Doing the Hard Thing

and other career stories

@qcmaude // Figma // October 29, 2024

Hi 👋

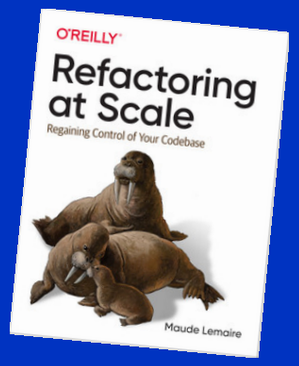
I’m Maude.

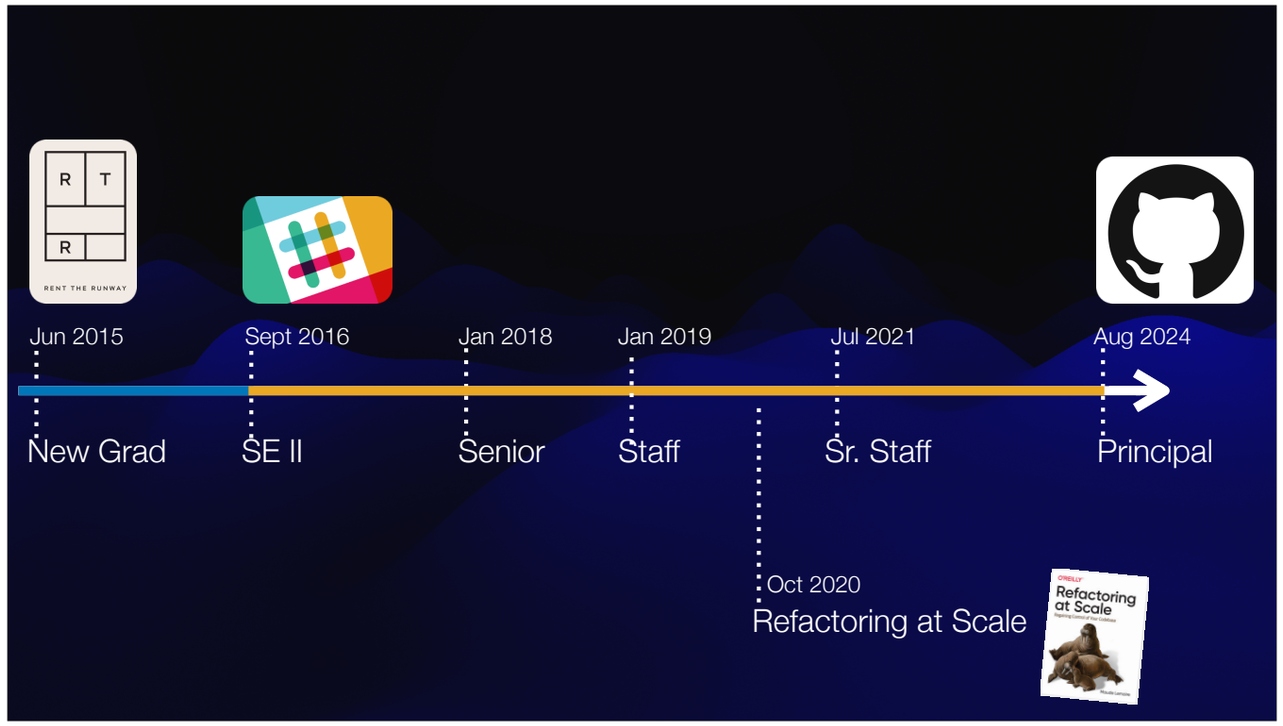
I’m a principal engineer at GitHub.

O’Reilly author.

