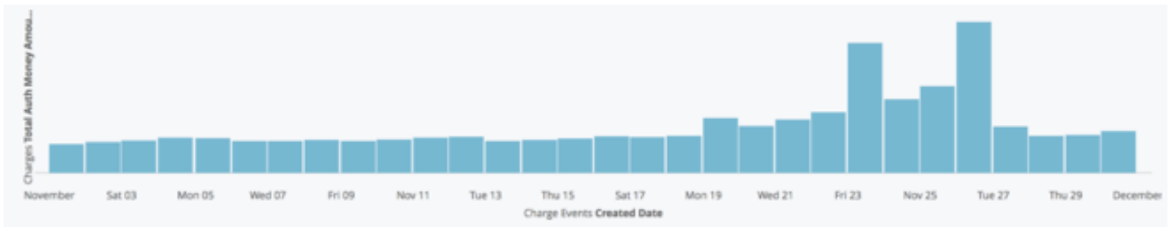
Handling Black Friday scale

Elaine Arbaugh

This year, over 165 million Americans shopped between Thanksgiving and Cyber Monday, and 130 million of them spent at least part of their money online. Given Affirm's partnership with thousands of retailers online, Cyber Weekend is the most important period of the year for us—so it was critical for us to make sure our infrastructure was ready to handle up to five times a normal day's traffic without any downtime or performance degradations.

Preparations for Cyber Weekend don't start weeks, or even months, in advance—our planning for Cyber Weekend 2018 actually started in late 2017, when we applied that year's Cyber Weekend learnings to prioritize infrastructure projects we foresaw would be most important for this year's peak shopping period.

The effort was truly cross-functional, with teams across the organization (including Customer Operations, Risk Operations, and Engineering) working during throughout the year and through Thanksgiving week to support customers and ensure platform stability. Thanks to everyone's hard work, Cyber Weekend was once again a huge success for Affirm! We were able to handle nearly three times 2017 Cyber Weekend's loan volume without any outages or performance degradations. At peak traffic levels, we were performing almost 150 underwriting decisions per minute, and serving over 150,000 promotional messages per minute.



Cyber Weekend loan volume compared to the rest of November

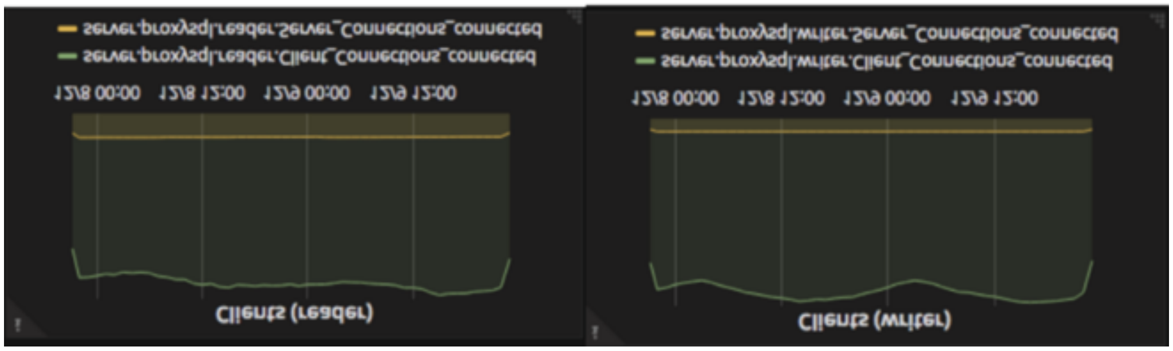
Early 2018: System improvements

ProxySQL

One potential bottleneck on our system was database connections. For the most part, we use Amazon Aurora MySQL databases in our platform, and with Aurora, it's very easy to create multiple read replicas to handle read-only database access. However, with our database access patterns, we access the master databases much more frequently than the read replicas, either because we are inserting or updating data or because we need up-to-date data and cannot tolerate replica lag. As a result, we create a lot of connections to the master database. MySQL can experience performance issues when scaling to handle large numbers of connections since it creates a new thread for every connection, causing high resource usage and significant overhead for thread management and scheduling. AWS also limits database connections on Aurora MySQL to 16,000, and we anticipated crossing this limit this year. In order to handle Cyber Weekend 2018 without downtime, we knew we had to change the way we handled database connections.

We decided to use ProxySQL to handle MySQL connection pooling. ProxySQL also has a lot of other cool features like query caching, slow query logging, query routing, and failover support which we plan to take advantage of later, but we started with just connection pooling since it was the most urgent issue. After debating alternatives, like Vitess; speccing out our solution; writing all the platform and configuration code to set up ProxySQL servers; and testing on our development and stage servers, we slowly and carefully rolled ProxySQL out in production.

With ProxySQL, the ratio of server connections (the actual number of connections ProxySQL makes to the database) to client connections (the number of connections our servers try to create) is between 5–15%. This change gave us a lot more breathing room before we hit any MySQL connections limits, allowing us to support Cyber Weekend traffic.



Client vs server connections with ProxySQL

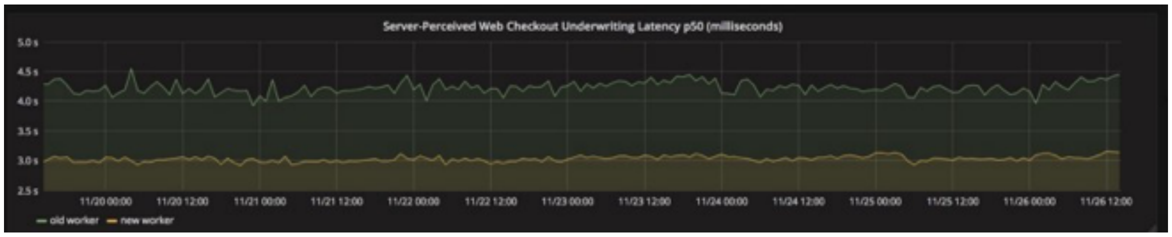
Autoscaling

Another project that was key to our Cyber Weekend success was moving to autoscaling groups for provisioning servers and automating server setup. Previously, if we wanted to add new machines in production to handle additional load, we had to manually run provisioning scripts as well as 10+ setup commands, a slow and potentially error prone process. Before this year, manually provisioning machines was good enough because we generally only had to add small numbers of machines infrequently. With the huge traffic increases we were expecting this year, it became less feasible to manually add the numbers of different types of machines we required for Cyber Weekend, so we decided to move to autoscaling groups. This project involved automating server provisioning using AWS autoscaling groups; setting up the servers, including setting up configs, deploying to the servers, and starting any required processes; and, if required, adding the servers to load balancers to route traffic to them.

We also took this opportunity to switch from using Amazon's Classic Load Balancers to Application Load Balancers and move our machines to the newest generation EC2 instances (specifically, we moved from c3s to c5s). Application load balancers allow routing traffic based on URL paths or host headers, meaning we could switch from routing requests to multiple load balancers using CloudFront rules to only having one load balancer and doing all the routing at the load balancer level, simplifying our architecture. Newer generation EC2 instances are cheaper and more performant, and we saw huge (20+%) latency improvements after we switched over.

