 1 - Aspects of our philosophy

Skillsets are the various functional disciplines of our team:

- **Design:** This describes the shape and behavior of our product. This skill set considers which features to implement, which to skip, how those features function (i.e., the features of a feature), and the user experience. While a product's user interface is its veneer, *design* is the overall experience it provides. Put another way, even the most beautiful UI won't fix an ugly design.
- **Architecture:** Architecture is to the insides what design is to the outsides of our product. It is the internal structure of the product. Architectural choices include which technologies and techniques (e.g., reactive) we adopt and how those are woven together and connected to create the framework within which we implement new functionality.
- **Implementation:** This is the actual writing of code, independent of other skill sets. In other words, it is the *how* of implementing a new feature, not the why or what, and writing code that adheres to best practices.
- **UI/UX:** This describes our direct human connection. Again, not all humans interacting with and affected by our product are experiencing it via the web UI. User interface and user experience (UX), like design, considers all users' concerns and all external effects of our product. UI/UX is closely aligned with design and architecture because poorly structured features will lead to worse experiences. UI/UX isn't just graphical user interface (GUI) functionality; it also includes APIs, file formats, and log messages.
- **Performance:** We believe that performance is the computational efficiency of our product. Performance is mostly concerned with raw numbers: how fast, how small, and how many. We want to build products that are enjoyable to use and, when it comes to computers, high performance is expected by humans.
- **Verification:** Verification ensures we deliver what we intended to build. Verification is asking, "Did we build it correctly?" (in contrast with *validation*, which answers "Are we building the right thing?"). Not only does this mean we build bug free (enough) products, but that our products do what we say they do; no more, no less.

Finally, *facets* are the best practices that we integrate into how we engineer our product:

- **Reactive (product resiliency):** We believe the tenets of the Reactive Manifesto (Bonér 2016) enable our design, architecture, and implementation to be an inherently robust product. By maintaining grace under load while simultaneously self-scaling to accommodate spikes, reactive designs are better at handling modern utilization patterns. While initially more complicated to build products in this manner, it ultimately results in less effort, more profit, and happier people (both on our team and in our market). Less effort comes in two forms: less time spent supporting a poorly built product and less time spent re-architecting the product to support higher scale.

- **Streaming analytics (team expertise):** Our team's unique value at our company is our specific focus on and expertise with *streaming analytics*. We use the best technologies and techniques to enable streaming analytics for our customers so that our team and our products are adept at providing real-time analytic results from massive amounts of data.
- **Kanban (status communication):** For internal status updates, we use *kanban* (Atlassian 2016) as our task management process because it is lightweight and is well suited for continuous delivery pipelines. Since we are pushing updates to production on frequent and irregular intervals, we need a way to manage tasks in a similar flow. To do this, we use Trello (Trello 2016) to record the status and description of pending, in process, and completed work and report that out at all times.
- **Education (technologies and techniques):** The best products come from highly-trained engineers that leverage the best technologies and techniques to solve our market's problems. We use internal training, vendor support subscriptions, and conferences to keep our skills fresh.
- **Review (ensuring quality):** We consider *quality* to include: Form, Function, and Necessity (i.e., does a feature increase profit?). To ensure quality, we review each change with respect to each skillset: Design, Architecture, Implementation, UI/UX, Performance, and Verification.