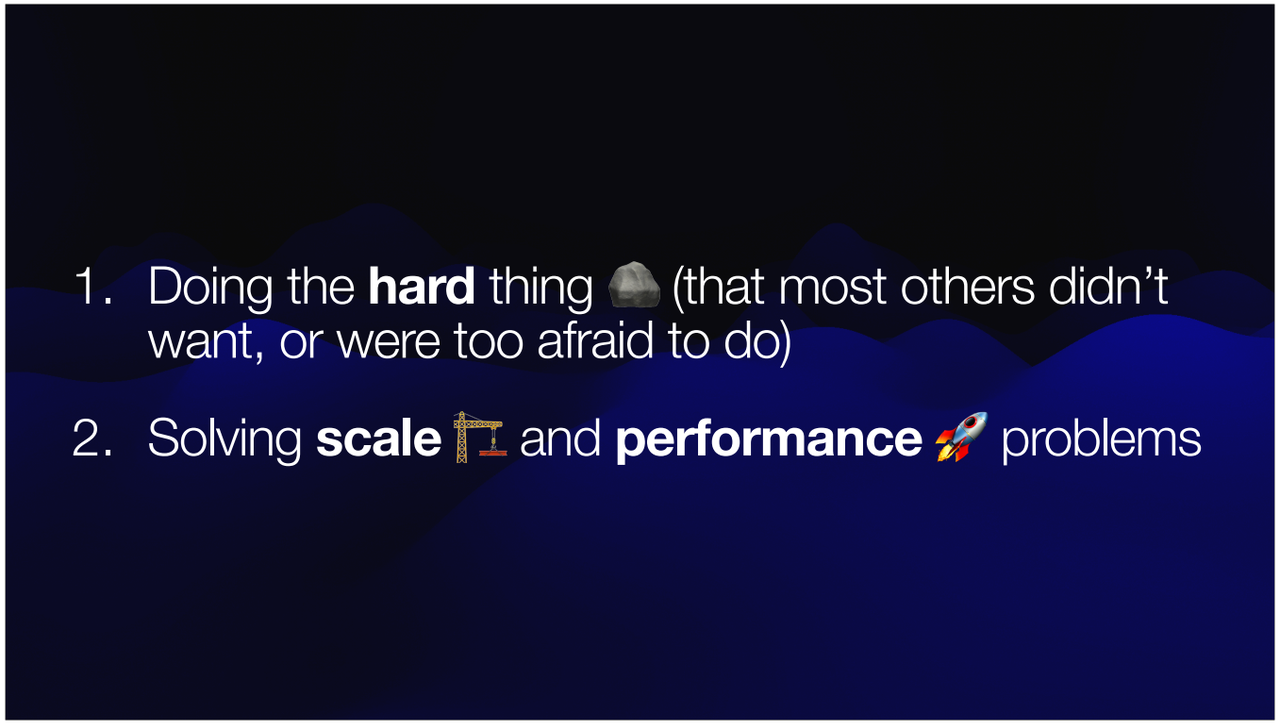
Mom to Henri (3yo) and pregnant with #2 🤰.

* married to @amorin for 8 years



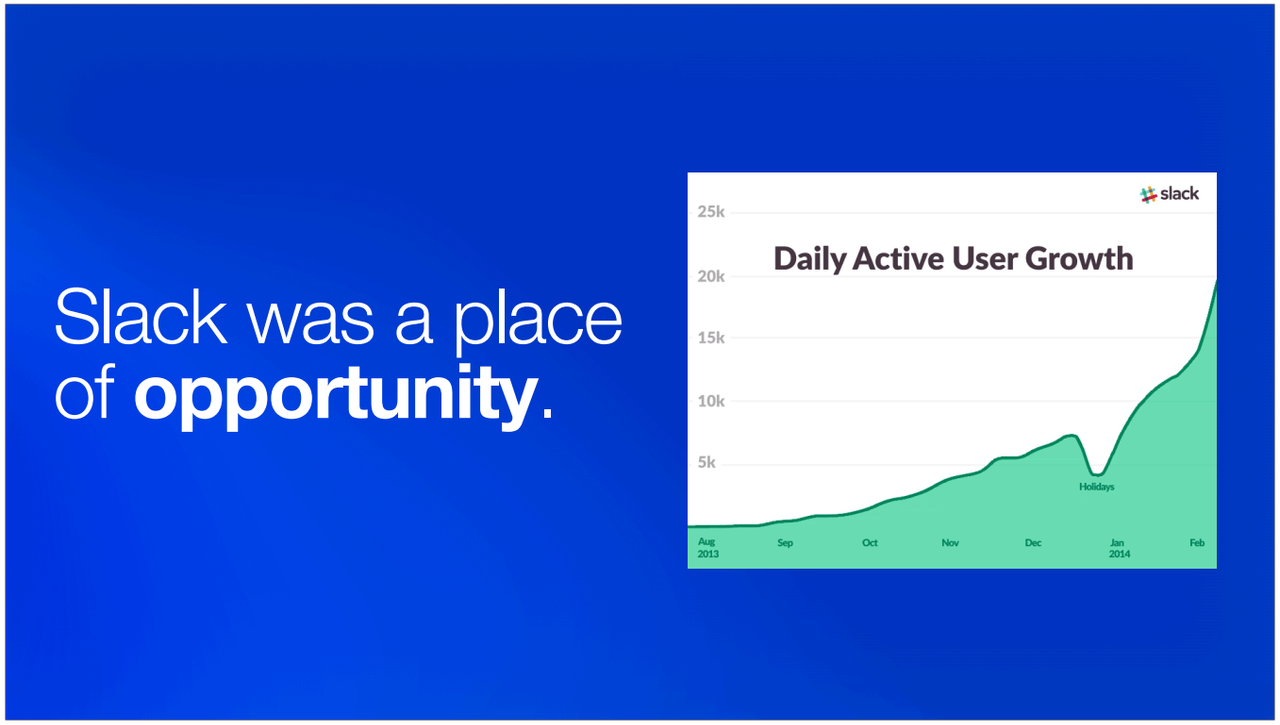

R T
R
RENT THE RUNWAY
Jun 2015
Sept 2016
Jan 2018
Jan 2019
Jul 2021
Aug 2024
New Grad
SE II
Senior
Staff
Sr. Staff
Principal
Oct 2020
Refactoring at Scale
Refactoring at Scale

In hindsight, there are **2 pervasive themes** in my career.