After the autoscaling project, all we have to do to set up and run new production servers is change a configuration in the AWS console, and the rest will be handled automatically. This project made it much easier to scale up for Cyber Weekend, and also made us confident that we could react quickly to any capacity emergencies we might have during the weekend.

Additional future work for the autoscaling project includes automatically scaling up to respond to increased traffic instead of having to manually update the number of machines.



Latency for a task on c3 (green) vs c5 (yellow) instances

Tech Debt

In addition to these infrastructure projects, in the months leading up to Black Friday engineers across Affirm worked to make our code more performant, especially on endpoints we knew to be slow or non-optimally written. As Garrett discussed in the last post, Affirm's Culture of Improving Product Quality, we improved our median Application Response time by around 40% in the months leading up to Black Friday. With these improvements, we require fewer resources to do the same amount of work, and we can make more efficient use of our machines. The latency improvements also improve user experience—some people may leave the checkout flow if we take too long to make a decision.

We also made some improvements to help guard against failure conditions. For example, we added timeouts or all our third party requests to guard against the possibility of massive latency increases from third-party requests putting extra load on our servers, which could impact other requests and cause user-facing errors.

The work that everyone in engineering did to speed up our code and more gracefully handle possible error cases was critical to our success on Cyber Weekend.

Early November: Scaling to handle 5x load

This year, over 165 million Americans shopped between Thanksgiving and Cyber Monday, and 130 million of them spent at least part of their money online. Given Affirm's partnership with thousands of retailers online, Cyber Weekend is the most important period of the year for us—so it was critical for us to make sure our infrastructure was ready to handle up to five times a normal day's traffic without any downtime or performance degradations.

Preparations for Cyber Weekend don't start weeks, or even months, in advance—our planning for Cyber Weekend 2018 actually started in late 2017, when we applied that year's Cyber Weekend learnings to prioritize infrastructure projects we foresaw would be most important for this year's peak shopping period.

The effort was truly cross-functional, with teams across the organization (including Customer Operations, Risk Operations, and Engineering) working during throughout the year and through Thanksgiving week to support customers and ensure platform stability. Thanks to everyone's hard work, Cyber Weekend was once again a huge success for Affirm! We were able to handle nearly three times 2017 Cyber Weekend's loan volume without any outages or performance degradations. At peak traffic levels, we were performing almost 150 underwriting decisions per minute, and serving over 150,000 promotional messages per minute.

Handling 5x scale

For many infrastructure components, handling 5x scale is as easy as either adding more of a component or expanding its capacity. For our EC2 instances, we added additional servers by scaling up our autoscaling groups. We distribute our servers between two availability zones to be more resilient—for example, if lightning strikes a data center and takes it offline, we won't go down. We also have duplicate instances for every machine type, so we have no single point of failure in our systems. For databases, we increased the database instance size (for MySQL) or provisioned throughput (for DynamoDB).

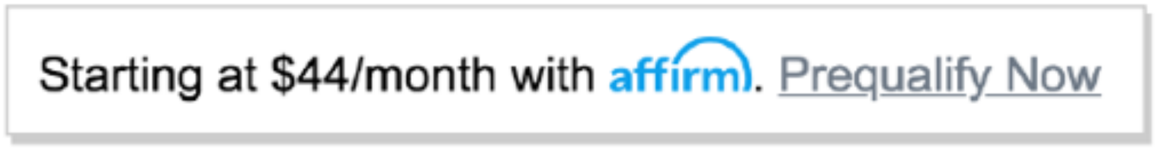
We also worked with our AWS enterprise support team to create an Infrastructure Event Management (IEM) plan. We gathered information about the AWS components that would be experiencing 5x load during Cyber Weekend and discussed what we needed to handle this load. The AWS team helped us reserve EC2 capacity for certain instance types to make sure that we could add machines if we needed them, and also set up prewarming for all our load balancers. Normally, load balancers can start failing with big traffic spikes, but prewarming them preemptively prepares them for the maximum traffic amount we anticipate so they can handle these spikes. The IEM was very helpful to ensure that all our AWS components would scale.

Finally, we contacted all of our third-party partners (for gathering data used in underwriting, creating virtual credit cards, making payments, etc.) to make sure that our traffic increase wouldn't cause any issues on their side.

Load testing

To validate that we added sufficient scale for our promos service, which outputs values for promotional text on merchants' sites based on merchant and user data, we load tested using locust.

Load testing our checkout flow and many of our other endpoints is difficult because so much of what we do requires real user data—for example, hitting external services to gather credit reports or data about a user; user input, like inputting a PIN as part of authentication; or writing data to the database. Since our promos service is much simpler and accesses data in a read-only pattern, it was straightforward to set up load testing. During our maintenance window, we ramped up promos traffic to 5x and monitored endpoint latency, error rates, and system stats for our promos servers. When we saw latencies increase, we added additional web servers to determine if server capacity was the bottleneck, and we used the results of this test to decide how many instances to use for Cyber Weekend.

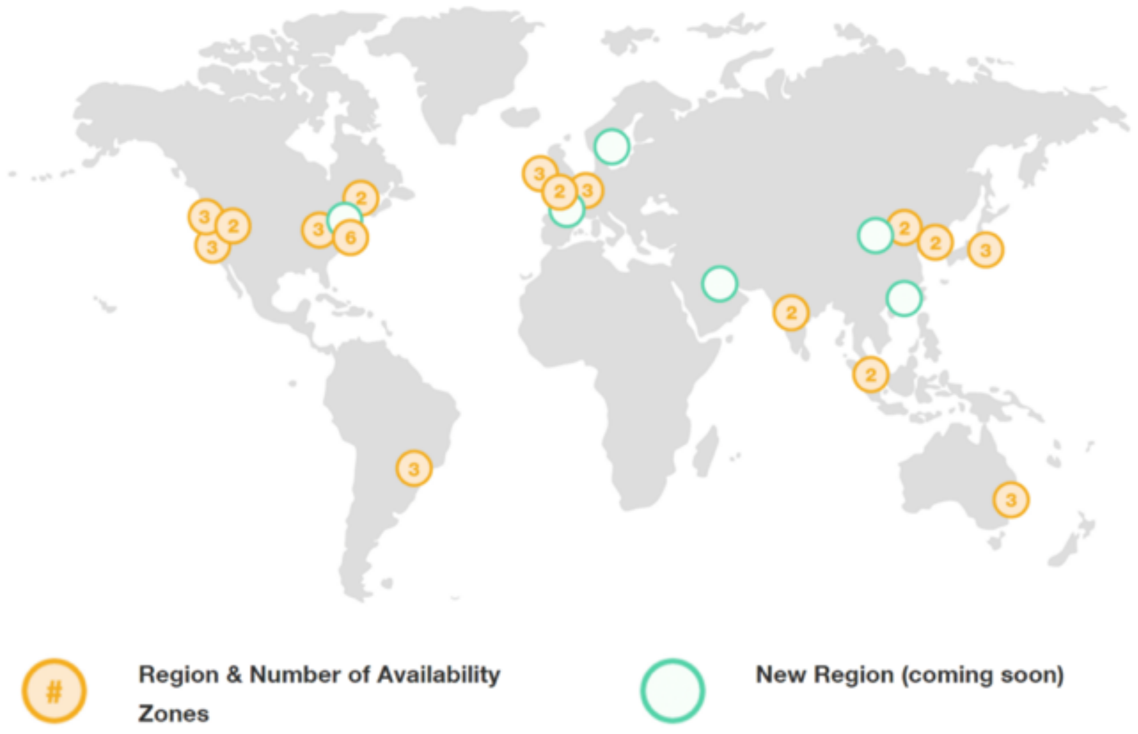

Starting at $44/month with affirm. Prequalify Now

Sample promo messaging

Redundant infrastructure

Although AWS is generally very reliable (knock on wood), we always prepare for all possible failures, including AWS outages. For all our production components, we have redundant infrastructure in multiple availability zones, which would keep us from going down in case of an issue affecting one data center (for instance, if lightning struck it or there was a power outage).