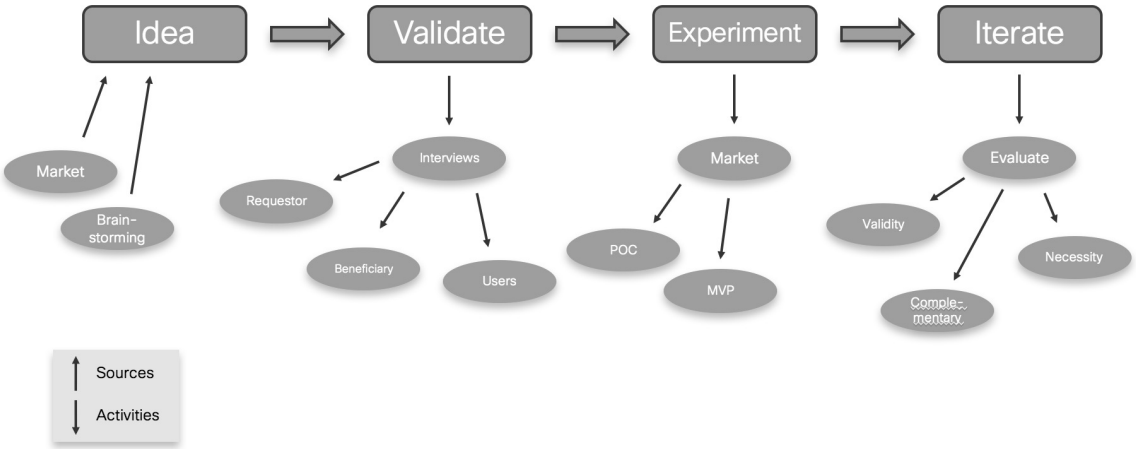

Figure 2 - Internal value flow

Our role is to take an idea and deliver it as value to the market as quickly and profitably as possible. Each step of the internal value flow (Figure 2) maximizes the likelihood that we are working on the right tasks, at the right time, to the right level of refinement, and nothing more.

- **Idea:** An *idea* is a described need. The need comes from the customer (market), but the idea might come from either the customer or internal brainstorming (i.e., ideation). In many cases, the underlying problem may not be understood or even known by the customer. In these cases, we must engage with the market to understand the need so that a precise solution can be built. We will manifest the idea both verbally and visually. Since verbal descriptions are often inadequate, visual engagement with either 2D or, preferably, 3D sketches is necessary. These descriptions are developed jointly with the customer to illustrate the problem needing to be solved, one or more proposed solutions, and the root "whys" of both the problem and its solution. The 2D/3D sketches will also be useful during the interviews in the next stage.
- **Interview:** Interviews are at the core of our process for validating ideas because the development team itself is usually unable to completely assess requirements simply by reading a written request or, worse, inventing it on their own and proceeding directly to implementation. Even purely internal technical tasks need to be scrutinized to ensure they are being built with precision (right scope, right time). The team interviews various personae in the pursuit of understanding the actual problem to be solved. At a minimum, we interview: *users* – the people who directly interact with the feature; *requestors* – the people who asked for the feature; and *beneficiaries* – the people who indirectly benefit from the feature existing.

- **Experiment:** We use the idea sketch and the interviews to identify the next step toward providing a solution. We strive to identify only the absolute minimal progress that must be made so that interviewees can provide feedback on direction. Ideally, this will be 1-2 days of implementation time. This is even less than minimum viable product (MVP). We're targeting a proof of concept (POC) of the next increment. With enough iteration, it will reach MVP. Further iterations, if necessary, will refine out the solution. If the level of effort is beyond 2-3 days, we simplify the criteria instead of spending more time on implementation. Remember, the goal is to spend as little time as possible moving in the wrong direction, which includes delivering a perfect solution too late. The *experiment* stage is where software engineers traditionally spend most of their time, so we want this time to be spent effectively.
- **Iterate:** Once the experiment is ready, we demonstrate it to the interviewees. We compare their feedback to the evaluation criteria to assess if we're moving in the right direction and to ensure that the solution is valid, necessary, and complementary. *Valid* means that the feature's design makes sense for the market. *Necessary* means that the market will pay enough for the feature to satisfy return on investment requirements. *Complementary* means that the feature fits nicely with the overall vision of the product. Lastly, we use the result of the demonstration to determine if another iteration is necessary (i.e., "Are we there yet?").

As you can see, bolting automated QA onto an existing product is insufficient. Though complex, all of these aspects (and others, of course) must be leveraged to efficiently deliver high quality products to the market. Weaving Lean and Agile Development as well as Design Thinking through our philosophy, while layering on relevant disciplines and best practices, allows us to minimize wasted effort. Together, this results in higher velocity, lower cycle times, and shorter test/deploy lead times and provides a foundation for automated QA to deliver maximum value.

3 Automated QA

Automating quality assurance (QA) steps for CI/CD pipelines requires correct technology selection as well as adapting QA techniques to the CI/CD environment. We use several testing layers ranging from UI-only testing using mocked data to full end-to-end testing with real or simulated data. Various technology combinations support each testing layer.

