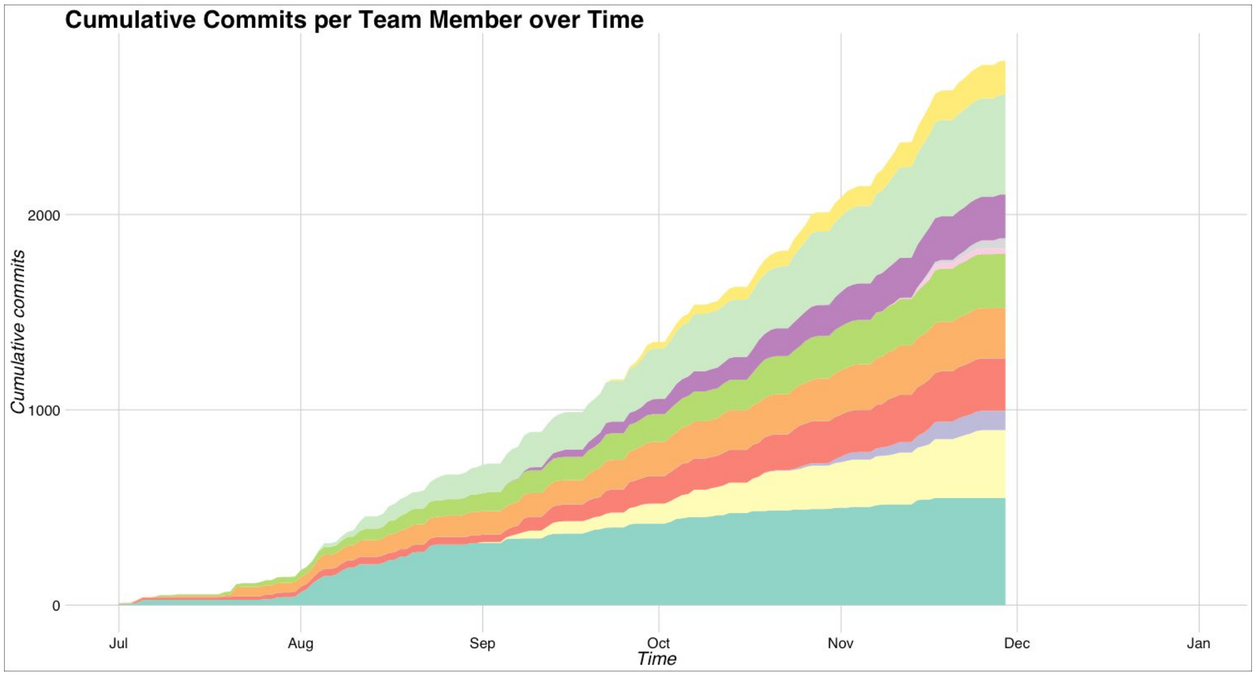
New team member on boarding

When you join the engineering team at fashionTrade, you will make your first commit on the second day of work the latest. This is also very likely to be your first deployment to production (more on that later). We started coding July 1st 2016. As expected in a healthy team, contributions / productivity are growing faster than the team. In the chart below you can clearly see when new members joined as well.



Your first production commit on day 2. How does that work?

As part of the onboarding, you start your first day with about two hours of introduction / lecture from one of your tech colleagues (apart from that, you also get to meet sales, operations and learn a great deal about the fashion industry). Your second day is a full day of pair programming with you in the driver's seat. This means on your first exposure to the codebase, instead of having to shave every yak yourself, you can just ask the person sitting next to you whenever you get stuck.

In most cases, the first task will be a small one, so if you get the work done on the first day of coding, chances are that someone will review the pull request right away and it gets merged to master, after which it is automatically deployed.

