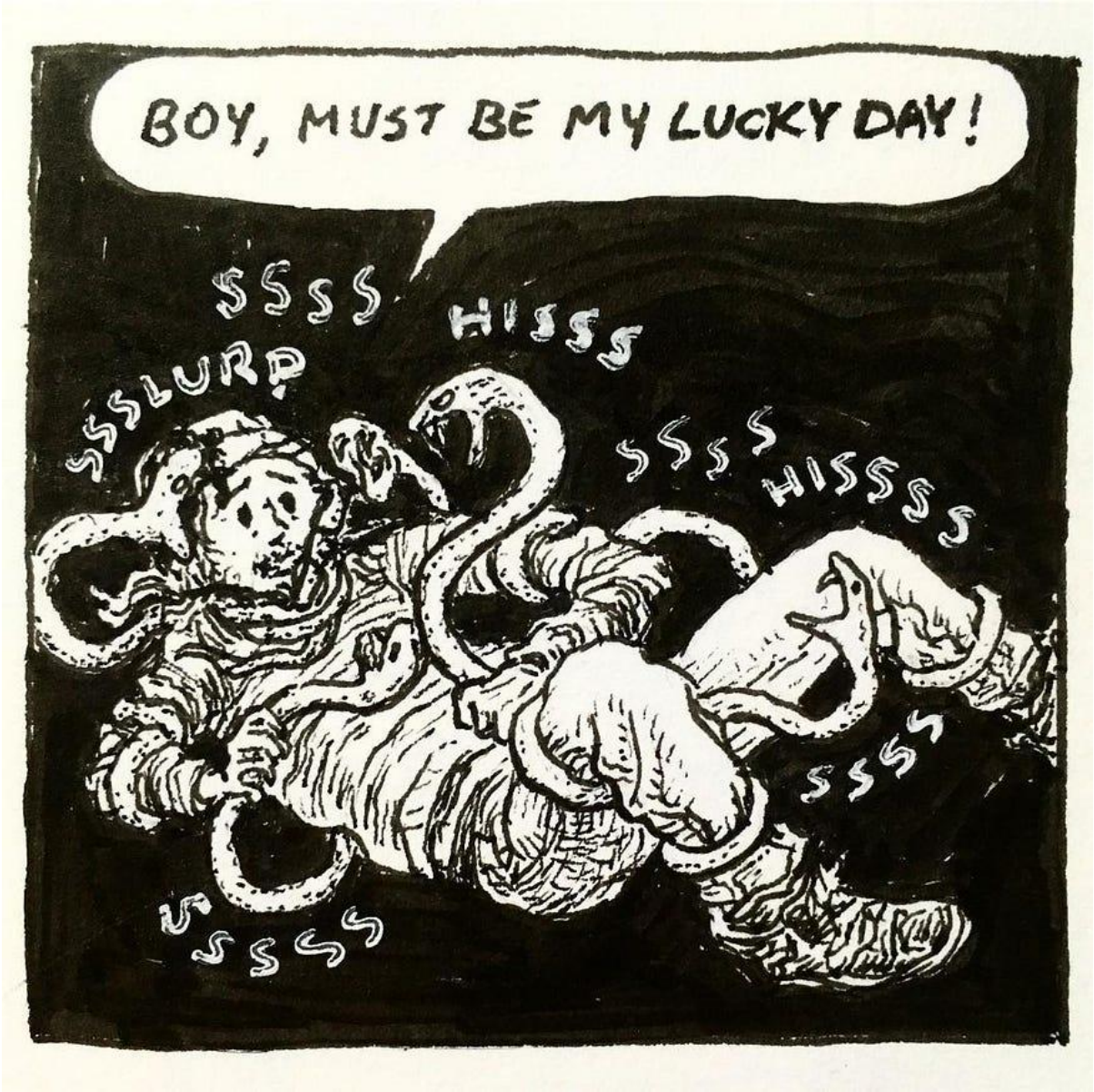
Synthetic Load Testing @ Klaviyo

Author: Chris Conlon

Claps: 325

Date: Mar 2, 2019

Every September at Klaviyo we start to prove our platform is ready for a surge of traffic during the Black Friday and Cyber Monday shopping holidays, consisting mainly of retailers sending out promotions and consumers making purchases. This surge of traffic is both an egress of over a billion email messages and an ingress of billions of ecommerce actions that Klaviyo tracks (like customers making purchases, viewing products, etc.). The combination sums up to a roughly 5â€“7x sustained load increase across the platform. To make sure that the load increase isnâ€™t noticeable to anyone using the platform, we make an annual investment to run several load tests in the lead up to the holiday.