The bulk of our UI testing is done using mocked service responses because these tests are easier to write and maintain, execute much more quickly, and provide highly targeted results (i.e., if a test fails, it's easy to know where and why). However, we also use end-to-end testing to test all the service components together for both build verification (quick tests) and covering the greatest number of components (longer running tests).[1]

3.1 Testing Technologies

Our web UI is built on AngularJS (AngularJS 2016) and Bootstrap (Bootstrap 2016). The UI retrieves data from our servers using REST interfaces that respond with JSON formatted data. This is a very common setup for today's web applications, but guides technology selections because of their specific focus on these aspects of our product. For non-AngularJS/Bootstrap UI stacks, other technologies that accomplish the same function might be preferred (for example, Selenium itself is a very full-featured automated testing tool that works with a wide variety of data and UI technologies).

Our CD pipeline (Figure 3) starts with code review and ends with deployment. Within this pipeline, various processes invoke testing functionality to ensure build viability. There are three main portions of the pipeline where tests run.

[1] End-to-end testing results in tests that cover the largest number of product components. However, they do not typically result in the highest amount of code coverage unless allowed to run for considerable duration. This lack of coverage is reduced by using fuzzing and long-running tests that exercise edge cases and other infrequent scenarios.

- **Code Review:** When a developer submits code for review, Jenkins builds the product to ensure the code does not break the build. Since we use unit tests throughout our product, this build also ensures all unit tests pass.
- **Post-merge:** After the code review is approved, it is submitted through Gerrit and Jenkins builds the product for deployment. If any of the tests fail, the build fails, and the artifacts are not pushed to Docker Registry. So that builds do not take a very long time, these tests are restricted to UI tests using a mocked environment.
- **Pre-deployment:** Before containers are distributed to Docker Registry, a complete set of automated tests ensure that nothing in the product has broken. These pre-deployment tests do a full test deployment of every container including the data simulator and UI testing components using Gulp and Protractor.

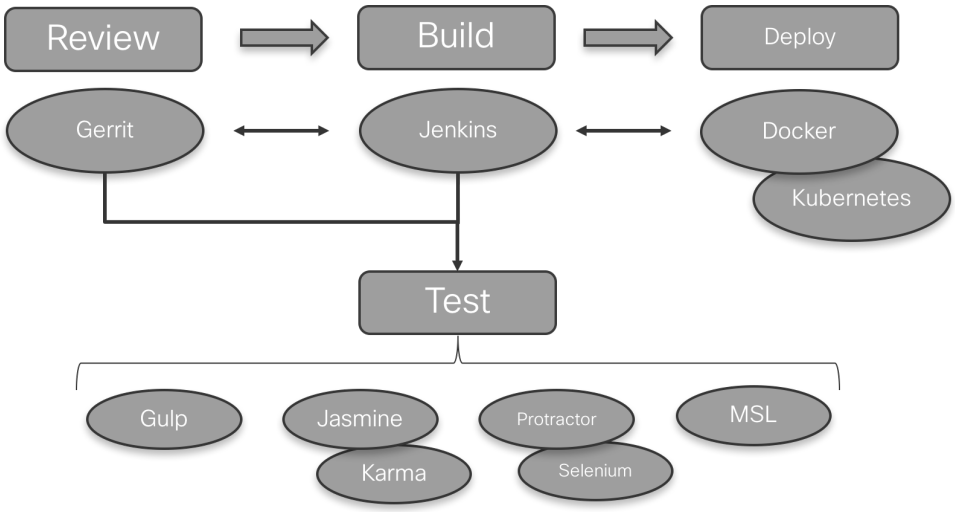

Figure 3 - Build and Test Pipeline (selected technologies)

With our pipeline, we're able to ensure high quality throughout the process starting even before developers check in their code. This is because their code must pass all unit tests and build verification tests before Gerrit will allow a non-draft submission. However, it's a complicated system that comes with a wide range of technologies and integration complexities. The alternative is slower throughput and reduced CD velocity. In some cases, that's acceptable, but when there's a high-priority customer outage that needs to be fixed in production as soon as possible, high velocity is critical.