Stack

Infrastructure

- We use Google Cloud Platform
  - Cheaper than AWS (https://thehftguy.com/2016/11/18/google-cloud-is-50-cheaper-than-aws/).
  - Provides what we need.
  - Sometimes immature, but progressing quickly.
  - So far we've had no exposure to the support process, but have managed to get all the help we needed from just talking to people at Google directly; AWS feels too big right now to be able to that.
  - There are compelling reasons to put some trust in Google's software engineering workforce.
- We use the following platform services:
  - VMs
  - Deployment Manager (similar to AWS CloudFormation).
  - Container engine (basically hosted Kubernetes).
  - Load balancing.
  - Networking.
  - Cloud Datastore (SaaS NoSQL DB).
- Apart from Google's tools, we use Ansible for provisioning everything that doesn't run in containers.

Shared platform services

Stuff we run ourselves, because there is no SaaS / cloud alternative that meets our requirements:

- Kafka (runs on VMs, not containers)
- ElasticSearch (also on VMs)
- Jenkins (fully on Kubernetes)
- openVPN server
- Redis
- Internal DNS (tricky stuff, much delegation between top level and container clusters and whatnot, but it works)

External SaaS things we use:

- Artifactory (for internal library dependencies, Maven, NPM, etc.)
- MailChimp / Mandrill (transactional email)
- Auth0 (external authentication provider)
- Logz.io
- OpenProvider (domain registration and management)
  - Protip: get a local company for this that's operated by professionals

        - Enough people lost control of their domains because hackers socially engineered their way around GoDaddy's support security. This can mean the end of business.
- We use a little bit of AWS for storing off site backups
    - Still working on this process
    - Push backup from GCP to AWS
        - When it lands on AWS' side, use a AWS lambda to move the raw files to a different bucket where the original user doesn't have access anymore
        - This way, if the core environment were compromised, we can still restore the entire platform from off site in a new GCP project / account
    - Similarly, also push backup to a storage bucket on GCP that is owned by a different project / user.
    - We aim to test the restore procedure at least once / 2 weeks.

Services

- Our main programming language is Java
    - We use Java >= 8, with heavy use of the functional programming constructs available there
    - We do async based on future composition using CompletableFuture
    - Use java.util.Stream wherever appropriate (there are very few for-loops in our code)
    - We use immutability and constructor injection or builders where required
        - This is to avoid shared mutable state.
        - As a bonus, there's reduced risk of stop-the-world with the G1 garbage collector.
        - If you know enough about the garbage collection internals to explain why the above is true, we should talk…
- Most services are built on top of Dropwizard, a lightweight JAX-RS based framework for REST-like services.

Persistence

Three persistence services:

- Google Cloud Datastore (GCD)
    - The only store that's allowed to be authoritative in our stack
    - SaaS based NoSQL datastore
        - Has transactions
        - Unit of storage is an entity. Entities are organised in groups in the context of a transaction
        - Has selective indexing
- Kafka
    - Used for async interservice (data) dependencies
    - Not authoritative in the sense that Kafka contents can be re-created from GCD

        - Is used for retention of master data events, etc.
        - Use queue compaction for refilling derived stores (e.g. build a new search index by resetting the Kafka consumer offset to 0)
    - ElasticSearch
        - You know, for search
        - Never allowed to be authoritative / primary transactional persistence for a service
        - Indexes are expected to be recreated frequently

Frontend

- Frontend is a single page app created using React + Redux.
- No server side rendering
    - As a B2B platform, all users are logged in
    - No need to render for Google (we can’t, the information is protected)

Runtime / deployment

Services run in containers on Kubernetes:

- Back end:
    - Alpine based docker image with Oracle JVM
- Frontend
    - Simple container with nginx for serving static files
    - No server side rendering

Services, my of course!