Credit: @tenth_door on Instagram

This sounds like a reasonable thing to do, I know, but **every year** I have to convince myself that running these load tests is worth the engineering time and infrastructure dollars. To be specific, Iâ€™m talking about synthetic load tests, which are run completely separate from any production infrastructure and end up being largely contrived (more on that later). Not only is it financially expensive to run synthetic load tests, it is also emotionally expensive for engineers that run these contrived tests sitting beside their teammates as they work through punch lists of tangible, critical platform issues to fix before the holiday.

Load Testing in Production Applications

I should start out by mentioning that we do run several load tests against our production infrastructure each year, which pay dividends immediately. â€œScale Outâ€ tests in production environments stand out as a solid example. This is where the platformâ€™s tunable resources (think of server counts and processing power) get turned up to the forecasted holiday scale and â€œburnt inâ€ over a few days. These progressive scale out tests were key in highlighting the leaky seams of the platform while running at scale. As an example, some of the early scale out tests Klaviyo ran this year revealed that our MySQL â€œmax_connectionsâ€ setting value was set too low on some database servers. As our processing servers came online during the scale out and opened new connections, the â€œmax_connectionsâ€ setting was crossed, after which MySQL began refusing new connections. This was an immediately actionable fix that would have caused an interruption during the holiday if not addressed. We were able to proactively expand our usage of ProxySQL in order to multiplex and cut down these stacked connections.

Unfortunately though, when youâ€™ve forecasted 5â€“7x volume for an upcoming holiday but have only tested with your applicationâ€™s organic volume, it is hard to move into that holiday with confidence only running load tests in production. It is possible to simulate increased volume in production applications, for example sending contrived email messages through the platform (as an example specific to Klaviyo), but stopping just short of actually delivering them. This leaves any actual delivery code untested though, and potentially code downstream of that action as well. Two years ago we did exactly this, and it turned out that some of our delivery code made an inefficient query that put CPU utilization on a core MySQL database over our alerting thresholds, resulting in a noticeable mail send slowdown during the holiday. This was something weâ€™d only be able to spot at 5â€“7x scale.

It also should be said that even if there are reasonable ways to artificially increase throughput in a production application, doing so runs the risk of creating a service interruption for your applicationâ€™s users. It may even be inviting one. This is where we have to turn to synthetic load testing to safely stress our applications with volume impossible or unsafe to achieve organically.

Synthetic Load Testing Applications

It sounds deceivingly straightforward to spin up a clone of an application in an isolated environment, but as applications (and the teams that maintain them) grow, they can become a sum of several independent parts, and those parts donâ€™t always follow the same pattern. At Klaviyo we have roughly 150 clusters of EC2 instances used for processing ecommerce data, serving user and API requests, and ephemeral persistence servers, as well as 40 clusters of MySQL databases. It is a tradeoff, where divergent patterns facilitate rapid building and creative solutions, but trade that for maintenance and upkeep debt.

At Klaviyo we use Terraform to manage our cloud infrastructure, but over the course of a couple years as we have migrated to using Terraform, a few different internal implementation patterns

have emerged. We also have a collection of server initialization automation that suffers from the same divergent pattern problem. Load testing is a place where the interest on that tech debt gets felt and ideally paid down some, specifically in coalescing all of the application components into a single orchestrating tool, like Terraform.

