Shopify's Architecture

TO HANDLE THE WORLD'S BIGGEST FLASH SALES

Bart de Water

Table of Contents

Introduction

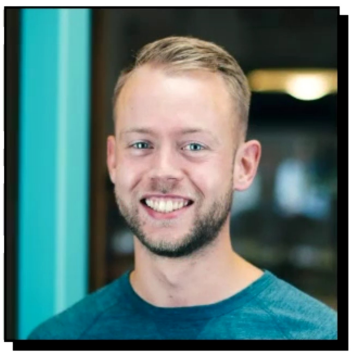
1

Original article at www.infoq.com (Oct 07, 2022)

TRACK
Architectures You've Always Wondered About

SESSION
Shopify's Architecture to Handle the World's Biggest Flash Sales

Bart de Water
Manager @Shopify Payments Team



De Water: My name is Bart de Water. I lead the insights and reporting team of Shopify Payments. Shopify Payments is our built-in payment processing solution that we offer to merchants in 17 countries around the world. We also support many payment integrations from third parties. I've been working on various parts of Shopify as payment processing systems for the past five years.

Shopify's Mission

2

In case you're unfamiliar with what we do at Shopify, our mission is to make commerce better for everyone. We do that by offering a multi-channel platform hosting millions of merchants, allowing them to sell wherever their customers are, whether that be via their online storefront, social media, and also in person at a brick and mortar or pop-up stores, all with a single integrated back office for merchants to run their business from.

Shopify handles some the the largest sales on the world

3

We've gotten really good over the past few years handling flash sales. That's why I'm giving this talk. A flash sale is a sale for a limited amount of time, often with limited stock. It's over in a flash because the product can sell out in seconds, even if there are thousands of items in inventory. You might think that's not new that sounds like a regular sale. Yes, you might have seen videos of crowded shopping malls around the holiday periods with customers rushing in as soon as the gates open. Today, these flash sales happen anytime of the year. The current iteration of flash sales was popularized by digital-first brands with product drops. This is a sale of a limited edition of something, for example, lipstick or a pair of sneakers. These brands create hype on their social media platforms, which then drives enormous amount of traffic and sales the moment the product is available. This type of sale poses an interesting engineering challenge, as the amount of merchants that we host grow and they grow their customer base, today's flash sale will be tomorrow's base load.

How Shopify is built

4

A quick rundown of how we build Shopify. Our main tool of choice for building backend system is Ruby on Rails with MySQL, Redis, and memcached as our datastores. We use Go and Lua in a couple of places as well, mostly for performance critical backend parts. On the frontend, we use React with GraphQL APIs. We use React Native for our mobile apps, including the point of sale that I just showed you earlier. You may have heard that Shopify's main Rails app is a monolith. We deploy this around 40 times a day as hundreds of developers worldwide are working on it. This is a familiar sight for folks, I presume. You are browsing for products, add it to cart, and check out. Then, this is the perspective from our merchant's admin. There's an order ready to be fulfilled.

We just saw three major sections of Shopify, and in this talk, I'll mostly focus on storefront and checkout since these two see the most traffic. You'll notice that storefront and checkout also have very different characteristics and requirements. Storefront is mostly about read traffic, while our checkout does most of the writing and has to interact with external systems as well.

Shopify Pod

4.1

Before we continue, I need to introduce a little bit of our terminology, a Shopify pod, not to be confused with a Kubernetes pod. A Shopify pod contains data for one to many shops, and it's basically a complete version of Shopify that can run anywhere in the world. It is more than just a relational data shard, which is why we use the name pod. We have multiple of these pods. These stateful systems are completely separate from each other. The pods run in a single region, and while the datastores are isolated the stateless workers are shared. This allows us to balance the load in a region where a shop on a certain pod is having a flash sale. In case something goes wrong with a certain pod, like an overloaded MySQL instance, this does not affect the other pods or the shops that are hosted on there. We also have multiple regions, and a Shopify pod is active in a single region at a time but exists in two with replication set up from the active to the non-active region. We can failover an entire pod to another region if need be, like some catastrophe happens.