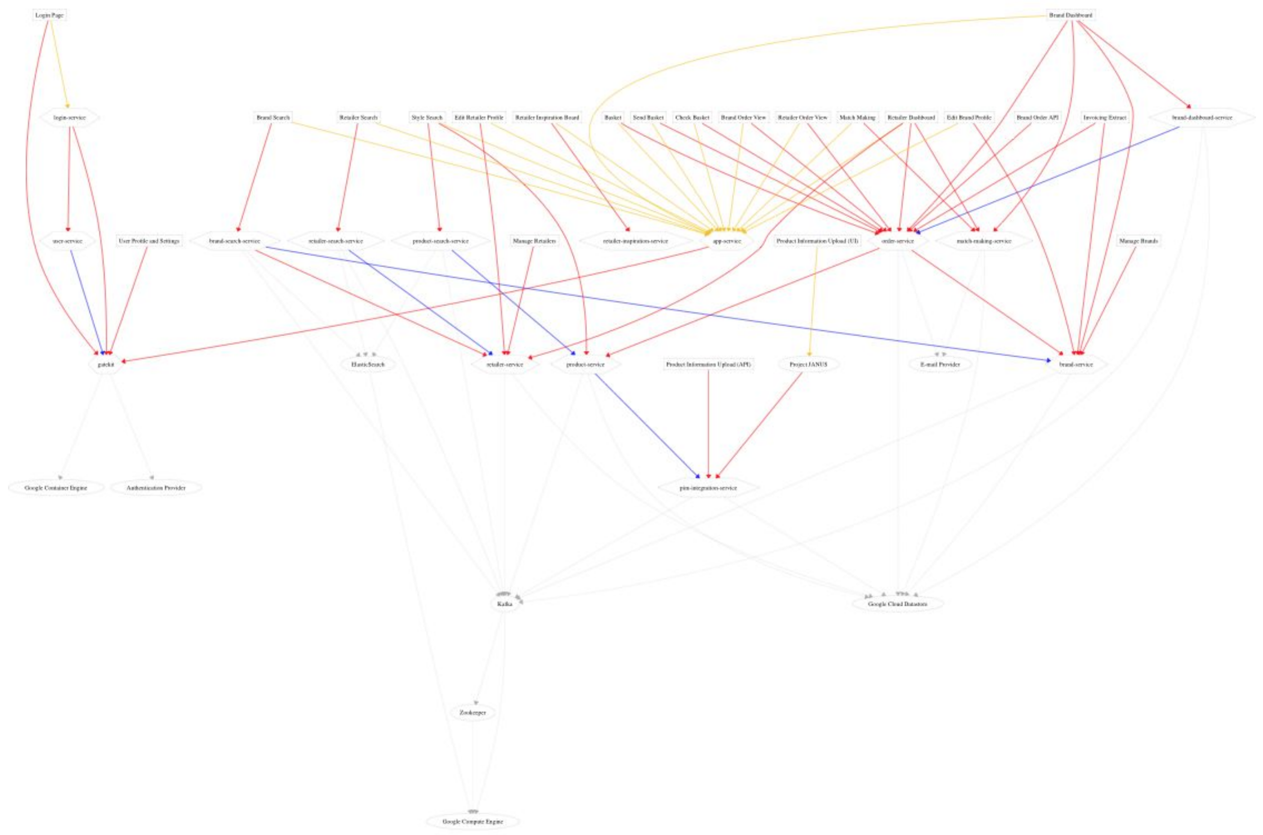
Yes, we do microservices:

- Different back-end functionalities go in different services
- Services cover function or technical domains
    - E.g. everything that has to do with orders goes in a order-service
    - The frontend is a separate service
- Services are autonomous
    - Manage their own state
    - Are responsible for their own data
        - Including schema evolutions / updates on deployment
        - Building search indexes
        - Backup / recovery
    - Two types of inter-service dependencies:
        - Asynchronous
            - Pure data dependency: one service consumes data that another produces on Kafka
            - Dependent service keeps its own derived state in its own storage

        - Dependent service can rebuild complete state by resetting Kafka consumer offset (this is how you build a new search index)
        - Kafka messages are keyed based on a business key; using queue compaction will always retain the latest version of every entity on the topic
    - Synchronous
        - Direct service dependency, using HTTP call
        - Services expose Swagger definitions
        - We use swagger-codegen to generate client code for internal services
            - For Java, we use a custom swagger-codegen template
                - Handles async using our own idioms
                - Can do transparent caching of calls (not implemented currently)
                    - Don't use caching services (e.g. Redis / memcached); complex and prone to staleness / invalidation errors
                    - Server side of services can control valid caching strategy using cache control headers; clients only have to implement the spec, which we control wholistically because it's generated code
            - Publish documentation generated based on Swagger definitions
                - Updates automatically after deployments
                - Also used for external facing services (our users use these to upload product information).
                - Uses a tool that allows to mix in handwritten documentation with generated documentation at specific hook points.
                - Generates service docs automatically (https://apidocs.fashiontrade.io, currently; will move elsewhere)
                - Never stale with actually deployed Swagger definitions