2017

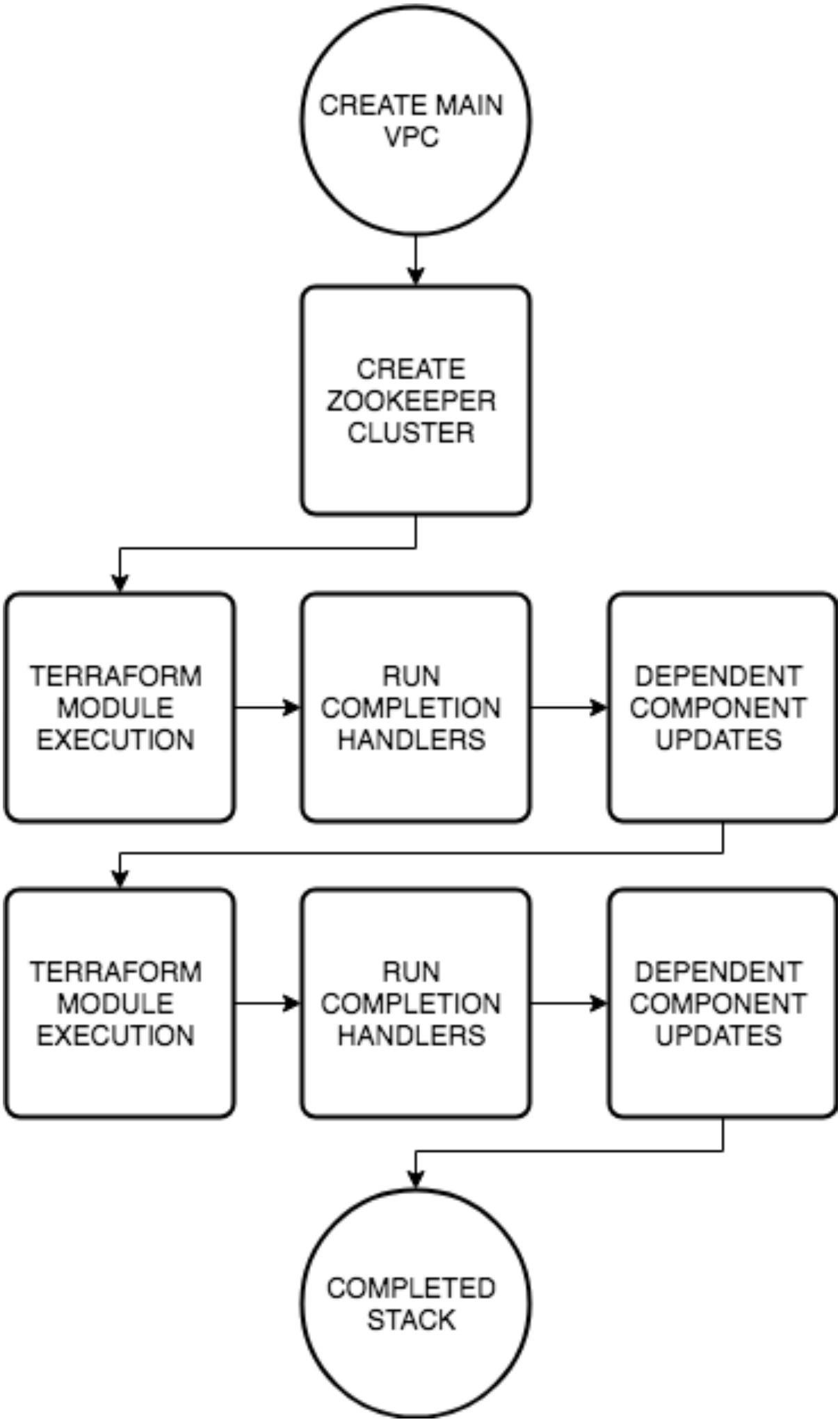
Ahead of the 2017 holiday, we used a single monolith Terraform module that pulled in all of our other disparate Terraform modules that were specific application components and then created a large dependency graph to enforce the order that the infrastructure came online. This was a great first step, but had several drawbacks:

- It took a nontrivial amount of time to spin up (several hours).
- It was difficult to make changes after the initial Terraform apply had run.
- It was difficult to run post-Terraform configuration and seed test data.
- It was difficult to keep up to date as production infrastructure changed

The iteration cycle time for fixing mistakes in the monolith module or testing new permutations became untenable as the monolith reached parity with production. It got us through our testing and built up our confidence ahead of the holiday, but it wasnâ€™t a sustainable pattern to continue testing with.