How requests get routed

4.2

With that context, we can now talk about how a request gets routed to the appropriate pod. Part of requests is talking about domains, which involves a little bit of branding. A merchant at signup can choose a domain that we give them for free, a subdomain of myshopify.com. They cannot change it later, and sometimes they don't like that. You might sign up with cool t-shirts as your brand name first, and then later decide that actually you want to be known as t-shirt hero, so you can buy or bring to us tshirthero.com, and we'll make sure that everything is taken care of from there on. We'll use this in the rest of our examples.

Once a request for our store tshirthero.com enters our network, the first thing it runs through is OpenResty. OpenResty is an NGINX distribution that supports Lua scripting. We use these scripting capabilities for many things before it actually hits the Ruby on Rails application layer. Stuff that we do includes blocking bots and routing traffic to the right pod. You might be wondering, what do bots have to do with it? The limited edition merchandise sold in flash sales, like those sneakers, could fetch double or triple the original price on the secondary market, and merchants don't want their products to be resold like that. Bots also hammer our systems much more than real buyers do, so we try to block them. Since the pandemic started impacting supply chain issues, the bot problem has also spread to other products like graphics cards or gaming consoles.

Back to routing traffic, we have a Lua module called Sorting Hat that looks up the host in the routing table and finds the corresponding pod that this shop belongs to. The request is then routed to the appropriate region where the pod is active. Our Rails application receives a request and handles it. Our Rails application is a monolith. It's probably one of the oldest and biggest Rails apps out there. Zooming in on the monolith, we see that the checkout component needs to collaborate with a few others in order to get its work done. This is a list of some but not all the components we've subdivided the Shopify app into. We have the checkout line items, any discounts or promotions that are applied. There are taxes involved, shipping lines. Maybe you paid more for extra shipping. All of these add up to a total amount that needs to be charged to our buyer.

Shopify payment methods

4.3

Shopify supports many different payment processors and payment methods. I'll also use a credit card as an example here, since processing credit cards comes with some additional interesting challenges compared to other payment methods. If you've ever dealt with credit card payments, you've heard about something called PCI compliance. This refers to the Payment Card Industry Data Security Standard. The standard sets out six groups of requirements that need to be adhered to in order to make sure that card information is handled in a secure manner. These all sound reasonable, let's just implement all of them. Get our yearly audit done, and slap a complaint sticker on it. Call it a day. Easy.

Bringing all of Shopify's monolith in scope would be a problem. We have hundreds of developers shipping the monolith around 40 times a day. Having to meet all of these requirements in the way auditors expect them to would really slow us down. We also allow merchants to completely customize their shop's look and feel with HTML templates and JavaScript. We have an ecosystem of apps that can add functionality to stores, so we need to keep all that sensitive card data away from these things. If you were to right click Inspect Element on any of these fields, you would actually notice that these are iframes hosted on a completely separate domain from the merchant store and their checkout. This is how we keep the Shopify monolith out of scope. We just don't let it see any card data in the first place. I'll show you how that works.

When you submit the payment step of the checkout, these iframe fields with the card number, the cardholder name, the CVV and the expiry date are first submitted to an application we call CardSink. CardSink takes this sensitive data, encrypts it, and returns a token back to the checkout JavaScript. The checkout code then submits this token alongside all the other checkout data to the main Shopify application. At this point, the buyer is presented with a waiting screen with a little spinner that says that their order is being processed. What actually happens is that a background job in the Shopify monolith passes the token and payment metadata to an app we call CardServer. CardServer uses the token to look up the encrypted data in CardSink and decrypts it. It then uses this card data and the other metadata sent from the monolith to find the right payment processor to talk to using an adapter library. It transforms the standardized call into the processor specific API call. From there, the authorize request goes through the acquiring bank, the card network, and the issuing bank that ultimately decides to authorize or decline the charge. Assuming the charge is authorized, a success response is returned back and the checkout is converted to an order. This is when the buyer sees the order status page thanking them for their business. There's also confirmation emails sent, any webhooks for apps are fired. A merchant has their order to fulfill from the admin.

Black Friday and Cyber Monday (BFCM)

5

This was just a single purchase. How about those big flashes? Black Friday and Cyber Monday or BFCM, as we call it, is our largest event of the year. We have a lot of merchants organizing flash sales during the long weekend at the end of November. To give you an idea of the scale that we're talking about, here are last year's numbers. $6.3 billion total sales with requests per minute peeking up to 32 million requests per minute. Every year is larger than the previous one too, as we have more merchants choosing our platform, and the merchants' businesses themselves are also growing bigger. In order to support this growth on all these fronts, we need to circle back a bit and talk more about our product architecture.

