At Monzo, we're building a banking system from scratch. Our systems must be available 24x7 no matter what happens, scalable to hundreds of millions of customers around the world, and very extensible. This first post in a series about our platform explains how we're building systems to meet these demands using modern, open-source technology.

From the start, we've built our backend as a collection of distributed microservices. For an early-stage startup, this architecture is quite unusual; most companies start with a centralised application using well-known frameworks and relational databases.

But with an ambition to build the **best current account** and **business bank accounts** in the world and scale it to hundreds of millions of customers, we knew that we had to be the bank with the best technology. **Daily batch processes**, single points of failure, and maintenance windows are not acceptable in a world where customers expect 24x7, always-on access to their money, and where we want to roll out new features in a matter of hours, not months.

Large internet companies like Amazon, Netflix, and Twitter have shown that single monolithic codebases **do not scale** to large numbers of users, and crucially large numbers of developers. It follows that if we want to operate in many markets, each with unique requirements, we will someday need many teams, each working on different areas of the product. These teams need to control their own development, deployment, and scale – without having to co-ordinate their changes with other teams. In short, the development process needs to be distributed and decoupled, just like our software.

With only 3 backend developers at the start, we had to be pragmatic. We chose some familiar technologies which took us in the right direction and got on with building a product, knowing "future us" could always change things later.

We held a workshop to plot some improvements to our platform.

When we launched the Monzo **beta**, our backend had grown to nearly 100 services (now about 150), and our engineering team had grown considerably too. We knew our **banking licence was imminent**, and it felt like a good time to reconsider some of the architectural choices we'd made. We held an engineering workshop and identified a few areas of focus:

1. **Cluster management**

We need an efficient, automated way to manage large numbers of servers, distribute work efficiently among them, and react to machine failure.

2. **Polyglot services**

Go is well-suited for building microservices and will probably remain the dominant language at Monzo, but using it exclusively means we can't take advantage of tools written in other languages.

3. **RPC transport**

With large numbers of services distributed across hosts, data centres, and even continents, the success of our system depends on having a solid RPC layer that can route around component failure, minimise latency, and help us understand its runtime behaviour. This layer has to make our system stronger than its weakest link.

4. **Asynchronous messaging**

To make our backend performant and resilient, we use message queues to enqueue work to happen "in the background." These queues have to provide strong guarantees that work can always be enqueued and dequeued, and messages will never be lost.

Cluster management

If we want to build a fault-tolerant, elastically-scalable system, it follows that we need the ability to add and remove servers as hardware fails, and as user demand changes. Running only a single service per host is wasteful (the service may not need all the resources of the host), and the traditional approach of manually partitioning services among hosts is tedious and difficult to scale.

Cluster schedulers are designed to abstract applications away from the hardware on which they run. They use algorithms to schedule applications onto hosts according to their resource requirements, scale them up and down, and replace them when they fail. It's our ultimate goal to run our entire system on a scheduler, which means we want to deploy state*ful* as well as state*less* apps in this way.

Containerisation – and **Docker** in particular – is hard to ignore these days. Behind the hype, they are simply a **combination** of **technologies** in the Linux kernel coupled with an image format, but they provide a powerful abstraction. By packaging an application *and all its dependencies* into a single, standardised image, the application is decoupled from the operating system, and isolated from other applications on the same host. A cluster manager that deals exclusively with containers makes a lot of sense.

After a year or so of using Mesos and Marathon, we decided to switch to **Kubernetes**, which we run on a fleet of **CoreOS** machines. It is very tightly integrated with Docker, and is the product of Google's **long experience** running containers in production at global scale. Deploying Kubernetes in a **highly available** configuration on AWS is not for the faint of heart and requires you to get familiar with its internals, but we are very pleased with the results. We regularly kill hosts in production and Kubernetes quickly reschedules applications to accommodate. In fact, we may soon deploy something like Netflix's **Simian Army** to deliberately induce failure and ensure our systems tolerate it transparently.

Our production infrastructure costs about a quarter of what it did before we switched to Kubernetes

In addition to improving our resilience, Kubernetes has given us impressive cost savings. Our production infrastructure costs about a quarter of what it did before we switched to Kubernetes. To illustrate one example why, consider our build systems. Previously, we ran several very beefy Jenkins hosts dedicated to the task, which were inefficient and expensive. Now, build jobs run under Kubernetes, using spare capacity in our *existing* infrastructure, which is basically free. **Resource allocations and limits** ensure that resource-intensive but low priority jobs like these do not affect the performance of critical, user-facing services.

Polyglot services

Go makes it easy to build low-latency, high-concurrency servers, but no one language ecosystem has everything we need to build a bank. We need to be able to use a wide range open-source and commercial tools, hire developers with diverse skill-sets, and evolve as technologies change.

Docker allows us to easily package and deploy applications regardless of what language they are written in or what dependencies they have. Common practice advocates abstracting shared code into libraries, but in a polyglot system, this doesn't work. If we have to replicate and maintain thousands or even hundreds of thousands of lines of shared code each time we use a new language, this overhead quickly becomes unmanageable.