2018

Ahead of 2018, we continued to iterate on the Terraform automation by writing a â€œKlaviyoStackâ€ framework in python. A KlaviyoStack is a container for components of infrastructure configuration, that most commonly contain executions of our Terraform modules and completion handlers. Think of the stack pattern like a glue between components of Klaviyo infrastructure. Stacks are composed of a series of ordered steps. Each step defines a unit of idempotent work, like spinning up an AWS autoscaling group and waiting for its EC2 instances to pass ELB health checks and then immediately running completion handler code, which has access to the attributes of the infrastructure the step just successfully created (like a load balancer endpoint, or a hostname for an RDS cluster).


CREATE MAIN VPC
CREATE ZOOKEEPER CLUSTER
TERRAFORM MODULE EXECUTION
RUN COMPLETION HANDLERS
DEPENDENT COMPONENT UPDATES
TERRAFORM MODULE EXECUTION
RUN COMPLETION HANDLERS
DEPENDENT COMPONENT UPDATES
COMPLETED STACK

An early diagram used in an internal RFC to convey how we were thinking about the structure and flow of a KlaviyoStack.

KlaviyoStacks and their steps are defined by a remote blob of JSON stored in S3. As steps are run, their remote state gets updated to reflect the progression of standing up the stack. The Terraform modules also have their own remote state stored in S3, and for each step the statefile is isolated by using Terraform workspaces which avoids one giant Terraform statefile. By keeping the step state remote, it made it easy to share stacks across workstations, which was difficult in the previous yearâ€™s iteration. This also had an added value of keeping secrets off of local machines and in a shared secrets manager, a good pattern to follow even in a throwaway testing environment.

Organizing our infrastructure automation with this pattern afforded us some improvements over the first iteration:

- Each step is designed to be idempotent and isolated from other components, so altering and iterating on them is simple and doesnâ€™t risk breaking other components.
- Completion handlers allow configuration outside of Terraform and make it easy and fast to iterate by making necessary updates as the component changes.
- Spin up time can be reduced by choosing to omit unneeded steps depending on the tested case.
- Since steps are arbitrary containers, they could even define sets of tests to run against the newly created stack and report on results, as opposed to doing this manually.

A sample **step** is displayed below, which could be plugged into a **stack**. This step runs a Terraform module named `mail_dumpster` which creates a load balancer, Auto Scaling group, and some DNS records to facilitate accepting mail over SMTP and just discarding it. After the Terraform module runs successfully, the `on_complete` handler is run, which in this case inspects the Terraform outputs for its module and updates a secret store (ZooKeeper) with the hostname of the load balancer that was just created.

This step would run a Terraform module that looks like the gist below.

The `mail_dumpster` module configures an output for `mail_dumpster_domain` which is used in the completion handler to update a separate secret store. If something about this underlying module changed, we could rerun the step, which would rerun the Terraform, which would update any other dependent infrastructure.

The following are a few valuable takeaways we have gleaned from load testing year over year.

The Value of Iteration Speed

Iterating quickly is crucial both when wrangling production infrastructure into â€œsyntheticâ€ testing infrastructure as well as when a test yields an unexpected result where you need to run different permutations to understand and remedy an issue. When this process is slow or manual, we can lose focus on the test we set out to perform in the first place.

Our stack pattern allowed us to create â€œminimalâ€ versions of each component step that could be substituted with the production grade steps. These were contrived versions of each production component that were easier and quicker to configure, for example spinning up a single tiny Redis instance rather than waiting for a production-like 20 node cluster to come online. The minimal steps were on average twice as fast to iterate with, and in cases where the production

version of a step had significantly more nodes than the minimal version (for example a large Cassandra cluster), it was much more than twice as fast. It is also worth noting that iterating with the minimal steps saves considerable infrastructure cash.

Using this pattern we were able to move quickly, building out minimal steps and then later extrapolating them out to production grade steps. The next iteration of this pattern would incorporate parallelism between step executions, which would unlock even faster spin up times.

The Value of Idempotency

To support iterating quickly, it is also crucial to be able to change infrastructure configurations for independent components and also disseminate the side effects of those changes to other components. Idempotency is a tenant of writing Terraform code, but Iâ€™m speaking here more to how a part of the greater application that the Terraform code would create (and the completion handler code a step would run) could be rerun without risk. An example would be if we make a change to a load balancer and had to distribute the updated connection details to other dependent components that would need to connect to it. Our stack pattern as a rule made each step idempotent so that it could be rerun at any time. The completion handler was responsible for ensuring that any settings stores were updated and dependent components notified. We found that Terraform provisioners and local/remote execution couldnâ€™t get us far enough in gluing our stack steps together.