Shopify's Product Architecture

5.1

Much like pods don't talk to each other, neither do shops. Every unit of work, be it a request or a background job, should operate on only a single shop at a time. This means that we write code that only cares about shops and not the pods that they're on. Since we don't care about these pods in application code, we can actually easily add more of them and horizontally scale out Shopify to support growth of the platform. As merchants also then grow at their individual pace, we occasionally need to rebalance pods by moving shops around in order to evenly spread load and disk usage. Of course, we need to do this with as little downtime as possible.

What also follows from the tenant isolation principle is that things that do need to work across multiple shops need to be either built in a separate application, or use our data warehouse for analysis. An example of such an app is Shop Pay, that's our one-click checkout solution that is an exclusive feature of Shopify Payments that allows you to easily pay for things across multiple Shopify stores. We'll be moving our example shop to pod 2, to show the process. We first start by copying over all data in MySQL, things like products, and orders, and we copy over all the rows where the shop ID matches. This covers all the existing data, but in order to make sure we get into complete sync, we also need to take into account new data as it comes in. We do this by replicating the MySQL binlog. The binlog is a stream of events for every row that is modified. We've open sourced a library that does this row copying and binlog replication. We call it ghostferry and it's available on GitHub.

These next few steps will go pretty fast. Once we've done our row copying and binlog replication and we're in near complete sync, we lock the shop for any writes and queue any writing data. This is also the moment when we start copying over jobs and other things in Redis. We then update the pod_id in the routing table once all of that is complete, and remove the lock. We can now serve requests from pod 2. This was pretty fast. We can do this for a lot of stores with less than 10 seconds of downtime. Even for our larger stores, we can do it in less than 20. You may have noticed that pod 1 still contains the old data for our store, tshirthero. It is deleted asynchronously in order to reclaim the space.

Bringing it back to the three sections we saw at the start, we had storefront, checkout, and admin. One of these is not like the other. Storefront has a very different traffic pattern, and it's also aiming to provide a different guarantee. It also sees much more traffic. Unfortunately, for our merchants, not every window shopper becomes a paying customer, we wish. Over the past couple of years, we have actually rewritten our storefront rendering code in a separate Ruby application from scratch, and we run it separately from the original monolith. While doing so, we had to make sure that we wouldn't break any of the existing merchant templates which are written in a language we call Liquid.

Again, OpenResty and Lua scripts were used to make the process seamless for our merchants and their

buyers. First, we ran the new rendering app in a shadow configuration to verify the HTML was equivalent before we started moving production traffic over. Having this flexibility in scripting also allowed to quickly revert back to rendering from the monolith in case of any regressions. Eventually, this new implementation became the standard one, and it's much faster and we can also independently scale it from other parts of Shopify. The next step in scaling this would just be to never render any HTML at all. Let the client do it and unlock infinite scale. We need to render the API responses still, so it's not that easy. Over the past 5-plus years, we've seen a rise in headless commerce, where our technically savvy merchants want to actually completely develop their own frontend. Often, these are single page applications using frameworks like React. We support that too. We're building a React based framework called Hydrogen, which can optionally be hosted on Shopify's network using Oxygen.

Load testing in Production 5.2

Now that we've talked a little bit about our architectural approach, how do we make sure it all works in practice? One way we do that is by load testing, and we do that in production. We have a tool for this that we developed that is called Genghis. Genghis is our load generator that spins up a lot of worker VMs that then execute Lua scripts. These Lua scripts describe certain steps of behavior that we see. We don't want to just hit the storefront a million times with the GET request, because that will only tell us how well the caching works. These Lua tests can be combined to describe end-to-end flows like browsing, adding to cart, checking out, and anything else we would see in real life. It's not a singular flow that we just unleash. For example, we have different flows describing logged in and anonymous customers. Since these two hit different code paths and caches, we want to test them separately and make sure all the edge cases are hit.