Although we host all our live infrastructure in one east coast AWS region, we also have redundant infrastructure in a west coast region that we are prepared to failover to quickly in case of a region-wide outage. For some components, like EC2, setting up redundant infrastructure was easy, but for others it required more work and planning.



AWS region locations

Servers

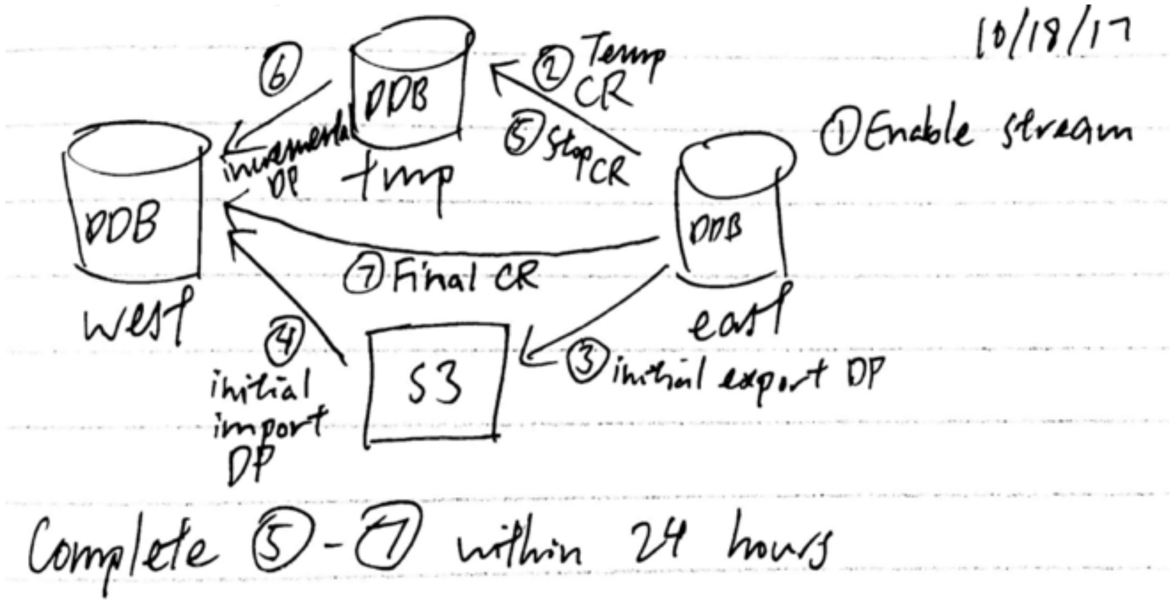
To scale up our EC2 instances in the west coast region, we added capacity to our autoscaling groups. Since we don't store any data on our machines, the only complication with adding west coast machines was updating configs to make sure that the west coast instances would talk to other servers and databases in the west coast (to avoid slow and costly cross region requests).

Databases

Multi-region database infrastructure is more complicated than application server infrastructure since it requires replicating data across regions. We use four database types in production: Aurora MySQL, DynamoDB, AWS ElastiCache for Redis (for caching high-volume MySQL query results), and AWS ElastiCache for Memcached (used as a celery result store).

Aurora MySQL databases are straightforward to setup since Aurora supports cross-region read replicas. So, we just had to add cross-region read replicas for all our RDS instances and double check that the parameter groups, which determine database configs, were the same in both regions.

DynamoDB multi-region infrastructure was more complicated to set up. WithGlobal Tables, AWS supports multi-region multi-master infrastructure for DynamoDB—but to set this up as a backup, we first had to copy all our east coast Dynamo data into Global Tables. To do this, we used AWS EMR and DataPipeline to import east coast table data into S3 and then export the data in S3 to the Global Tables following these instructions. With this process, we could do a one-time, bulk upload of Dynamo data up to some point in time. For the data added to the table after the bulk export started, we used a DynamoDB Cross-Region Replication library to apply real-time updates to the Global Table by sending the data to a Kinesis stream which applied updates to the table after we finished the bulk upload. For our larger tables, we had to alter this process slightly since a Kinesis stream can only hold up to 24 hours of data, and the data import/export process took over 24 hours. So, we instead streamed real-time updates into a blank table, and after the bulk upload finished, copied that table into the real table and changed the stream output to send data to the real table.



DynamoDB data replication process

For Redis and Memcached, we added duplicate infrastructure in east coast and west coast, but did not replicate data. For the caching use case, we would just hit the database directly for all requests, and although this would put more load on the database, we determined that the databases could still handle the load. For the celery result store use case, the celery results are only accessed very soon after they are written, so a small amount of data loss would not cause much service disruption.

S3

S3 supports cross-region replication, so we set up replication for all our S3 buckets that are critical to our operations. We have a branch with the changes required to switch over the west coast buckets ready to deploy in case we had to fail over S3.

Monitoring/Alerting

We also duplicated our monitoring and alerting infrastructure, which I discussed in a previous blog post, into the west coast region. We added duplicate Riemann (metrics aggregation) infrastructure in the region, and updated the Riemann configuration in both regions to send metrics to Elasticsearch instances in both regions. If we failed over servers, we would send metrics to the west coast Riemann servers automatically, and we would have minimal disruption in our metrics pipeline. If Elasticsearch went down in either region, we could point Grafana (which we use for creating graphs based on time series data) and Cabot (our alerting software) to the other region's Elasticsearch cluster.

We also added duplicate servers, load balancers, and a read replica database for Cabot in west coast. We actually have active-active Cabot infrastructure—our west coast Cabot server runs celery tasks, including ones that run status checks and send out alerts. We also added duplicate Cabot servers in another availability zone, which run both celery and a gunicorn webserver.

Failover process

Most of our components in production are accessed based on a Route 53 DNS entry. This makes failing over components very simple: we set up a weighted routing policy for the corresponding east coast and west coast components, initially with the east coast component at 100% and the west coast component at 0%. If we wanted to failover, we would just set the east coast component to 0% and the west coast component to 100%.

For databases, to fail over we would promote another machine to be the master in the AWS console as well as updating the DNS entry.