Gerrit, Jenkins, and Docker work together in concert to ensure product quality. Gerrit uses Jenkins to build the product, which includes automated testing even before new code can be checked into the main codebase. These tests require components to be built as Docker containers so that they can be executed in virtual machines as a part of build verification. Since these containers are identical to the artifacts that will eventually be pushed to production, we're testing bit-for-bit identical versions of the code we want to release.

3.1.1 Jenkins, Gerrit, and SBT

We use Jenkins as our build system orchestrator and Gerrit to manage our code reviews (Gerrit 2016). When a developer wants to submit a change for review, Gerrit triggers Jenkins to ensure the change doesn't break the build. The build process includes not just unit tests, but also some quick running automated tests to provide more coverage even before the code review request can be raised. Lastly,

since we write all our code in Scala[2], we use SBT (which stands for Scala Build Tool) (Lightbend 2016) to assemble runtime packages.

3.1.2 Docker and Kubernetes

All of our components run in Docker containers. These include proprietary product components, database servers, and frontend web servers. We use Docker Registry to distribute containers from the build system to internal and external deployments. (Docker 2016)

To control our clustered deployments, we use Kubernetes (Kubernetes 2016) pods and services to manage our component containers. Though not strictly a testing technology, Kubernetes provides automated cluster management so that production-grade automated E2E testing can occur as part of the build process.

3.1.3 Data and REST Call Generator

We created a highly-scalable traffic simulator to generate high load levels for stress testing deployments. It's also used in lower volume configurations to perform verification tests. The traffic simulator generates the proprietary data we receive via our southbound interface (the part of our system that collects data), inserts data directly into our database to simulate large amounts of historical data, and performs queries and UI calls via the northbound (web) interface. In the future, we will also add a web interface to this simulator so that automated end-to-end testing can generate and verify that specific data makes it all the way through the system as expected.

3.1.4 Gulp

We use Gulp (Gulp 2016) to orchestrate our JavaScript build process, which includes executing our automated tests. For us, Gulp functions similarly to Make, Maven, or SBT. However, in addition to just packaging our JavaScript files (e.g., cleanup, removing debug code, minification, and obfuscation), it also executes our automated tests by starting the test tooling and then the tests themselves. We have Gulp start by launching JShint to detect errors and find other problems with our test scripts. Next, it launches the MSL server (see 3.1.7) and kicks off the automated tests.

A nice feature of Gulp is that during the dev/build loop, it watches for changes to source files and automatically updates the browser so that developers can evaluate changes instantly. These files are loaded without going through any minification steps. This makes it easy for developers to identify exactly where in their code the problem is occurring.

3.1.5 Jasmine and Karma

Jasmine (Jasmine 2016) provides a framework for writing and running behavior driven development (BDD) and test driven development (TDD) tests for web UI. Supporting both parallel execution and running specific subsets of tests, it supports our commitment to BDD/TDD. We also find it extremely powerful, but easy to use.

Karma (Karma 2016) is a Jasmine test runner. With Karma, developers choose a JavaScript engine to test against (e.g. PhantomJS, Chrome, or Firefox) and Karma runs each Jasmine test and reports the results. Whereas Jasmine is the BDD/TDD test environment, Karma is the tool that actually runs those tests.

[2] Our team has used Scala for more than 7 years for multiple reasons: As practitioners of the Reactive Manifesto, we wanted the best access to Akka, Play, Spark, and other libraries that support Scala first; we take full use of its type system; we leverage various functional programming aspects such as object immutability; and we rely on its compile-time checks to ensure we aren't accidentally creating any unsafe runtime code.

3.1.6 Protractor

Since we use AngularJS as our web application framework, we use Protractor (Protractor 2016) as the end-to-end (E2E) test framework. Though E2E tests can be difficult to maintain, the benefit of testing an entire E2E dataflow for our application across various test cases is critical. Protractor drives Chrome (Google 2016) running in a virtualized X environment (Ho 2015) to verify web UI functionality and E2E tests. We also tried to use PhantomJS, but it does not support some AngularJS code operations our product requires.