When we set up load testing, remember that Shopify has a sharded multi-tenant architecture. We run Genghis against a specific set of benchmark stores in our production environment, every pod has at least one. In order to realistically load test, we also need to stress our PCI environment. Most payment processors would rather not be on the receiving end of these load tests. Instead, we have set these stores up with a simple benchmark gateway, also a Go app. The benchmark gateway can respond with both successful and failed payments with a realistic distribution of response time latencies that we see in production. We run these tests at least weekly. We want to make sure we find out about bottlenecks first and not through a major merchant's flash sale. All these major pieces of infrastructure have their service level objectives set in an application called services DB, and if during the weekly load test these SLOs are not met, the owners are notified and requested to prioritize a fix. We consider being fast an important feature.

At our scale, it is inevitable that something will go wrong, so we plan for that. We need to understand our dependencies, their failure modes, and how these impact the end user experience. We do this by creating what we call a resilience matrix. Using this matrix, we can develop a plan to run a game day and simulate an outage. During these game days, we look to see if our alerts and automations trigger as expected and if our developers have the knowledge and process in place to properly respond. For the people who are paying close attention to this slide, no, Shipify is not a typo, it's the name of our application that provides shipping services such as purchasing shipping labels.

Circuit Breaker 5.3

One pattern we apply throughout applications to respond to failures is the circuit breaker. Like the circuit breaker pictured, once the circuit is opened or tripped, no current flows through. We use it to protect datastores and APIs, both internal and external APIs. Over time, a circuit breaker looks a little like this. We are talking to a service and we're hitting timeouts a few times in a row. Knowing that an external resource that goes down tends to stay down for a while, we instead, once the service breaker is tripped, we raise instantly and we don't wait for another timeout to happen. These instantly raising exceptions can be rescued and in some cases used to provide alternatives. A degraded service is better than being completely down. For example, if Redis is storing our user sessions, and it goes down, we could present a

logged out view instead, so that potential customers could still browse our storefront instead of getting an HTTP 500 served. Circuit breakers are also an obvious place to put alerting and monitoring around. In order to know if a resource is back up again, after a certain amount of time, we do need to allow a couple of requests through in the half open state. If these requests are successful, the circuit resets to its closed state. If not, the circuit breaker stays open and keeps raising errors for the next little while. Shopify developed its own library called Semian, which implements circuit breakers for Ruby's HTTP client, as well as clients to connect to Redis, MySQL, and gRPC.

You might be wondering, how are you going to test this failure in your unit tests? For this, we have an app called Toxiproxy. It's a proxy written in Go that not only allows us to simulate a service from going down, but also inject other kinds of failures. From our test suite, we can reliably trigger these failures in Toxiproxy and prevent accidental regressions. For example, in these test cases, we make sure that the user.get_session call doesn't error out if Redis is down. One interesting aspect to consider is what do we scope our circuit breaker actually to? Some services can be partially down, and with a subset of requests failing, most of them are fine. A reasonable default from the Semian README pictured here, is keying the circuit breaker identifier off of the API endpoint. The Semian configuration lambda is called in every HTTP request with the host and port arguments forwarded. We concatenate these together to create the identifier for the circuit breaker to check. Most of the time, this is fine. We don't want to know more about how the other side works, because we don't want to tightly couple ourselves to an implementation on the other side.

There are a couple of cases where it might make sense to deviate from this. Payments, for example, have a country dimension to them. Of course, there's regulation, but also acquiring done is at the local level for lower costs and better authorization rates. Even if you call the same host API worldwide, behind the scenes, it actually takes a country specific path. By adding the country to the circuit breaker identifier, if there's some upstream outage that prevents payments from being processed for Canadian merchants, we don't want the open circuit to affect American or Mexican merchants, and blocking their payments from going through. Another example here could be the shipping label aggregator. It might make sense to incorporate the name of the carrier instead of a country name.

Idempotent Request and Action

5.4

In the phase of failures, we'd like to retry before giving up completely. The side effects of retrying a call like the one pictured here could end up costing someone money. That's not just something that quickly drains the trust of our merchant's customers, but if these double charges are not corrected after the fact, this could also end up with the cardholder opening disputes with their bank to get their money back. This process would in turn end up costing the merchant time and money in dispute fees as well. In short, for processing payments, we want exactly-once semantics when we send this type of request. We can do that by including an idempotency into requests that uniquely identifies this payment attempt.

