Engineering at Wealthsimple: Reinventing our Trading Platform for Scale

Every day we evaluate and trade thousands of accounts. The trading hours are limited, so it's all done in a very short period of time. What worked when we had 30 accounts no longer worked when we hit 3k. Just as now what worked for 3k accounts doesn't work at 30k accounts. Each time we've reached a new magnitude of scale, we've had to re-think our approach and build something that could scale to the next level.

We started redesigning our trading platform shortly after the previous version just made it out the door. Our continued growth in Canada, entry in the US market, and advisor offerings guided how we needed to engineer our trading platform for the present and future.

Simple Beginnings

Our very first trading platform at launch was really just a simple Google docs spreadsheet. Eventually the number of clients grew to around 2k and we developed what was a script, with a web interface. At the time, our business was fairly simple:

- Canadian market
- Few account types (TFSAs, RRSPs, non-registered accounts)
- Few target portfolios
- Fixed fee structures

At 2–5k accounts, the volume of trades is fairly low. And our solution was fairly unsophisticated, we like to keep things simple!

Over time, Wealthsimple evolved into a microservice architecture. Each service is responsible for some set of behaviours and actions on a key resource.

The Order Generator was one of these services. It calculated and created our orders. Every morning, data was pushed from our services to the Order Generator. This included the client's desired target asset mixes and current holdings. It was a black box, but the output was a set of orders. These orders were then sent to our brokerage.

The Order Generator was a multi-threaded Java microservice and it processed one account at a time moving through a chain of steps.

For the size of problem at the time, it worked quite well. We knew there would be problems in the horizon, notably:

Scalability. The application was multi-threaded, but it couldn't scale horizontally. It was a single process application and could only scale vertically, meaning each time we had larger loads we would need beefier hardware.

Regional Differences. Internationalization is a beast of its own when it comes to the financial industry; regulations vary dramatically across regions. What might make sense to a client in Canada makes little sense to a client in the U.S. and vice-versa. For example, to do Tax-Loss

Harvesting (TLH) in Canada, we only need to consider data from the last 30 days. In the U.S., we look at a full calendar year of data because of short-term vs longer-term capital gains tax rules. We also use different thresholds for when we determine whether it is worth it to harvest the loss.

Unforeseen Problems. It's one of those things that you can't get away from. Stuff happens. In our original system, if something happened, it happened for everyone. That is, we had no way of continuing to create orders for everyone else. We had to replay everything, see what went wrong, fix it and try again.

So what could we do to address some of these concerns and design something that we could scale out and continue to improve? The obvious solution was to just improve our current system: split work between servers, address the various regional and domain problems, and add better investigation capability.

We did something else...

Design for Scale

We began prototyping a **workflow engine**. A workflow engine is used to execute a business process modelled out in some descriptive language or with some editor. When you distill what a workflow engine is, it's really just a **finite state machine** with a historical record of every state change saved into a database.

Like most companies, Wealthsimple has a lot of business processes. Each one has a series of well-defined steps. Deposits, withdrawals, and transfers are a few of these. The order generation graph shown earlier is one as well. As a workflow executes each of these step, it accumulates data, storing it in an execution context. This allows proceeding steps to act on this data. This is exactly what the workflow engine is designed for.

Workflow

The workflow engine we created at Wealthsimple, uses JSON to define the workflow. This workflow definition is then saved in a database. Each node in the workflow can be a call out to a service, a wait state, or a terminal state.

Suppose we created this "dating" workflow:

We would then use the following json to define it:

The **workflow** defines that it needs some initial data and a start state. This initial data is passed in when an instance of the workflow is created.

Each node that needs to get data from a service or do some action references a service name and the name of the workflow action available on that service. Each node must also specify what data to send to the service, available from the initial seed data or from some data collected along the way at a previous step.

When the service is called, it will respond with any type of transition, which we then map to our desired next step.

We also support a wait state in this example. The workflow engine will evaluate a bit of ruby code to determine a time for when the instance of the workflow will be next eligible to be executed. This code can also reference any of the data contained within the initial context or data collected thus far. We expect that this code will return a ruby Date or DateTime object. These objects have the benefit of being timezone aware.