Documentation and planning

Before Black Friday, we also updated our documentation for handling production issues. For all the multi-region components, this meant documenting the process for failing over to west coast, and any other components that would be affected if we failed over. For example, if we failed over our servers, we would also have to failover databases since cross region database calls are very slow. In general, we made our instructions as detailed and clear as possible (for example, we linked pull requests for anything requiring code changes) so that the oncall engineer would be able to execute the instructions quickly with minimal room for error.

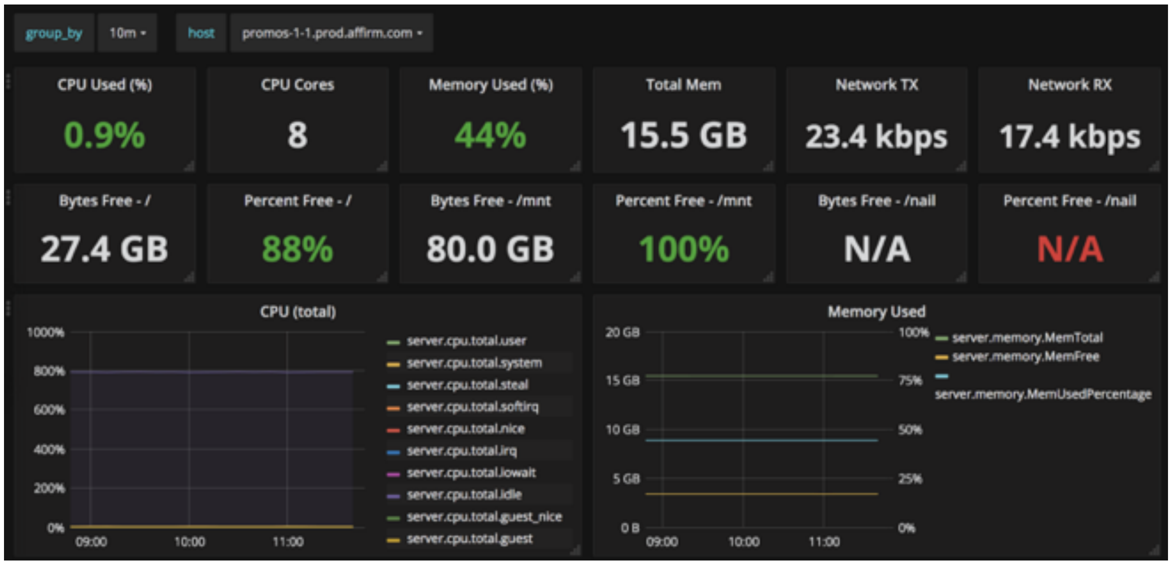
We also discussed the criticality of components so that we would know which components to fail over first. Our first priority is making sure users can checkout, so we would fix that first, then move on to other components. Our SRE team met every day in the week leading up to Black Friday to discuss anything outstanding we had to finish before the date and walk through out responses to failure scenarios.

In addition, we established and broadly communicated a chain of command in case of incidents. We wanted to establish an escalation path for anyone who noticed an issues, as well as avoid a "too many cooks" scenario when trying to resolve a problem. We also planned to follow our existing incident communication plan to update merchants, the teams who work with them, and our operations teams about any ongoing issues and resolutions.

The main event: Cyber Weekend

For Black Friday and Cyber Monday, we chunked our platform and infrastructure oncall schedules into 8 hour shifts so that our oncall engineers wouldn't get too fatigued. The oncall engineers, as well as many others, actively monitored our critical Grafana dashboards and Rollbar so that we would catch critical errors as soon as possible—although we have alerts that would call us for many critical issues, they may not go off until a couple minutes after an incident begins, which is a big deal on Cyber Weekend. Some of the dashboards we monitored included:

- HTTP status codes, latencies, and throughputs for critical endpoints
- Cloudfront and ELB 500 rate
- Celery task successes, errors, and latency
- System stats for servers and databases
- Checkout funnel metrics
- Third party request successes, errors, and latencies



group_by 10m host promos-1-1.prod.affirm.com

CPU Used (%)	CPU Cores	Memory Used (%)	Total Mem	Network TX	Network RX
0.9%	8	44%	15.5 GB	23.4 kbps	17.4 kbps

Bytes Free - /	Percent Free - /	Bytes Free - /mnt	Percent Free - /mnt	Bytes Free - /nail	Percent Free - /nail
27.4 GB	88%	80.0 GB	100%	N/A	N/A

DynamoDB data replication process

We caught one minor issue earlier on the morning of Black Friday. Some of our autoscaling servers had a config incorrectly setup which caused us to call AWS STS AssumeRole every time we hit our credentials store, and we were calling it frequently enough to get rate limited by AWS. Since we were actively monitoring, we noticed this issue after only one error, and were able to update the configs before we saw any major impact (we had fewer than 15 total errors, and the errors did not impact checkouts).

Preparing for 2019

After a successful Cyber Weekend this year, our team is already preparing for what we need to do for next Cyber Weekend. We're also planning ahead for significant traffic increases even earlier in the year as a result of exciting merchant launches ahead!

Some of our plans for 2019 include setting up multi-region active-active infrastructure so that we can handle region-specific outages more smoothly without manual intervention. We're also planning on adding policies so that our autoscaling groups actually scale with traffic instead of requiring a manual config change, and resharding our databases and reallocating our databases on RDS instances so that we can scale out our master databases even more.

Preparing for Black Friday scale in 2018 was a long term, cross functional effort. Thanks to everyone's hard work and preparation, we were able to handle Black Friday scale gracefully this year and are ready for even bigger challenges in 2019.

If handling these type of challenges sounds interesting to you (or if you love Segways), apply to join our team on our careers page!

My Internship at Affirm: Crafting a Reliable Metrics and Alerting Framework

Rohan Varma

This past spring, I had the opportunity to join the Risk Engineering team at Affirm as a software engineering intern. As a whole, the Risk Eng team is responsible for the decisioning systems that run Affirm's identity verification, fraud, and credit underwriting processes. Every time a borrower applies for a loan with Affirm, our systems are responsible for confirming their identity, determining if there is fraudulent intent, and deciding how much credit to extend to the user. To this extent, we build pipelines to transform raw data from different sources into useful signals, maintain business logic to make decisions, and support the development of machine learning models that are trained on historical data.

A reliable monitoring and metrics framework is an important component of a robust decision-making system. We want to know when our systems are experiencing above-average error rates or when there is a significant shift in the underlying data that we consume in order to respond appropriately. To this end, I was tasked with redesigning our monitoring systems that track and log our signal data over time, and alert us when an anomaly is detected. The overarching goal was to design and implement a reliable monitoring framework that was reliable, accurate, easy to maintain and build upon, and robust in the case of failures.

Metrics & Monitoring: An Overview

At Affirm, we rely on many different signals—defined here as schematized instances of data (e.g. a user's income to debt ratio)—to power our decisioning systems. For example, we train machine learning models on these signals to inform our decision to extend credit to applicants or not. The underlying assumption here is that the data a model is trained on is roughly similar to the data it does real-time predictions on. However, this assumption sometimes fails to hold true due to several factors like seasonal variation and underlying changes in the distribution of Affirm's customers. When this happens, it's important to have a metrics framework tracking our signal data to notify us about these changes in signal values and to allow us to respond appropriately.