Dependencies

Dependencies are tricky. We keep track of them using a single YAML file that lists everything we have in terms of functionality, our own services, infrastructure services, platform components, external components (SaaS) and how all these things relate to each other. The YAML file is used to generate our platform architecture overview:

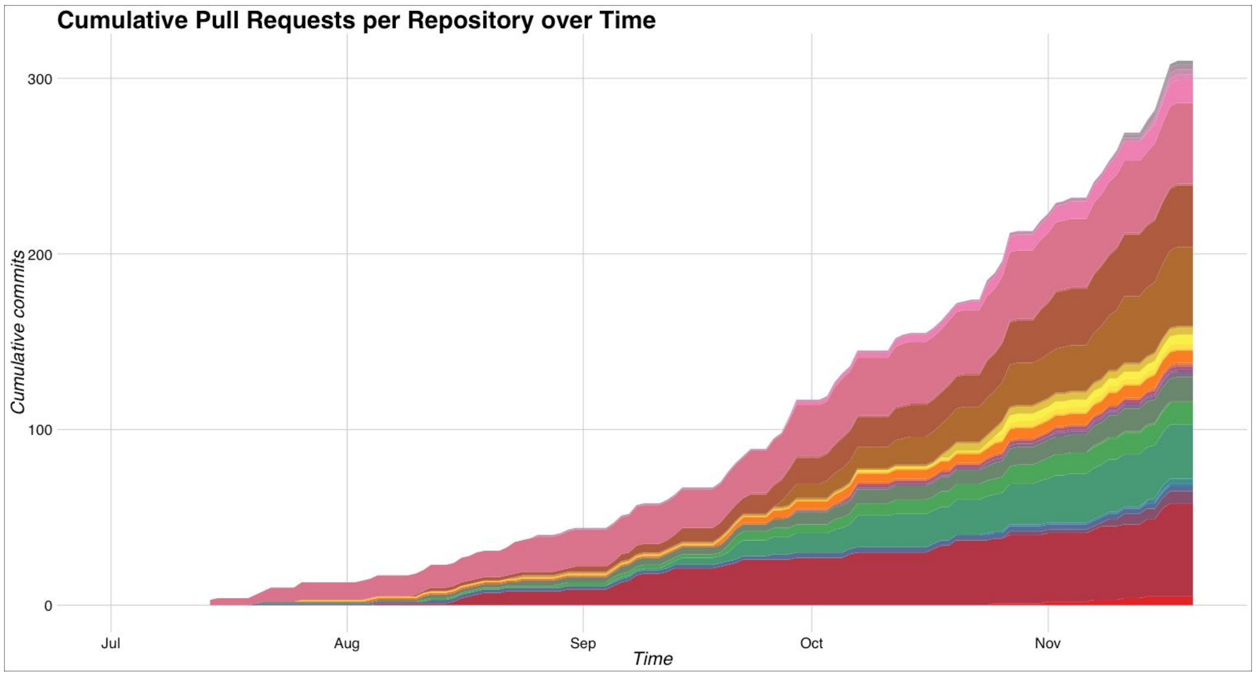


We print this on a big paper and hang it on the wall to explain new devs joining the team how everything relates (we make a new print about once a week). This shows features (functionality), services, direct dependencies, data dependencies and infrastructure in one go. We never touch a diagramming tool; this image is generated, otherwise it goes stale like any piece of documentation that's laborious to maintain.