Even though we have optimized the tests and run them in parallel (presently we execute 16 tests at a time), a drawback to this approach is that every test runs via the Selenium server (Selenium 2016) and uses real service components throughout. Since the execution time on these tests can be an issue, we only run a subset of quick-running tests during the build (i.e., compiling/containerizing) process and then longer-running tests in a post-build timeframe (e.g., in a lab soak[3] test that runs for hours or days in parallel with deployments).

3.1.7 Mocking Service Layer (MSL)

The Mocking Service Layer (MSL; pronounced "missile") (MSL 2016) gives us the ability to test the UI without needing a full service deployment behind it. MSL uses the Node.js server to provide a fake web server that provides mocked JSON responses to the UI under test.

MSL, and mocking in general, provides several advantages:

- We can test the UI layer without worrying if a failed test was due to something buried within our application. It gets rid of the databases, containers, virtual machines, networking, and everything else that is required to provide real JSON. However, this also means that if the UI test will cause a legitimate failure in the application, it won't be found until later in the test cycle.
- Because we don't need the service to be running, the UI for a new feature can be tested before the service is ready. We still do end-to-end testing and integration testing to ensure that the use cases supported by the UI are actually supported by the service, but completing these is decoupled from the early UI development.
- Edge cases that are difficult to trigger in the service are trivial to mock. This helps us make sure that rare events function as expected in the UI without needing to set up complex test cases within the service to encounter them.
- Since the service isn't handling the REST calls, the responses coming from MSL are nearly instantaneous. This means that we can complete a testing iteration in much less time, have better coverage[4], and improve iteration cycle times.

3.2 Testing Techniques

3.2.1 Unit Testing

Unit testing generally tests only a single piece of functionality in a single way. Multiple unit tests combine to fully test a single piece of functionality. We don't consider anything more than this to be unit testing, but more likely some form of integration testing. We use Gulp, Karma, and Jasmine for our UI unit tests and ScalaTest for our server code.

---

[3] A *soak test* executes for many hours or days (and sometimes longer) to explore what happens with long uptime durations. Build and deployment oriented testing is essentially ephemeral where the components are only up for a few minutes. This doesn't allow development teams to know with any certainty what will happen when the application is ultimately deployed into production and then left to run there for a long time before components are ultimately restarted.

[4] Mocked tests run much faster than full end-to-end tests. Since we want to minimize the time spent testing during the build process itself (i.e., the time actually building product packages), a mocked test environment means we can run more tests in the same amount of time. Therefore, mocking allows us to have greater code coverage at build time than we would otherwise be able to tolerate.

Unit tests are an important first filter for catching faulty code and, since they execute very quickly, can be run by a developer or a build system on every change. Also, by following the TDD philosophy, failing tests that exercise a bug can be put into place before the fix so that resolution is verified and tracked. Unit tests will always have a place in the test hierarchy, but end-to-end testing and other QA layers are necessary to fully test a product.

3.2.2 End-to-end Testing

End-to-end testing (E2E) is the best way to make sure the whole system will work once it's deployed. While unit testing and mocked-data testing ensure that portions of the product work in isolation, E2E is the way we check functionality all the way from data ingest to display.

Our team has developed a scalable data simulator that can provide data to our ingest endpoints as well as hit our REST interfaces to place a deployment under high amounts of load. In automated tests, we don't use real data sources because they are too slow and too difficult to control.

The primary advantage of E2E tests is that they actually exercise the entire product. However, E2E tests have a number of disadvantages:

- Tests are fragile and cumbersome to maintain since test data and application flow is vast and complex.
- Tests take a longer time to run because the entire system must be up and running. There is also the end-to-end lag time to get data through the system because data must actually be processed by real components of the solution.
- If a component (non-E2E) test could have found the fault within a single component that will ultimately result in an E2E test failure, then that component test would save all the E2E build/deploy/test time.
- Requires more investigation as bugs found from the test can come from any part of the system. Developers need to investigate the root cause of test failures as it may not actually be due to the feature being tested.
- Valid changes to datasets or lower-level components can cause test scripts to fail even if they shouldn't, requiring developers to update tests unrelated to their work.

Setting up automated E2E testing is the easy part. Maintaining the tests is an ongoing and perpetual task. However, the payoff is that every release meets quality standards and releases can deploy quite frequently. Additionally, by automating regression tests and tests that exercise edge cases, we can fix those faults before they are detected by customers.