Instead, we encapsulate shared code into services in their own right. To obtain a distributed lock, a service can call a locking service which fronts **etcd** via "vanilla" RPC. We're considering building services like these to front all "shared infrastructure" (including databases and message queues) so we can reduce the required code for each language to just the client needed to perform RPC calls. This approach has already allowed us to build services in Java, Python, and Scala.

By making RPC the one common component, we can still get the benefits of code *reuse* without the **drawbacks of tight coupling** from code *sharing*. It actually becomes much easier to build tools which apply quotas, rate limiting, and access controls.

RPC

Given we want to be able to write services in many languages, whatever protocol we chose had to have good support in lots of languages. HTTP was the clear winner here: every language out there has an implementation in its standard library.

However, there is still a bunch of additional functionality you need in the RPC layer to build a robust microservices platform. To mention some of the more interesting ones:

- Load balancing: most HTTP client libraries can perform round-robin load **balancing based on DNS**, but this is quite a blunt instrument. Ideally, a load balancer will select the most appropriate host to minimise failure *and* latency – that is, even in the presence of failure and slow replicas, the system will remain up and fast.
- Automatic retries: in a distributed system, **failure is inevitable**. If a downstream service fails to process an **idempotent** request, it can be safely retried to a different replica, ensuring the system as a whole is resilient even in the presence of failing components.
- Connection pooling: opening a new connection for each request has a hugely **negative impact on latency**. Ideally, requests would be multiplexed onto pre-existing connections.
- Routing: it is very powerful to be able to change the runtime behaviour of the RPC system. For example, when we deploy a new version of a service, we might want to send a fraction of overall traffic to it to check that it behaves as we expect before ramping it up to 100%. This routing mechanism can also be used for internal testing: instead of having an entirely separate staging system, it becomes possible to stage individual services *in production* for certain users.

Looking around, **Finagle** was clearly the most sophisticated RPC system. It has all the functionality we want and more, and it has a very clean, modular architecture. Having been in production use at Twitter for several years, it has been put through its paces in one of the world's largest microservices deployments. As luck would have it, **linkerd** – an out-of-process proxy built on Finagle – was released earlier this year, meaning we could take advantage of all of these features without having to write our applications on the JVM.

Instead of talking directly to other servers, our services talk to a local copy of linkerd, which then routes the request to a downstream node chosen using a **Power of Two Choices + Peak EWMA** load balancer. When idempotent requests fail, they are automatically retried (within a **budget**). None of this complex logic has to be contained within individual services, meaning we're free to write our services in any language without having to maintain many RPC libraries.

We deploy linkerd as a **daemon set** in Kubernetes, so services always talk to a linkerd on localhost, which forwards the request. In some cases, an RPC will never actually traverse the network if replicas of both communicating services exist on a host. As we have rolled out linkerd we've **found and contributed fixes** for a few bugs in the Kubernetes integration.

Asynchronous messaging

Much of the work in our backend is asynchronous. For example, even though end-to-end payment processing takes less than a second, we want to respond to payment networks within tens of milliseconds to "approve" or "decline" a transaction on a Monzo card. The work of enriching merchant data, sending push notifications, and even inserting the transaction into the user's feed happens asynchronously.

Despite this asynchronicity, these steps are very important and must *never* be skipped. Even if an error happens which can't be automatically recovered, it must be possible for us to fix the issue and resume the process. This gave us several requirements for our async architecture:

- Highly available: publishers must be able to "fire-and-forget" messages to the queue, regardless of failed nodes or the state of downstream message consumers.
- Scalable: we must be able to add capacity to the message queue without interrupting the service and without upgrading hardware – the message queue must itself be horizontally scalable, just like the rest of our services.
- Persistent: if a node of the message queue fails, we must not lose data. Similarly, if a message consumer fails, it should be possible to redeliver the message and try again.
- Playback: it is very useful to be able to replay the message stream from a point in history, so that new processes can be run (and re-run) on older data.
- At least once delivery: all messages must be delivered to their consumers, and although it's **generally impossible** to deliver a message *exactly* once, the "normal" mode of operation should not be that messages will be delivered many times.

With all the other message queues out there, **Kafka** seemed like a natural fit for these requirements. Its architecture is very unusual for a message queue – it is actually a replicated **commit log**, but this makes it a good fit for our use-case. Its replicated, partitioned design means it can tolerate node failure and scale up and down without service interruption. Its consumer design is also interesting: where most systems maintain message queues for each consumer, Kafka consumers are simply "cursors" into the message log. This makes **pub/sub** systems much less expensive, and because messages are kept around for a while no matter what, we can play earlier events back to certain services easily.

I hope this post gave you a taste for how we think about backend architecture. I've barely scratched the surface of our platform and there's lots more to write about in future posts – for

example how we store data, how we manage infrastructure security, and how we mix cloud and physical hardware when we need to connect to things like payment providers over **leased lines**.