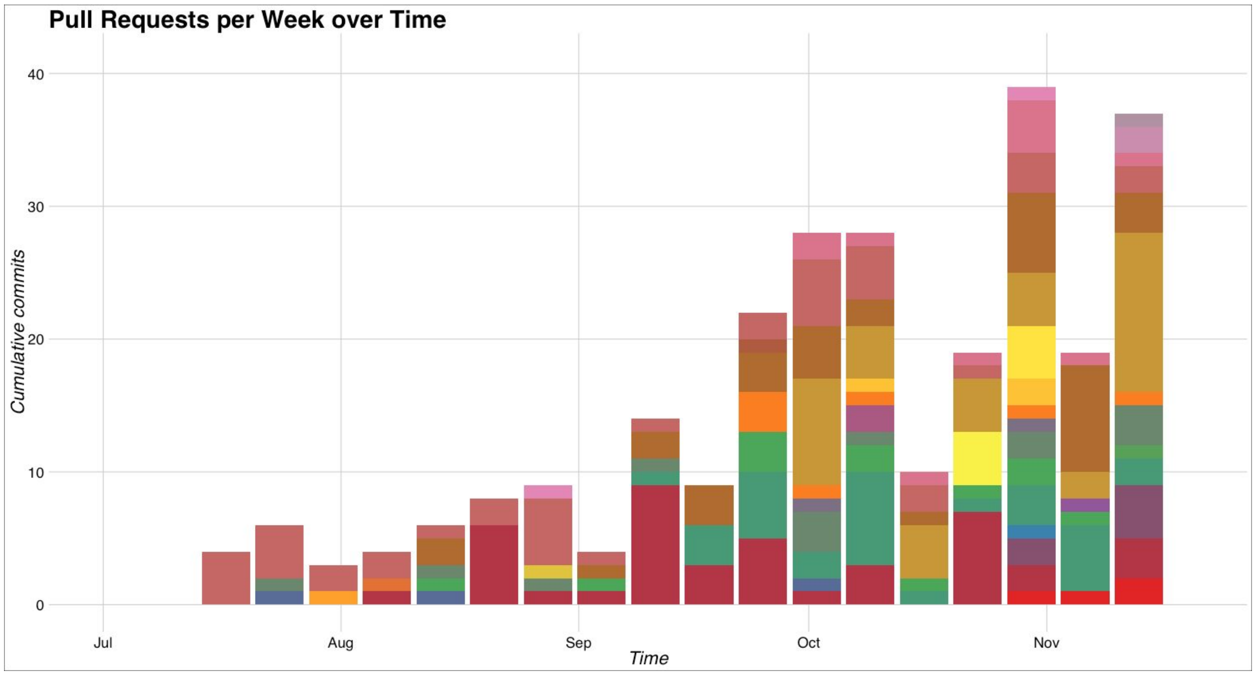
Organisational scalability

One of the goals of microservices is organisational scalability. Deployments are independent, functional ownership is distributed across the team.