Key Requirements of a Reliable Metrics Framework

There are several different things to consider when architecting a monitoring framework. We want to design for simplicity and ease of use, but without sacrificing configurability and reliability. Ideally, the system's services should also be consumable by product, business, and legal teams. With this in mind, we designed our solution with a few key requirements:

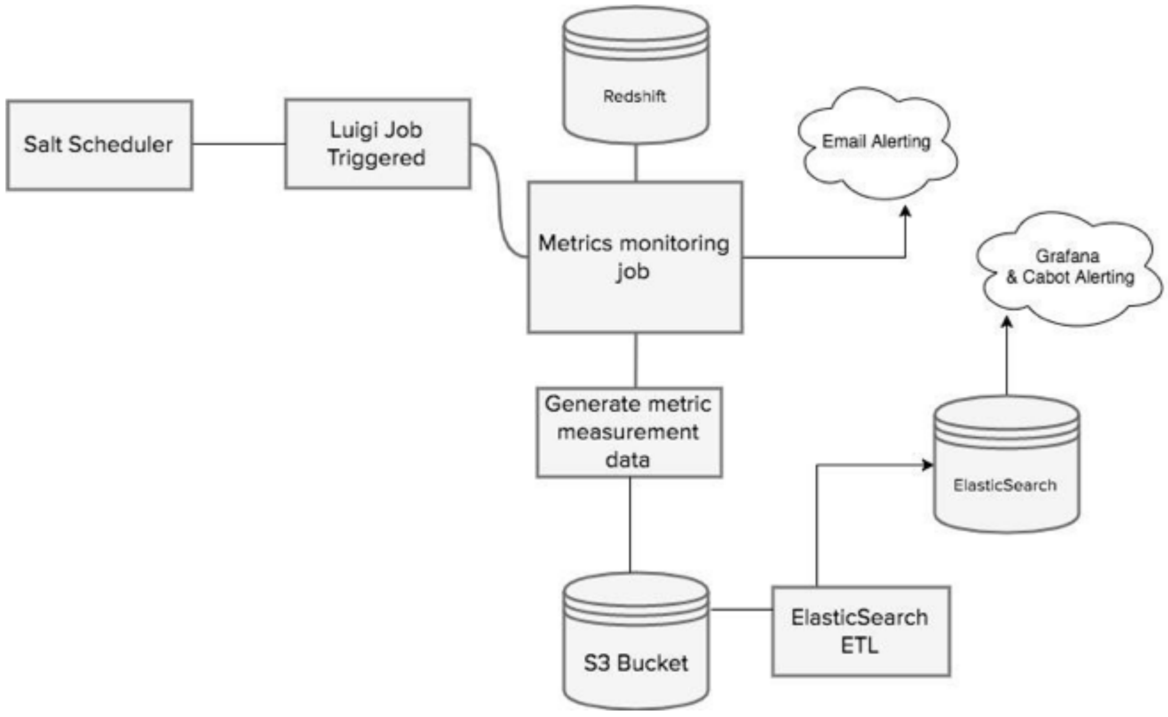
1. User Friendliness. Metrics should be easily interpretable by any audience. A low-friction setup should exist to quickly add and deploy additional metrics.

2. Reliability. We use metrics to ensure that our systems are behaving as expected. How can we ensure that the monitoring system itself is reliable?

3. Accuracy. Our monitoring system should quickly alert us at the correct urgency level in the case of regressions. We should also ensure that our system isn't too noisy, so that each alert is truly meaningful and addressed appropriately.

4. The Big Picture. Metrics are important to several teams including data science, engineering, product, and legal. Given this, how do we know what metrics are the most important to track and who to alert in case of anomalies?

Our solution



We decided to use Luigi, an open-source package for managing batch pipelines that supports complex job management and automatic retries, to encapsulate the different tasks which query our underlying data stores and wrangle data.

We set up configs for our signal monitoring tasks in files generated with saltstates. These configs include which table to read data from, where to upload data to, how to aggregate the results, and whether to send an email notifying a list of users after the job has completed.

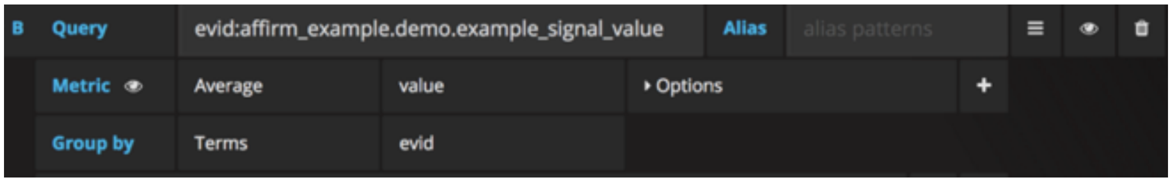
Our job then reads data from tables hosted in Redshift, Amazon's data warehousing solution that allows for complex querying against large amounts of data. The job then does some intermediate data-wrangling, including reading historical metrics data from S3, Amazon's key-value storage solution, and computing scores that quantify how much that day's signals have deviated from past history.

Next, these results are encapsulated into a schematized format and uploaded to S3. Later, in a separate task, the metrics are loaded into Elasticsearch—an open-source engine that allows for search and analysis of data.

Visualizing Metrics with Grafana and Cabot

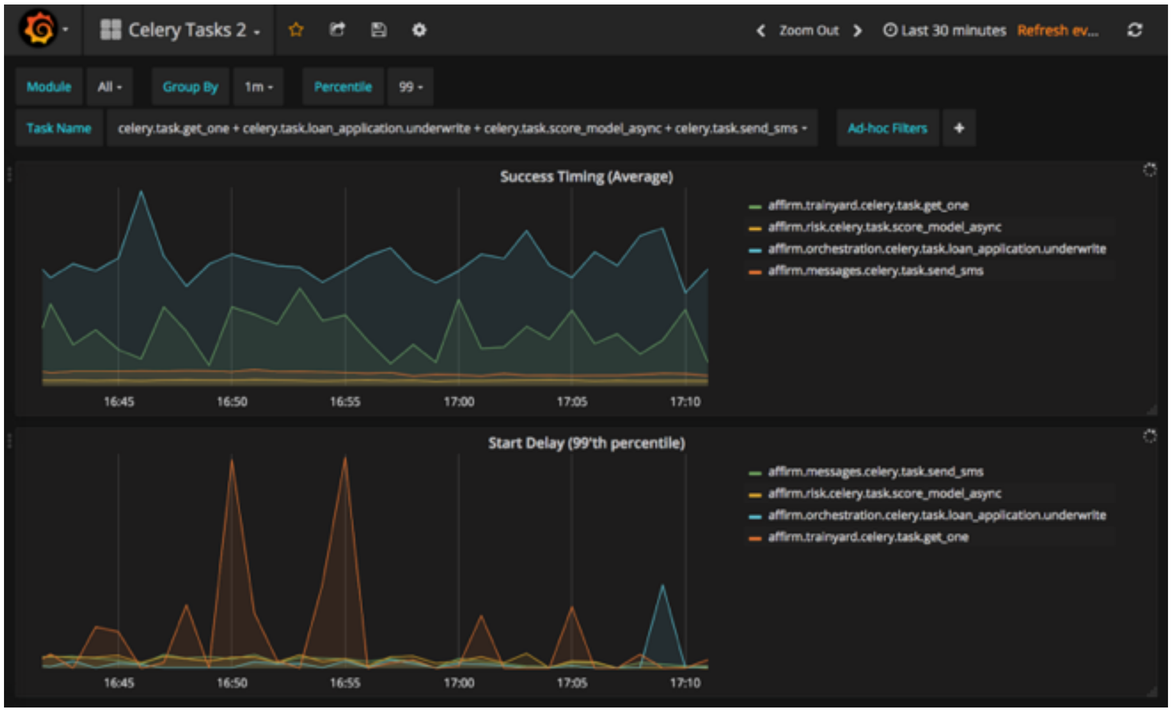
In addition to a reliable framework to measure and upload metrics for data on a regular basis, we also found it important to be able to easily visualize this data over time. Moreover, we want to automatically be alerted when unexpected spikes occur. To accomplish this, we leverage Grafana, a dashboarding service, and Cabot, an alerting system, respectively.