Once a workflow instance is executing, it creates several ***activity records***. At each step of the workflow this activity record is created to keep a historical view of data sent and received. The metadata for the workflow instance keeps track of what current step the workflow instance is in.

The task of *running* these workflow instances falls onto a workflow worker within the workflow engine. Each worker will try to acquire an eligible task from the database to work on.

Portfolio Manager

When we first start building out our microservices we began by adding (in some cases duplicating) APIs to each of the services. All these services already had REST APIs, but that approach will not work for the workflow engine. Rather than just serving data, some of these APIs had to make decisions about the work performed.

A workflow itself is pretty "dumb". It does not make decisions. It simply collects data and moves the pointer from one state to the next, based on a response's instruction. We left some of this decision making to a new service we named Portfolio Manager. It takes place of the *Order Generator*in the previous architecture diagram. It really doesn't do much aside from starting work at the beginning of the day, and providing an endpoint that a workflow can dump all its final data into. It also provides the ability for our portfolio managers to review and submit any orders.

What does this gain us from a reliability aspect?

Let's look at what the workflow itself offers:

- Each step can be resumed
- Each step can be forcefully paused
- Each step can be inspected
- No single workflow instance will prevent another one from making progress
- Can be scaled both via threads and additional instances

Doing the work, is now a shared responsibility across our microservices. The biggest benefit to this is that each individual service can now be scaled independently. Based on the type of work the service primarily handles, we can assign different machine types and scale them horizontally to meet the demand. For example, fetching the positions is more IO bound, but our order calculation step is more CPU bound as it only does computation.

Services can now be can be written in any language or technology of a team's choosing. In our case, we moved our order calculation to a python service that allows our data analysts to utilize SciPy and NumPy.

We no longer have to encode and push data to the Order Generation service. We can now use up-to-date information by reaching out for this data. This was an easy win, and something we could not have done previously.

Starting versions of our trading workflows with tweaked parameters or adding extra steps can now be accomplished without having to make code changes or needing to do a deployment. Similarly, if we are testing something, we can simply create another version of a workflow to play with without affecting our production pipeline.

We've also began creating other workflows that re-use many of the same endpoints we created for the order generation, removing a lot of potential duplication but also to simulate new business processes.

Future Plans

Just as before, even before we finished launching the new platform we were already planning what we can do next to improve the new system.

Workflow framework

Because the workflow engine is "dumb," it needs the external service to tell it what transition to take next ("making a decision"). Deployment of an external service any time you update or add a new step means the external service is no longer agnostic to the existence of the workflow engine.

This external service is now strongly coupled as it must implement an endpoint that conforms to the API style of the workflow engine. This is more of a concern if that endpoint holds logic specific to a defined workflow as it can no longer be reused.

One of our attempts at addressing this used Amazon's Lambda Service. Although we found it highly parallel, its performance was quite poor and it often timed out before the work was finished. On top of it, you still needed to set up a deployment pipeline. A single dedicated t2.small instance, one of the lower end virtual machines Amazon offers, if executing the same code was more responsive and performant. Neither really stood out as an ideal solution, but for now we still stuck with a dedicated service for a few things.

Improving Workers

Currently, each workflow worker runs in its own thread and will query the database for tasks to work on independently of other workers. Occasionally, there are collisions and they need to retry to find another task. To improve performance, we're planning on adding a broker that can quickly provide each worker with tasks.

Postgres?

We're not sure if using Postgres for the workflow engine was the best idea. It didn't take long at all before we had millions of activity records for every single thing we did. We also had a ton of workflow instances (after all every day we trade upwards of 20k accounts). We found no one really used the data from the workflow engine after successful executions, and definitely not in any relationship modelling. Our database performance really started to go down hill as our engine tried to find work to do among all the dead entries.

We are looking at alternatives. Can we just reduce our data use? Do we need to record all the data sent, or just responses? Should we be looking at a key-value storage system? Perhaps Redis or Cassandra?

For now, we've modified our workflow engine so that after a couple of weeks it removes finished workflow executions from the database. We archive everything in a S3 bucket *just* in case.

Summary

We managed to scale our software to handle a lot more work. We made it distributed and massively parallel. We made it more transparent for investigative purposes and it's also highly customizable. This gave us huge gains.