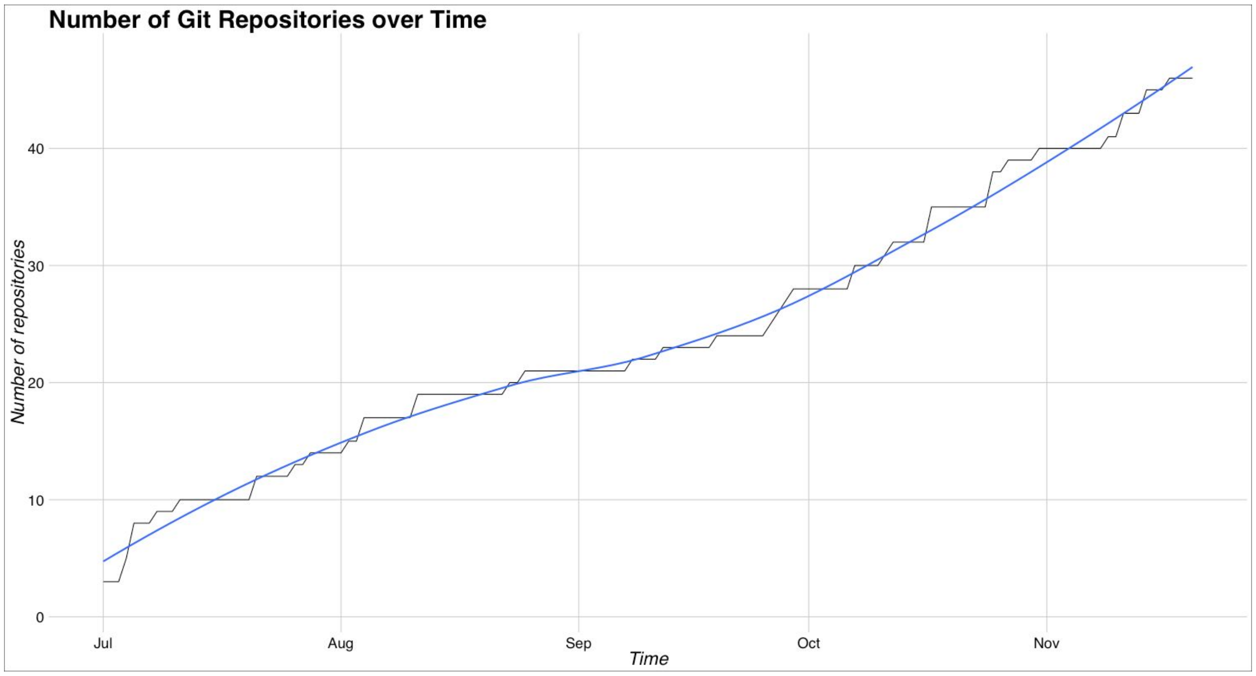
So, how often do you deploy? Whenever we merge a PR to master. Here's how often that's happened thus far:



The colors are individual repos / services. Here’s the breakdown in number of production deploys per week:

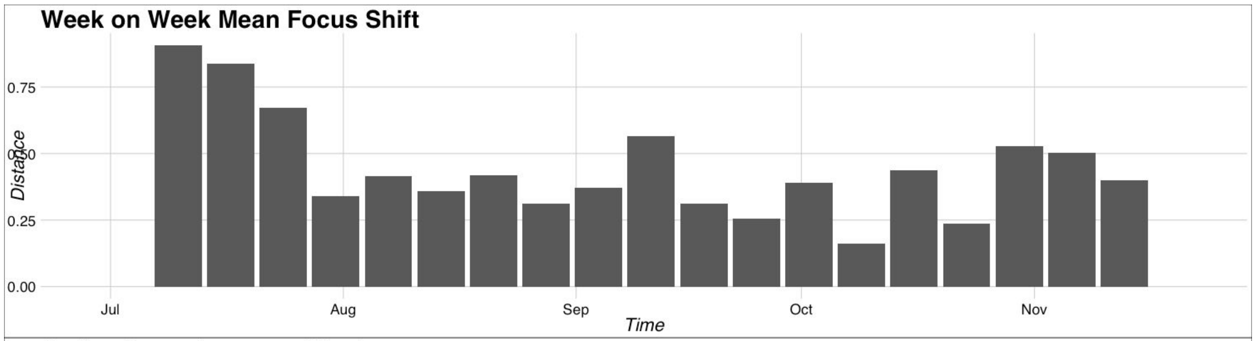


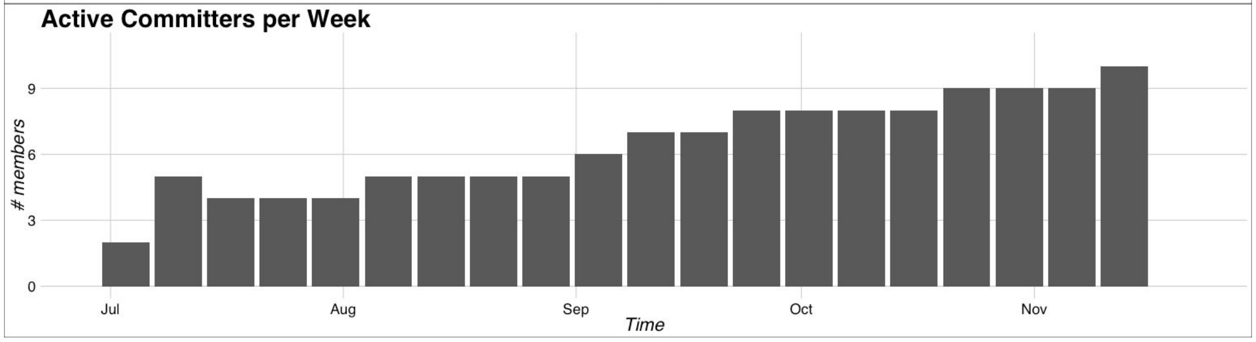
The thing about services is that you can keep track of things by keeping track of repos, not packages or something else that’s less explicitly a unit of scalability. As a result, you get many repos:



Then, for organisational scalability the question becomes: *who works on what?* At FashionTrade all functional discernable parts of the platform have primary owners, who are the most likely to pick up changes to their own parts, but not always. Everyone is allowed to create a PR against services in someone else's domain; you're never blocked on someone else for a change. Of course you cannot push directly to master, but always go through a PR with a review.

A nice thing about repos as the unit of scaling the process, is that it allows you to look at development focus. If someone works on the same repos a lot, it means they're focussed on a particular part of the system instead of going all over the place. This allows us to quantify focus: we look at how much focus change people have from week to week. It looks like this:





The top chart is how much change there is in what parts of the platform individual people work on and the bottom chart show the growth of the team over the past couple of months. You see that there is a lot focus shift in the beginning and that it stabilises a bit over time. There is a slight effect of new people joining the team as well, but it’s a bit too soon to draw any conclusions there.

(Note: the focus shift metric is not just counting the changes in the repos that someone works on this week vs. previous week. Instead, to generate this metric, we represent team members as a distribution of their commit volume over repositories and then take the cosine distance between the distributions for different time periods. This is intuitively more correct and generalises better to extended time windows.)

How does deployment work

Short answer: automated and scalable.

The works:

- The build for a service always produces a Docker image
- The Docker images contain metadata about the deployment
    - Information about routing
    - Information about required authentication
- So, metadata is distributed; there is no single place where all deployments are listed or something like that
- Builds are done by Jenkins
    - A “seed job” automatically creates builds for all new repositories

    - Repositories adhere to a naming convention for services, libraries, docker images, etc. so Jenkins knows what to do with it
    - A build pipeline is created for all development branches
    - A deploy pipeline is created for all master branches
- The deploy step in Jenkins:
    - Discovers the metadata in the Docker image
    - Uses the metadata to generate a Kubernetes manifest
    - Applies the generated manifest against the target k8s cluster (e.g. production or some other environment)
    - The service metadata from the Docker image is added as labels to the deployed service, which makes the same information available at runtime
- There is a API / service gateway (which is also a service), responsible for two things:
    - Service routing
    - Authentication
- Service routing
    - The API gateway listens to changes in the k8s cluster.
    - When new services are deployed, it fetches the service metadata from the cluster (the metadata which was copied from the Docker image, which was copied from what's in the Docker file, which is in the original source repo, remember; hence it's distributed metadata).
    - The API gateway dynamically updates its routing state based on the metadata for deployed services.
    - This way, routing is not configuration, it's runtime state based on the currently deployed services in an environment.
- Authentication / authorization:
    - User authentication happens in the API gateway
    - Individual services don't know about the authentication provider
    - The gateway translates the auth information into a number header that are set on the upstream internal request (the gateway is a reverse proxy)
    - Internal services use the header to check for the user's principal and role information
    - In the current implementation, the gateway authenticates users against Stormpath, a SaaS based auth solution

About productivity

How do you establish development productivity?