This was a major improvement over the previous iteration which would require manual interaction to disseminate component changes, which was error prone and time consuming.

Currently, the only dependencies that steps could place on each other are their order, because some needed to run before others. This was a very loose association though, it was just an ordered list, so the next iteration would build out a smarter dependency graph between component steps, which would support running steps in parallel.

The Value of Seeding Realistic Test Data

After successfully creating a stack, it is still â€œemptyâ€. Running tests against an empty stack makes it too easy to hit throughput goals, so seeding realistic test data is crucial to giving merit to test results.

In the previous year, each stack was responsible for migrating its own database schemas and creating its own fixture data. This could take hours and wasnâ€™t a straightforward process, for example when creating several thousand sharded tables, across several database servers. On this iteration we decided to have two options, restoring our databases from snapshots with empty already migrated schemas and restoring from migrated and pre-seeded schemas. This took nearly all of the complexity out of creating schemas and seeding data, while yielding consistent fixture data and consistent seeding times out of the box.

Unfortunately due to how the storage engine works with AWS Aurora, it takes 30â€“45 minutes to use either of these restore processes. For other persistence layers like Redis, it would likely be much quicker to have a similar process but instead restoring from an RDB snapshot.

Shortcomings

While this stack framework served us well in preparation for the 2018 holiday, it has some shortcomings worth talking about.

Most notably that there is no process in place to ensure that the â€œKlaviyoStackâ€ continues to be operational as infrastructure code changes and drifts. This was a major issue listed out at the beginning of this write up that we never got to addressing. For this iteration to be sustainable, there would need to be a way to enforce that changes elsewhere in the infrastructure configuration automatically propagate to the â€œKlaviyoStackâ€.

This framework also doesnâ€™t address the divergent patterns for how we manage our infrastructure mentioned earlier. Instead, it glues them together and likely hides them away from the light. This sets us up for more of the same difficult testing setup each year.

Lastly, while being faster to spin up than last year, it is still just too slow. By taking so much time to spin up, engineers can be encouraged to leave their stacks running which allows them to become outdated and even worse burn cash while theyâ€™re idle.

Wrap Up

This is a lot of information, but before wrapping up I want to return to my initial question, is this all worth it? And also extend it to include the question, will we throw this away next year?

Yes, it always ends up being worth it, even if it doesnâ€™t seem that way going in. In many ways a single discovery from load testing can justify the engineering time investment by preventing a service disruption to our users. During the 2018 holiday synthetic testing, we were able to find a few platform issues that could only be teased out at scale.

- We perform a check on every email recipient to see if they would push the account that was sending to them over their monthly messaging limit. At synthetic scale this buckled the underlying persistence layer. This led us to check our production telemetry and find the same stress indicators under normal production load, which would only be exacerbated by holiday 5â€“7x scale.
- We append a custom unsubscribe link to all email messages sent. At synthetic scale the timing on this operation degraded significantly which led us to find a regression where the link was redundantly created. This also was observable in our production telemetry and would lead to degraded performance at holiday scale.
- When we process incoming open email events we do some post processing on the browser useragent. At synthetic scale this processing was inefficient and needed to be refactored. There was also a lazy loaded resource in this pipeline that was caught and refactored.

After refactoring the useragent post processing and lazily loaded resource in our event processing pipeline we can see in our telemetry the dramatic latency improvement.

Yes, it may be thrown away next year. It is a bit jarring to think that the framework we built this year may be discarded next year as we prepare, but I take comfort knowing that until then it can still be used to prove out infrastructure changes before they are released into production, and also that it served its purpose in 2018. I also think of it as a valuable research spike, if nothing else for learning a pattern that we didnâ€™t like. Realistically, I believe that ahead of 2019 we will work on a new pattern or framework, but be able to build upon the learnings from 2018, as the approach to 2018 did from 2017, and so on and so forth. If you want to be a part of the 2019 implementation, weâ€™re hiring.