After we've set up a system to load our data into S3 and Elasticsearch, we can configure a Grafana dashboard to read data from an Elasticsearch index and render it in a dashboard. Below is an example configuration for a dashboard:



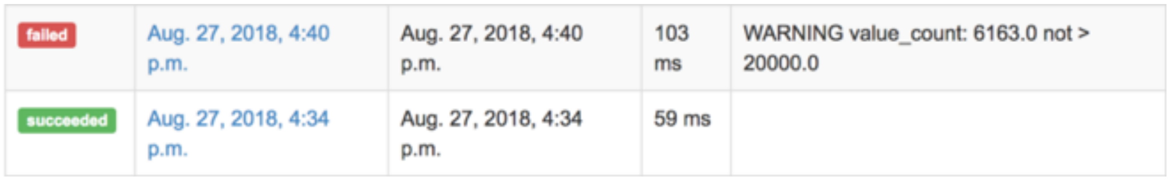
Query	evid:affirm_example.demo.example_signal_value	Alias	alias patterns
Metric	Average	value	Options
Group by	Terms	evid	

After configuring several different panels and ensuring that our jobs to upload data to the correct indexes are running reliably, we end up with a dashboard like the below (with some of our information redacted):


Celery Tasks 2
Zoom Out
Last 30 minutes
Refresh ev...
Ad-hoc Filters
Success Timing (Average)
affirm.trainyard.celery.task.get_one
affirm.risk.celery.task.score_model_async
affirm.orchestration.celery.task.loan_application.underwrite
affirm.messages.celery.task.send_sms
16:45
16:50
16:55
17:00
17:05
17:10
Start Delay (99'th percentile)
affirm.messages.celery.task.send_sms
affirm.risk.celery.task.score_model_async
affirm.orchestration.celery.task.loan_application.underwrite
affirm.trainyard.celery.task.get_one

Module: All
Group By: 1m
Percentile: 99
Task Name: celery.task.get_one + celery.task.loan_application.underwrite + celery.task.score_model_async + celery.task.send_sms

The final piece of our monitoring system is to configure Cabot to alert us based on our Grafana dashboards. Cabot allows you to configure alerts based on threshold values, and has support for different levels of urgency and notifications to users or on call schedules via Hipchat, email, text, and phone call. An example alert is shown below:



failed	Aug. 27, 2018, 4:40 p.m.	Aug. 27, 2018, 4:40 p.m.	103 ms	WARNING value_count: 6163.0 not > 20000.0
succeeded	Aug. 27, 2018, 4:34 p.m.	Aug. 27, 2018, 4:34 p.m.	59 ms	

Putting it all Together

In the end, we ended up with a reliable monitoring and alerting system that automatically retries in the case of failures, aggregates our data according to configurable specifications, and automatically emits schematized metrics that we can visualize and be alerted on.

There's still a lot more work to be done, however. Future improvements include designing better algorithms for anomaly detection and increasing automation around the process of adding queries for additional metrics and setting up dashboards in Grafana. Nonetheless, since we launched this solution we've been able to monitor fluctuations in key data with much more reliability and consistency, allowing our teams to be alerted when something is off with our signal data. Overall, this was a great project to work on, since it allowed me to think through how a system should be designed and architected while exposing me to several different technologies and providing value to the rest of Affirm.

How Affirm Is Different: Understanding the Fundamentals of Credit

Sandeep Bhandari

Despite the ubiquity of credit, few consumers know enough about its complex terminology and cost structure to make fully informed decisions. And most credit providers take advantage, charging a constellation of fees in order to maximize profits. The result is that in 2016 alone, American consumers paid more than $90 billion in fees.

At Affirm, we believe consumers should be able to benefit from a more honest and transparent form of credit. We quote customers a fixed price for their credit up front, so that when they sign up for a loan, they know exactly what they will end up paying. No matter what happens after they confirm their loan, they'll never owe us a penny more than the amount they initially chose.

This FAQ is designed to clarify some of the subtleties around credit (including interest), how it affects consumers, and how Affirm is different.

What is APR?

Let's start with the basics. When applying for and using a credit card, consumers typically pay attention to one number: Annual Percentage Rate or APR. The Truth in Lending act requires lenders, by law, to express interest on an annual basis—hence, APR. The purpose of expressing an APR is to offer customers an easy way to compare credit products.

But It's Not That Straightforward

Lenders must express interest on an annual basis, but that's not necessarily how interest is calculated and charged. Interest on loans via Affirm, like most credit cards, actually accrues on a daily basis.

Some experts believe APR can be misleading in relation to short-term credit products, like credit cards. For one, most credit cards offer variable APR rates, meaning they will fluctuate according to the market, an index, or the U.S. prime rate. Second, the daily percentage rate could change depending on a number of factors including the current principal balance and potential fees, like late fees or a yearly fee. And most credit card fees—on average, six, but as high as twelve per card—are never part of the APR calculation.

These are just a few of the reasons why calculating the actual cost of a purchase made on a credit card can be difficult for the everyday consumer. And most credit card issuers don't offer any of this information to consumers—they must figure it out themselves.

So How is Affirm Different?

While Affirm does express interest rates in terms of APR—as required by law—we differ in four major ways:

1. Simplicity: Our user experience is mobile optimized and only asks users for five pieces of information to make a credit decision.

2. Transparency: Affirm shows consumers up front—before even accepting the loan—what the total purchase cost will be, including the total interest amount.

3. Predictability: Affirm doesn't charge any late, penalty, or annual fees which could add to the principal balance, and therefore complicate interest costs.