Resumption

5.5

At Shopify, we've written a library called Resumption to make it more easy to write idempotent actions. For the Ruby developers, we're looking into open sourcing this. We define an action class that contains multiple steps, and each step of progress made is recorded in the database. Each step describes the action it takes and how an idempotent call can recover any state before continuing if it's a retried call. In this example, we have the same idempotency_key shown in the earlier example. Unfortunately, when we want to actually call out to our processor, an error is raised. When our client retries the request with the same idempotency, we first look up the progress stored in the database. We find it and then we run through the recover steps first, before trying to call the remote API again. This time, it succeeds, and we handle the remote response given.

Peak CPU Across Shopify

5.6

This graph visualizes how everything I just talked about culminated over the course of the year. We scaled up our systems. Prepared the resilience matrices, ran our load tests and game days, and our merchants had another successful Black Friday, Cyber Monday weekend. You'll notice in this graph, two types of load tests were run last year. The architecture tests focused on testing the new components introduced, like the storefront renderer, and changes to our MySQL infrastructure. We ran these tests early to make sure that any got you's were caught out early, and that we had time to address them. Later in the year, the scale tests were the more traditional load tests for which we scaled up to 2021 levels and made sure we were ready for the actual BFCM event.

Recap

6

We started with how a request gets routed to the right Shopify pod, and how we can pay a checkout with a credit card without exposing the card details and the PCI compliance requirements to the main Shopify monolith. Then we talked about scaling up for Black Friday, Cyber Monday by adding new pods, moving shops around, and extracting storefront rendering from the Rails monolith. Finally, we talked about doing load testing in production and our strategies for handling anticipated failures gracefully, using circuit breakers, Toxiproxy, and idempotent payment APIs. A lot of these subjects are also covered on our engineering blog, which you can find at shopify.engineering.

Questions and Answers

7

Ignatowicz: Some special customers that have a high load and really big stars, they receive a special treatment on Shopify's infrastructure, is that correct?

De Water: As I would say with any multi-tenant system, there's always that handful or maybe few dozen tenants that are 10, 20 times bigger than the next set of customers. Given that we have a sharded or a poded architecture, some of these extra-large merchants have an entire pod for themselves. Yes. In other cases where we have just regular big merchants, we have other systems in place to make sure that when they do a flash sale, they don't monopolize all the capacity and that smaller merchants who are co-located on that pod also have a chance to conduct their business.

Ignatowicz: How do you test this to prepare for a flash sale to avoid that one big hotspot turning off the small merchants?

De Water: This is a continuous process. Once a merchant signs up, they're assigned a round robin to an available pod. Then it might just happen to be that that one particular merchant is going to be one of these extra-large flash sellers. That's why we have the shop mover so we can adjust to see how merchants are behaving, how they're growing on their pods, and then rebalance as needed.

Ignatowicz: I imagine that pods are very large pets. How do you run ownership of the main pets?

De Water: I'm going to assume here you mean pets as in pets versus cattle. Cattle being completely automized rollouts of things and pets being named servers that you babysit, and typically are hand built and set up. Pods in that regard are cattle. It's a way to reason about the fact like, where is this merchant? Is this merchant on pod number nine? Then, yes, that is a complete set of Shopify that runs in isolation. We can set up new pods very easily and even recreate them if need be. For example, when we have to do Kubernetes upgrades, we can run an entire pod in a single data center, upgrade the passive data center, or even just completely rebuild it over there and then move everything back. It's definitely not pets.

Ignatowicz: Just curious, if you looked into Temporal before building Resumption, and if so, how does it compare?

De Water: We did not. If you look at Shopify's background job system, like history, Tobi, our CEO, was a Rails core member back in the day. He built a Delayed Job where we insert jobs into database, and a separate process pops them off a queue, works them, and updates the database again. If you can imagine that at some point MySQL is just not meant for all that writing and updating of jobs in high throughput scenarios. Then GitHub came along with Resque, which was Redis based. For the monolith, we're now running a heavily customized fork of Resque. It's basically something new, at this point. I only heard about Temporal quite recently. Yes, Shopify's background job system has evolved in its own way over the years.

Ignatowicz: How do you determine, what is the limit on pod size?

De Water: That's with load testing. I had a slide on that, where there was the various load testing that we did. There were two types of load tests that we did do. It's like we have the regular load tests, where we just are more concerned with the correctness of all the protection layers working. Then there's also the scale tests. Those are the ones where it's just like, let's try and break it. Let's just keep throwing more checkouts at it until we find the limit and then figure out, is this a limit that we expected there to be there? Is it something that we could solve? Or, given that Black Friday is in two weeks from now we're just going to assume this is the limit, since it is good enough, and we don't want to introduce last minute changes for stability.