3.3 Lessons Learned

Automated testing at this level of complexity is clearly nontrivial. In our experience, most issues fall into two categories: stale tests and incompatible technologies. Overcoming these issues requires continuous investment and undying commitment to establishing and maintaining the tests and their underlying infrastructure.

3.3.1 Stale Tests

The well-known problem with maintaining unit tests of occasionally needing to fix a test is magnified many fold with automated UI and E2E tests. Sometimes, seemingly minor changes, or sometimes no explicit change at all result in massive test failures. Changing UI resource trees (which the tests traverse to test UI components), updating a technology that uses a different algorithm for naming or ordering objects, or even reinstalling some deep dependency can cause many tests become invalid and need repair.

Add to this the constant code changes by developers. The automated QA engineer often hears from developers, "Oh, I just changed this one little thing." Those developers often don't realize the impact of their changes or consider alternative ways to execute them that won't cause mass test die-off. Even

something as seemingly innocuous as updating a small upstream dependency can change the way things work just enough that tests must be updated.

In the case of E2E test execution, a non-breaking change in one part of the architecture may result in a breaking change in another part of the architecture. Perhaps changing the order of two elements in a list is inconsequential to the software using the list, but it can cause false negatives if the tests are not written with similar order-independent assumptions.

3.3.2 Incompatible Technologies

Early on, we knew we needed a headless browser to run the UI unit tests so that our build process could incorporate them without needing to start up ephemeral VMs with real (virtual) displays. However, we later found out that the technology we'd chosen (PhantomJS) did not properly execute some of our UI JavaScript (JS). This not only meant the tests failed, but that they could never succeed. As a result, we shifted to using a real Chrome browser running with xvfb (Ho 2015) so that Selenium could drive a solid JS engine.

Consider also that many of our technologies are built around using AngularJS, such as our selection of Protector. This means that any decision to change away from AngularJS also carries with it replacing that portion of our test infrastructure and rewriting all the tests that assume AngularJS is part of our architecture, even if the new component otherwise results in no philosophical change to our architecture (i.e., it behaves the same way, but is implemented slightly differently).

Finally, some components in our architecture are stubborn when it comes to adapting them to the Docker/Kubernetes environment. Part of the reason we wanted to use containers is because of the ease of creating and destroying deployments for the purposes of testing within our build process. However, some components don't easily adapt to containerization. Similarly, when we add new components to our architecture, we must also consider their applicability to our environment and our automated QA infrastructure lest our entire product become untestable.

3.4 Example Test Script

This test script shows how Jasmine, Protractor, and MSL all work together to test product functionality. In this test, we're logging in and testing that some data exists on the page that loads immediately afterward. The test case steps are found as comments in the `describe` block.

```
// Requires the MSL client API
var msl = require('msl-client');

console.log("Executing Mock page....");
describe("Mock Signin", function () {
    "use strict";
    var subscriberCount = element.all(
        by.css('div[class*="table-responsive"] tr[pagination-id="search-results"]'));

    describe("Mocking UI Test", function () {
        /* Step-1 Go to Signin page
         * Step-2 Enter user name
         * Step-3 Enter password
         * Step-4 Click submit button
         * Verify 10 subscribers exist in first page */

        var mockResponseObj = {};
        mockResponseObj.requestPath = "/api/authenticate";
        mockResponseObj.responseText = [{"username":"tester","password":"testme"}];
        mockResponseObj.contentType = "application/json";
```

```
        mockResponseObj.statusCode = 200;
        mockResponseObj.delayTime = 0;
        mockResponseObj.header = {"X-App-Token" : "r4nd0mT3x7"};

        it("should sign in", function () {
            // signin mock response
            msl.setMockRespond("localhost", 8000, mockResponseObj);

            // open a browser and go to sign-in page
            browser.get('http://localhost:8000');

            // sign in
            element(by.id('username')).sendKeys('user');
            element(by.id('passphrase')).sendKeys('password');
            element(by.css("form[ng-submit='signInCtrl.submit()'] button")).click();
        });

        // make sure it has what it should have immediately after signing in
        it ("should have 10 subscribers in first page", function () {
            expect(subscriberCount.count()).toBe(10);
        });
    });
});
```

4 Conclusion and Next Steps

Our customers demand the time-to-fix responsiveness that CI/CD can provide, but we can only provide high quality updates through automated testing that's integrated into our pipeline. Building high quality and high value modern web applications capable of high reliability is a complex endeavor where just the automated testing infrastructure and individual tests is complicated. However, it's a necessary step in continuously delivering product improvements to our customers and, without it, we wouldn't be able to maintain the responsiveness they demand.