4. Control: Unlike credit cards, Affirm's app and point-of-sale loans are not a revolving line of credit. Instead, we approve customers only for the amount they're looking to purchase—on their terms. They can select to pay over 3, 6, or 12 months. And there's no penalty for paying it off early.

Credit cards, on the other hand, can be confusing and complex when determining interest and total cost on a single purchase. Especially if it charges compounding interest.

Wait, What's Compounding Interest?

Most credit cards charge compounding interest, or interest on interest. More specifically, compounding interest is interest charged not only on the principal balance but also on interest or some fees that have already accumulated.

Affirm Doesn't Do That?

No. Interest on loans through Affirm are only charged interest on the purchase amount—or, principal balance. It's why we can be transparent about the total cost at the time of credit approval, even before the user accepts it. And because we never charge any late or penalty fees, that amount will never change. Ever.

It's Also Important to Understand The Effective Interest

As stated above, expressing an interest rate for a credit product via an annual rate (APR) does not tell the whole story.

That's why it's important to understand what the effective interest rate is on a purchase. Basically, it is the interest rate that is actually paid over a given time period on a loan or credit product.

For example, if someone chooses to pay for a purchase with Affirm over a six-month term at 20% APR, the effective interest they will pay on that purchase will not exceed 5.91%.

Here's how it works. Someone applies, and is approved, for a $600 purchase with Affirm. The total payment amount for each month is $105.91. The total interest on the $600 is $35.48, or 5.91%.

What About Affirm's 0% APR Promotion? Credit Cards Offer That All the Time.

Actually, most 0% APR promotions offered by other credit cards—typically store-branded cards—are deferred interest promotions.

Most of these offers are 0% APR for 12 months, meaning that no interest will be applied to any purchases made within that time frame. The catch, however, is that if any balance remains after those 12 months—even just $1—or no payment has been made in 60 days, interest will not only be charged on the remaining balance but also on the full original balance. This doesn't include any penalty or late fees that could also be applied.

Affirm's 0% APR promotions, on the other hand, will never include deferred interest or penalty fees of any kind. Users can just select their term length—3, 6, or 12 months—and are shown the full amount owed for each month and in total. The selected amount will never change, no matter what.

Affirm loans are made by Cross River Bank, a New Jersey State Chartered Commercial Bank, Member FDIC.

End-to-End Testing with WebdriverIO

Mike Phillips

At the start of 2018, the front-end infrastructure team at Affirm began to replace our existing end-to-end (E2E) testing tools with WebdriverIO (WDIO). In this blog post, we'll share some details around how WebdriverIO has helped us improve the front-end developer experience and the overall testing culture at Affirm.

The Importance of E2E Testing at Affirm

Affirm's team, codebase, and business have undergone rapid change within the past few years. In 2016, Affirm's front-end team consisted of about two or three front-end developers working on two web apps and a public JS library (affirm.js). As of July 2018, there are approximately thirty engineers making front-end contributions from nine distinct teams, working on fourteen different web apps with twenty-one internal libraries.

Affirm, as a whole, has undergone similar growth. This time last year, we had one office that housed around 220 employees. Today, we have almost 400 employees split between our SF headquarters and NY office, and are soon opening a third space here in San Francisco.

With the rapid growth of Affirm's engineering team, the increasing complexity of our codebase, and the scale of Affirm's business, E2E testing is more critical than ever. It's simply not feasible to manually QA each app that is affected by any given update. These tests help prevent regressions in critical functionality and give engineers confidence as they deploy their work throughout the day.

Motivation to Switch

Our previous E2E testing tools used a combination of Python, behave, selenium, and tox to get the job done. Unfortunately, after several years of serving their purpose these tools became a common source of frustration.

Context switching between Python, JS, and a lot of custom boilerplate code made it difficult for engineers to write, maintain, and debug these tests. A few recent post-mortems had also highlighted the need for cross-browser and visual regression testing capabilities.

The shortcomings of our existing tools were actively preventing or discouraging engineers from writing important E2E tests. We needed to find a tool that could handle all of our testing needs while also providing an easy and rewarding developer experience.

Our Goals

Switching tools is often a major undertaking for any codebase. We wanted to ensure that whatever tool we chose for end-to-end testing would meet our immediate needs and also provide a solid foundation for the future. Specifically, we wanted to find a tool with the following qualities:

Stable

Few things are more frustrating than flaky tests. An end-to-end testing tool that meets all of our needs but is buggy and unreliable is a non-starter.
In addition to being stable itself, we needed a testing tool that made it easy for developers to write stable tests. For E2E tests, this means handling things like waiting for page loads or interacting with elements outside the viewport.

Capable

We wanted an E2E testing tool that would:
• enable developers to write tests that can do everything their users can
• integrate with a cloud testing service like Saucelabs or Browserstack
• support testing capabilities like cross-browser and visual regression testing
• provide flexible configuration and customization

Easy

Finally, it was really important to find a tool that would be easy to use. Writing tests is an important part of the work that goes into front-end engineering, but it shouldn't be a difficult part. We wanted to find an end-to-end testing tool that would be easy to learn, easy to use, and easy to debug.
After some investigation and experimentation with different tools, we eventually settled on WebdriverIO. The remainder of this post will cover some of our favorite things about WebdriverIO and how it helps us accomplish these aforementioned goals.

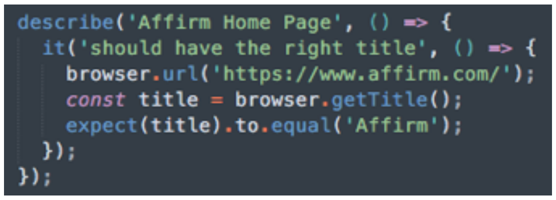
WebdriverIO Highlights

Documentation

First off, WebdriverIO has excellent documentation. The Developer Guide makes it easy to learn how to set things up, and the API docs explain how to interact with the browser and write your first test.

Synchronous Mode

One of our favorite features, "synchronous mode," was introduced to WebdriverIO in v4. Synchronous mode enables you to write your asynchronous test steps as if they were simple, synchronous function calls.



```
describe('Affirm Home Page', () => {
  it('should have the right title', () => {
    browser.url('https://www.affirm.com/');
    const title = browser.getTitle();
    expect(title).to.equal('Affirm');
  });
});
```

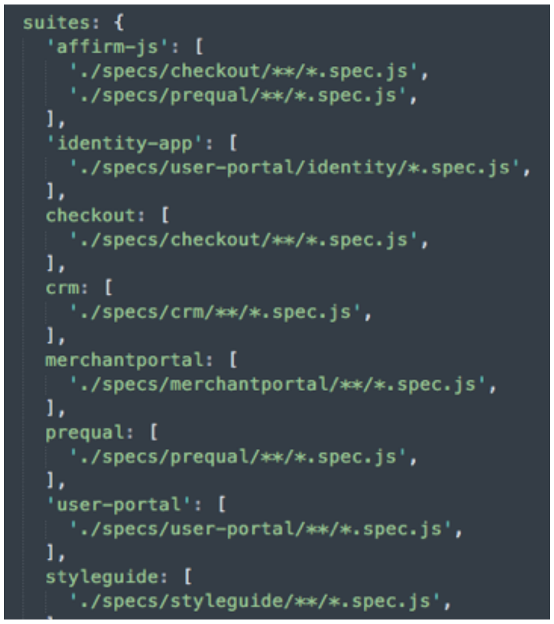
One of our favorite features, "synchronous mode," was introduced to WebdriverIO in v4. Synchronous mode enables you to write your asynchronous test steps as if they were simple, synchronous function calls.