- Idioms, idioms, idioms!
    - There are five thousand ways of handling async call in Java; we use one and only that one
    - There are five thousand ways of writing a Kafka producer in Java; we use one and only that one
    - There are five thousand HTTP client libraries for Java; we use one and only one

    - On top of that, most of our HTTP client code is generated, because it’s for internal services
    - You get the point
- When something isn’t standard
    - Of course there is one-off stuff that isn’t common enough to establish a idiom
    - Whoever is working on those services will come up with a solution and make it work
        - It’s not like we have software architects or other inflated importance roles
        - You are responsible
- When the idiom doesn’t make sense for your use case
    - We can change things
        - E.g. we can add services in other languages than Java (we haven’t yet)
        - E.g. not all services must be based on Dropwizard (API gateway is based on Undertow, as it is a more low level solution and there are different performance considerations)
    - Whenever such a need arises:
        - You bring it up in the upcoming full development team tech meeting (happens weekly)
        - If you can’t wait for that, you suggest pulling a design discussion into the sprint
            - These are always time boxed
            - You invite a relevant part of the team to take part
    - You see: our instrument for inflated importance is not a role, but a discussion to reach a shared understanding of why we do something
        - Avoids allowing people to impose their particular design religion onto the team
        - Avoids the organisational bottleneck of having only one person allowed to come up with the solution
        - Makes it more fun to work here

Wheels reinvented

It’s not our mission to write more software than strictly necessary to build our platform. That said, there are things we have reinvented instead of used existing stuff. The most notable occurrences are:

- The API gateway
    - Similar solutions exist
    - HAProxy could have also worked
    - Yet, we decided to roll our own, so we have full control over something very important
    - It’s also a cross cutting concern that will later arise when we start to think about data collection / AB-testing / online experimentation; this stuff is tricky (see here:

        https://nbviewer.jupyter.org/format/slides/gist/friso/0cf541bf96357036eba119e656cf1895)
- Google Cloud Datastore client library (wrapper)
    - The default one for Java isn't fully async, doesn't work with our desired future implementation (Java's standard CompletableFuture) and doesn't support entity mapping for our idiomatic immutable Java classes.
    - We decided to write our own wrapper around the existing gRPC client.
    - This is worth it; see the point about idioms above.

A word on technical debt

Nobody's perfect. We take short cuts. We're in a hurry. The most compromising thing we did was to use a shared library to define the entities that are published to Kafka. This shouldn't have been a library, but a collection of schemas instead. Something that defines a shared representation should not have logic in it. Libraries are code and will always attract logic over time. That's happened. We will get rid of this after we go live. Promised.

Why didn't you use <insert technology here>?

Scala?

We find that the Scala community has a hard time converging on idioms and preferred ways of doing things. That makes it risky to an extent; not from technology perspective, but from a people perspective, which is outside of our control. The pool of people who write Scala and agree on how it's done is too small to setup a reliable hiring funnel in the Amsterdam area at the moment.

That said, it's definitely an interesting initiative; I actively keep track of how it and its community evolve and hope it one day reaches the stability that I would dare to bet my career on.

Python?

We use Python for different internal tools, infrastructure automation and one off analyses. There's no real issue with using the language for any outward facing services, but so far there hasn't been a need.

Node.js?

I think node.js is great! It combines all the readability of assembler with all the performance of JavaScript.

That said, the V8 VM is one of the best in class and a lot of optimisation work is done for it (because more than half of all web pages rely on it being fast). Their JIT implementation not only does on stack replacement, but also in loop replacement where it even tries to minimise register and cache line pollution when making the code switch. If you have in-depth understanding of what all those things are, we might allow you to write production node.js code if you insist…

Go?

Basically because we use Java instead. There’s no compelling reason to adopt two languages for core backend systems at this point. They both have concurrency and threading models, library ecosystems and battle tested runtimes. Choose one and move on. We chose Java.

Google Cloud Pub/Sub?

It doesn’t have long term retention and queue compaction, both of which Kafka has. We’re in ongoing conversations with Google’s product manager for pub/sub to see if we could at some point use it for our platform. Troubleshooting Kafka isn’t a chore that’s high on my list of fun activities for Sunday morning 3am.

What is it really like over here?

We’re a fashion company. It’s fancy here. Look, our fridge is a denim wrapped Smeg; it dresses better than most if not all of the dev team:

Of course we fill it with hipster sodas free for all and craft beers for Friday’s (or whenever you feel like it at the end of the day). We are in the TQ building, which is full of activity and startups celebrating their little and bigger successes.

But also, we’re a team of focussed professionals with the ambition to make our product work. You should be challenged here.