Ignatowicz: Is this monolith something that's going to be broken into microservices?

De Water: Not anytime soon. There is a blog post on our engineering blog called deconstructing the monolith, which goes into our efforts to componentize these various parts of Shopify. The way that we're thinking about it right now is that things that pertain to a single shop, like an order or a chargeback or a refund, it's just easier to keep it in a monolith. We do want to make sure that we don't accidentally evolve into a big pile of spaghetti. We built a system that can enforce these component boundaries and create API boundaries, like you would have with a microservice, but instead we're not going over the network to make these calls. This way, we are looking to combine best of both worlds, no latency, no extra complicated debugging with logs from extra systems, but still have some of the isolation principles.

Ignatowicz: How does the rebalancing work? Can it be done without downtime?

De Water: Very little downtime during the cutover phase where data is in like 99.9999% sync, where we do have to say, at the old pod for like, ok, we need to stop creating data here, otherwise we're never going to be done. Then update our routing layer to now point it to the new pod.

Ignatowicz: What was your rationale or your theme behind choosing a pod based architecture? In that moment, on the rationale that you did in that moment, what are the challenges that you didn't foresee?

De Water: This was before my time. I believe Shopify outgrew a single database around 2010, 2012, I think it was. Basically, I imagine part of it was that sharding is like the next logical step in case you cannot just throw more money or bigger hardware at the problem. That's how it continued to evolve. Going for a rewrite around that time, I imagine that Shopify was already too big, and new features needed to be shipped in a pace that made a rewrite just not tenable.

Ignatowicz: How do you aggregate data from all the pods, together with analytics?

De Water: All of the pods' data are fed from a read replica into our data warehouse. I think it's Presto that we use under the hood there. Basically, our data scientists or even myself, in case I am curious about something, I'll need to go to the data warehouse to aggregate across all of our data.

Ignatowicz: What is the challenge that you're working on right now in your team?

De Water: Right now I am looking to make things even go faster than they already are going today. There's a couple of initiatives that I'm looking into with our financial partners to optimize our request patterns, to see that we can make individual payment requests go faster. Then, therefore, maybe defer some of the work to be after the payment has been done. Then that way make the critical moments, which is transaction time, faster for them and for us. Then, therefore, also for our buyers, because they are not waiting at that spinner for like 3 seconds, but maybe 2 seconds. It's good for all involved.

Ignatowicz: Can you elaborate the scaling storefront from checkout? I assume all the system's storefront and checkout are part of the monolith.

De Water: Yes. That hooks into the multi-repo or mono-repo question as well. The new storefront renderer system that was built that is now completely separate from the monolith. It's a very recent part of our evolution, that only happened in the last two years or so, that that project kicked off and is now fully shipped reasonably. That means that for over 10 years, we've scaled up monolithically very well. Checkout is still part of the monolith, because it is tied to so many other systems. If you reduce storefront to its most simplest form, it's almost all read only, very heavy cacheable traffic. Checkout, however, needs to write to a lot of things on the database. It needs to store the fact that the payment happened from the checkout, create an order object, create shipping orders. The balance that needs to be paid out to the merchants needs to be updated. Checkout is in so many places in that regard that building that into a completely separate service, it would be possible. Nothing is impossible, if you just have enough time and willingness. That would be such a monumental undertaking. You're basically rewriting Shopify at that point. I don't see that happening anytime soon.

Ignatowicz: How does the storefront talk with the checkout?

De Water: It doesn't. That's the beauty of it. What happens is that, you have a cart in which you're placing the products. The moment that you start entering information like your email address or your shipping address, that's when we start calling it a checkout instead of just a cart, because it seems there's the intent to pay at some point at the end of this. Other than that, it's very decoupled. Literally, the cart is purely client side until you actually start storing some buyer information.

Ignatowicz: Coming back to the topic of how do you enforce boundaries between the teams, like an interface to people doesn't create the spaghetti code? How do you enforce these? Do you have some features and functions to keep your architecture to test that the code doesn't bother? Because the tendency of people is when they are under pressure, they usually take shortcuts. How do you enforce these? Do you enforce these automatically or not?