Helpful Browser Commands

WebdriverIO's API provides over 200 different browser commands to help you get the job done.
Many of WDIO's browser commands are just implementations of the Webdriver protocol spec. But, WDIO then uses these commands to build more intuitive, easy-to-use commands.
For example, the browser.setValue() command can be used to set the value of an input. Under the hood, this command uses the browser.elements(), browser.elementIdClear(), and browser.elementIdValue() commands as building blocks.
In addition to implementing and extending the Webdriver protocol, WDIO also provides utility commands that are directly integrated with WDIO and make it easy to work with.
Some good examples include the browser.debug() command, which makes it easy to pause your tests and debug your application, as well as "wait" utilities like browser.waitForVisible() that help developers write stable tests.
Finally, WDIO also provides commands that use Appium to enable running tests on mobile devices.

Flexible configuration

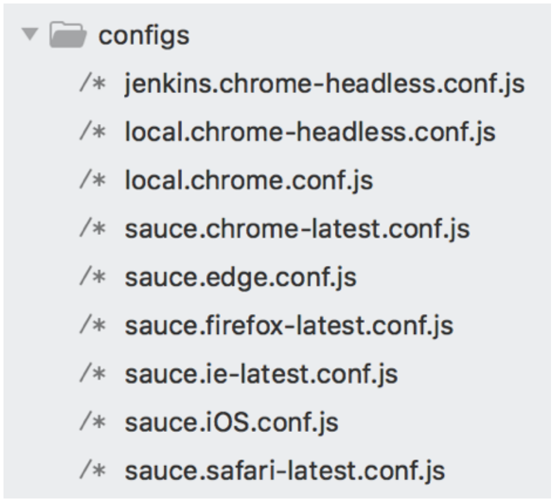
WebdriverIO provides flexible configuration and tons of customization opportunities.
The heart of our WDIO project is the base configuration file. The configuration file gives you control over a wide array of options like log levels, default timeout durations, and what frameworks you use to run your specs (mocha, jasmine, or cucumber).
One particularly helpful configuration option is the ability to organize your tests into custom suites. Since we keep all of our end-to-end tests in the same directory, we choose to organize them by application. This makes it easy for us to set up efficient test automation on Jenkins that only runs the subset of our full test suite relevant to the application being deployed.



```
suites: {
  'affirm-js': [
    './specs/checkout/**/*.spec.js',
    './specs/prequal/**/*.spec.js',
  ],
  'identity-app': [
    './specs/user-portal/identity/*.spec.js',
  ],
  checkout: [
    './specs/checkout/**/*.spec.js',
  ],
  crm: [
    './specs/crm/**/*.spec.js',
  ],
  merchantportal: [
    './specs/merchantportal/**/*.spec.js',
  ],
  prequal: [
    './specs/prequal/**/*.spec.js',
  ],
  'user-portal': [
    './specs/user-portal/**/*.spec.js',
  ],
  styleguide: [
    './specs/styleguide/**/*.spec.js',
```

WDIO makes it easy to extend this base config so you have the flexibility to do things like per-app or per-environment customization. We drew inspiration for our config organization from Kevin Lamping's blog post on the topic.

Our current list of configs is below. If we want to extend this list to other browsers or environments in the future, it will be easy to do so.



```
▼ configs
    /* jenkins.chrome-headless.conf.js
    /* local.chrome-headless.conf.js
    /* local.chrome.conf.js
    /* sauce.chrome-latest.conf.js
    /* sauce.edge.conf.js
    /* sauce.firefox-latest.conf.js
    /* sauce.ie-latest.conf.js
    /* sauce.iOS.conf.js
    /* sauce.safari-latest.conf.js
```

We use a Makefile and environment variables to make running a specific suite or config as simple as:

```
make wdio DOMAIN=example.com SUITES=user-portal CONFIG=sauce.ie-latest.conf.js
```

Powerful Add-on Services

Services are optional libraries that you can use to extend WebdriverIO's core functionality to provide easy integrations and additional capabilities.

The Chromedriver service uses the wdio-chromedriver-service library to make it painless to set up and run WDIO tests on Chrome. With our setup, a new developer at Affirm can start running end-to-end tests in Chrome on their local machine in a few simple steps:

```
git clone <AffirmFrontendRepo>
cd web-ux/e2e-tests/
make install
make wdio DOMAIN=example.com CONFIG=local.chrome.conf.js
```

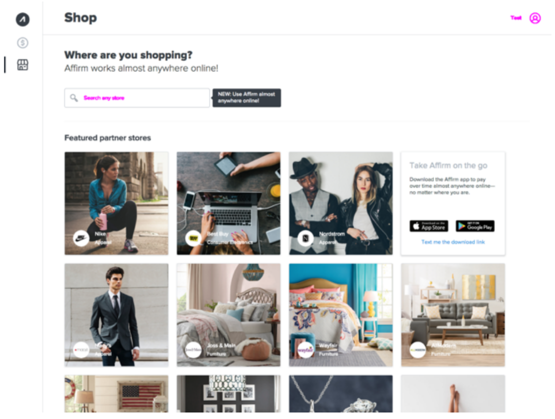
The Sauce service uses the wdio-sauce-service library to provide a simple integration with Saucelabs. Paired with our custom configs mentioned above, it is incredibly simple to run any browser and device configuration on Saucelabs.

```
make wdio DOMAIN=example.com CONFIG=sauce.chrome-latest.conf.js
make wdio DOMAIN=example.com CONFIG=sauce.firefox-latest.conf.js
make wdio DOMAIN=example.com CONFIG=sauce.safari-latest.conf.js
make wdio DOMAIN=example.com CONFIG=sauce.ie-latest.conf.js
```

The Visual Regression service uses the wdio-visual-regression-service library to provide custom browser commands that make it easy to add visual regression tests to any of your WDIO tests.

The visual regression service provides three new browser commands, as well as a system for managing screenshots and generating pixel diffs. With these building blocks, it's possible to build a flexible visual regression testing workflow customized to the needs of your team.

```
browser.checkElement(elementSelector, [{options}]);
browser.checkDocument([{options}]);
browser.checkViewport([{options}]);
```



Where are you shopping?
Affirm works almost anywhere online!

Download the Affirm app to pay over time almost anywhere online—no matter where you are.

Affirm currently uses all of these services with WebdriverIO, but they are only a small subset of its larger list of available services.

Future Work

We're still in the early days of WebdriverIO at Affirm, but it's been a pleasure to use so far. Ever since our initial experience, it's proven to be a stable, capable, and easy-to-use tool for our team.

This post highlights one of the many major front-end initiatives happening at Affirm. If you'd like to learn more and get involved, check out our Careers pageand get in touch!