Now that we have the essential portions of our automated QA in place, we're adding to our system in two ways. First, there are always more tests to add for both new features and to improve code coverage of existing features. When we say "more tests," we don't mean just more of the same. The high-performance simulator that's a key part of our pipeline can only generate random data. This makes E2E testing that evaluates the accuracy of the data presented in the UI impossible. We know we have *some* data, but not necessarily the *right* data. Furthermore, our simulator has the ability to put a deployment under high load, but we don't have any coordinated UI or E2E tests against such a deployment to confirm the system responds as expected under that load.

Second, other types of testing that we can take advantage of within our existing pipeline are integration tests and soak tests. For integration tests, we could, for example, test the database-to-frontend interface in isolation. Presently, this interface is only exercised during E2E testing and a failure will only be caught if it causes a UI-layer failure. Relying on E2E testing also precludes any fuzzing or other intensive interface-targeted testing. With soak tests, we could put the system under high load and observe it over several hours, days, or weeks. While this obviously wouldn't be a part of the CI/CD testing, it still relies on automated testing so that the data feeds and frontend REST calls are realistic.

We're continuously learning how to improve our development methodology to ensure that we deliver the highest quality and value to our customers. Applying these learnings to our pipeline should be relatively efficient since the framework is flexible and highly capable.

5 References

AngularJS. “Angular JS - superheroic JavaScript MVW framework.” https://angularjs.org/ (accessed July 5, 2016).

Atlassian. “A brief introduction to kanban.” https://www.atlassian.com/agile/kanban (accessed July 5, 2016).

Bonér, Jonas; Farley, Dave; Kuhn, Roland; Thompson, Martin. “Reactive manifesto.” http://www.reactivemanifesto.org/ (accessed July 5, 2016).

Bootstrap. “Bootstrap - The world's most popular mobile-first and responsive front-end framework.” http://getbootstrap.com/ (accessed July 5, 2016).

Cross, Nigel. 1982. “Designerly ways of knowing.” *Design Studies* 3.4: 221-27.

Docker, Inc. “Docker - Build, Ship, and Run Any App, Anywhere.” https://www.docker.com/ (accessed July 5, 2016).

Gerrit. “Gerrit code review.” https://www.gerritcodereview.com/ (accessed July 5, 2016).

Google, Inc. “Chrome.” https://www.google.com/chrome/ (accessed July 5, 2016).

Gulp. “Gulp” https://www.npmjs.com/package/gulp (accessed July 5, 2016).

Ho, Toby. January 9, 2015. “Headless browser testing with xvfb.” http://tobyho.com/2015/01/09/headless-browser-testing-xvfb/ (accessed July 5, 2016).

Jasmine. “Jasmine: Behavior driven JavaScript.” http://jasmine.github.io/ (accessed July 5, 2016).

Karma. “Karma - Spectacular test runner for JavaScript.” https://karma-runner.github.io/1.0/index.html (accessed July 5, 2016).

Kubernetes. “Kubernetes - Production-grade container orchestration.” http://kubernetes.io/ (accessed July 5, 2016).

Lightbend, Inc. “sbt - The interactive build tool.” http://www.scala-sbt.org/ (accessed July 5, 2016).

MSL. “Mock service layer.” http://finraos.github.io/MSL/ (accessed July 5, 2016).

Protractor. “Protractor - end to end testing for AngularJS.” http://www.protractortest.org/ (accessed July 5, 2016).

Selenium. “Selenium - Web browser automation.” http://www.seleniumhq.org/ (accessed July 5, 2016).

Trello. “Trello tour.” https://trello.com/tour (accessed July 5, 2016).

Wikipedia. “Design thinking.” https://en.wikipedia.org/wiki/Design_thinking (accessed July 5, 2016).