De Water: We do use that automatically. It is an open source library called Packwerk. Using static analysis, this allows us to analyze, is something in the payment processing component talking to something in the order component, but is directly using the order object without going through the order public API to update the amount. You can imagine with an order, there's a lot of state on it. There is the amount before discount. The amount before taxes. The amount before discount and taxes. If I, from the payments component, just start updating one little number, their internal consistency is all messed up, and it's going to lead to a lot of pain for everybody involved. Especially the order team, I'm using them as the example, because they are like the checkout, intertwined into almost everything else. They're the ones that have traditionally felt the most pain there. They were the ones that were very eager to adopt these new practices and make sure that there would be no way anymore for people to internally mutate that state, either intentionally or by accident.

Ignatowicz: This library that you shared is like a feature and a function that breaks the build, basically, if I do this?

De Water: Yes. As you can imagine, initially, we just started logging warnings. Eventually, these warnings are added to a list. Being like, ok, we know that everything on this list that this is bad, but if you introduce a new thing that is not yet on that list, then at least it breaks the build. Which means that me as the breaker needs to now make a call, am I going to add to that list? Which is not great, but sometimes for reasons of urgency, you sometimes just need to add a little bit more badness before you can make it good again. In other cases, it's just, `No, go back, redo your PR. Your use case does not warrant making it worse than it is.'' Boy-scout rule, make it better. Typically, at Shopify, we're all trying to make the system better. Usually, if you don't know how to do it, or maybe you're missing an internal API or not sure which one to use. You hit up the order team and being like,` The build broke. I understand this was not how it's supposed to be. Can you help me figure out how I am supposed to do it then?'' It's one of these things where the answer is, it depends.

Ignatowicz: Usually, these type of problems behave in that way.

One more philosophical question about the feature is that Shopify is well known by being one of the highest scale of Ruby use cases in the world. How do you feel the future is of the language and the platform and ecosystem?

De Water: Shopify employs people who work on Ruby and on Rails itself. In Ruby 3.0, there is an experimental just-in-time compiler called YJIT, or yet another just-in-time compiler. Where earlier efforts in Ruby to build just-in-time compilation have shown effectiveness in benchmarks, the elusive thing was to see an actual speedup in Rails applications or web apps in general. Since this team was working on Shopify, of course their mission was to make a JIT that would actually also speed up Rails use cases. In Ruby 3.0, it shipped as an experimental feature. I think it was somewhere around 20% speedup in an average Rails app. They are for Ruby 3.1, which is due to be released this Christmas. They've decided to rewrite it in Rust, because they figured it's going to be easier to maintain in Rust than it is in C. While they are rewriting YJIT in Rust, they've also seized a few opportunities to make some small improvements along the way, as well.

Ignatowicz: This is something that we are looking forward for the ecosystem, especially because there's a

lot of investment by Shopify in this effort, because I follow this work closely. If Shopify works, you'll soon be able to produce a great use case for this technology. This could be a breakthrough for the whole Ruby ecosystem, I think.

De Water: You can imagine, at our scale, running so many application servers, and many applications, a 20% increase for us is very sizeable as well. That means we have to scale up less servers for the next Black Friday.

Ignatowicz: Probably on your scale of the code base, you probably have all possible corner cases in Ruby and Rails framework to be tested by such technology, so you'll be a good validation for the whole community on this.

De Water: Yes. Our monolith runs very recent development versions of Ruby and of Rails. Typically, when something changes in Rails, we're among the first to catch any regressions, either in performance or compatibility. We can then fix that and contribute that back, so by the time that version actually is released, in a stable version, it's basically been tested for at least a year at Shopify already. We also then spend time upgrading Gems or whatever that needs fixing.

Ignatowicz: Do you use anything similar to Packwerk for your Go code as well, to avoid the dependency having spaghetti code?

De Water: My work doesn't involve a lot of writing Go. I don't believe we have something like Packwerk in Go because we don't have a very big app, like our monolith in Go at all. Go is mostly used for very specific pieces of our infrastructure, less so on the merchant or buyer facing things. They tend to be small, very focused apps. Nothing big and monolithic. I don't know Go that well. I think also Go's packaging system right out of the box is more strict than what a typical Rails app would prescribe you.