Elisa and I were chatting a bit via email to coordinate this session, and she was eager for someone to come and talk about their career journey from a more technical/ story-telling perspective, so I'm going to attempt to do just that.

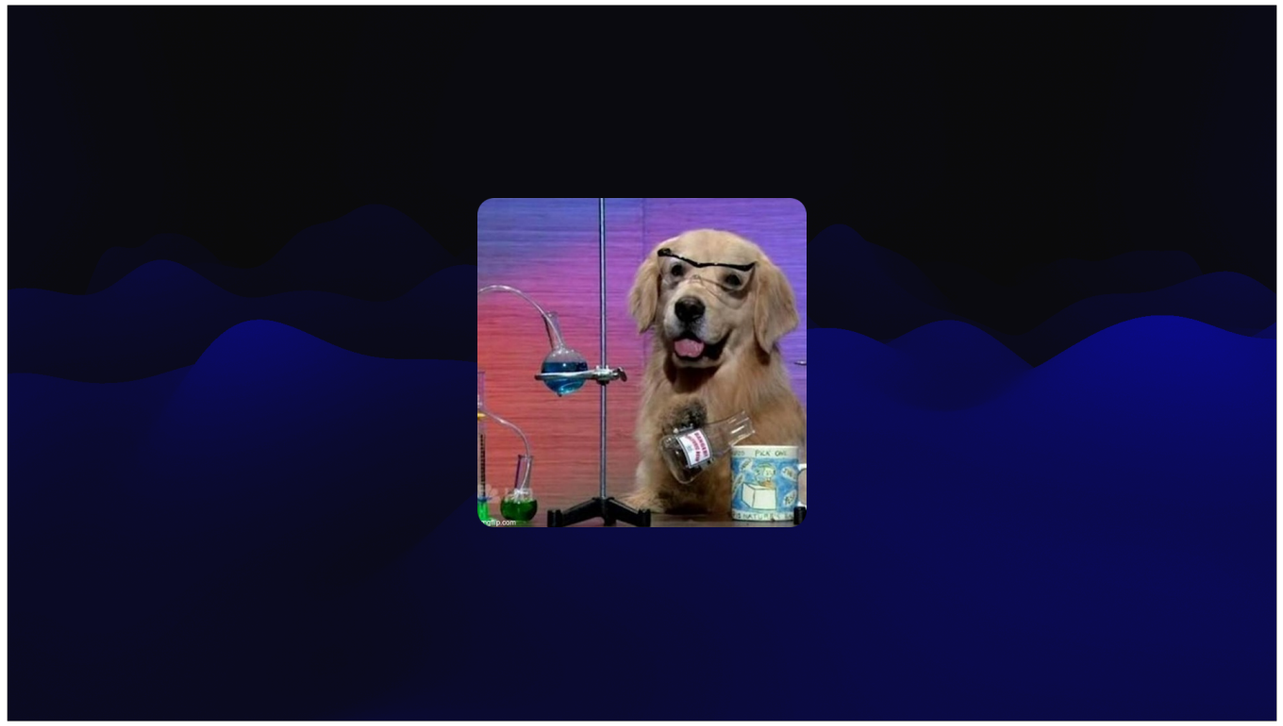
While I did some of these things during my (relatively) short tenure at Rent the Runway, it wasn't until I joined Slack that I really got my footing.



Here’s the thing with Slack.

When I joined in October 2016, there were maybe ~120 engineers. The product was *taking off*.
We had this many people to execute on a disproportionately large problem space. There were *so many opportunities* for everyone, regardless of level.

If you’d gotten through the door, folks trusted your ability to figure things out and have an impact.



My onboarding buddy, Moishe, was hard at work addressing performance problems for IBM (our largest beta customer) ahead of GA in January. He tossed me a few perf-related JIRA tickets to juggle alongside my feature work to help me get better acquainted with the broad service area covered by our team. I'd never written a single line of PHP or SQL before, and often found myself wondering whether I'd bitten off more than I could chew in deciding to move into a backend role.

Fortunately, no one seemed to notice that I had no idea what I was doing. In fact, I'd feigned enough confidence that when we spun up a team focused entirely on handling Enterprise-related performance problems, I made the cut. We had no shortage of problems to solve: everything was on fire.

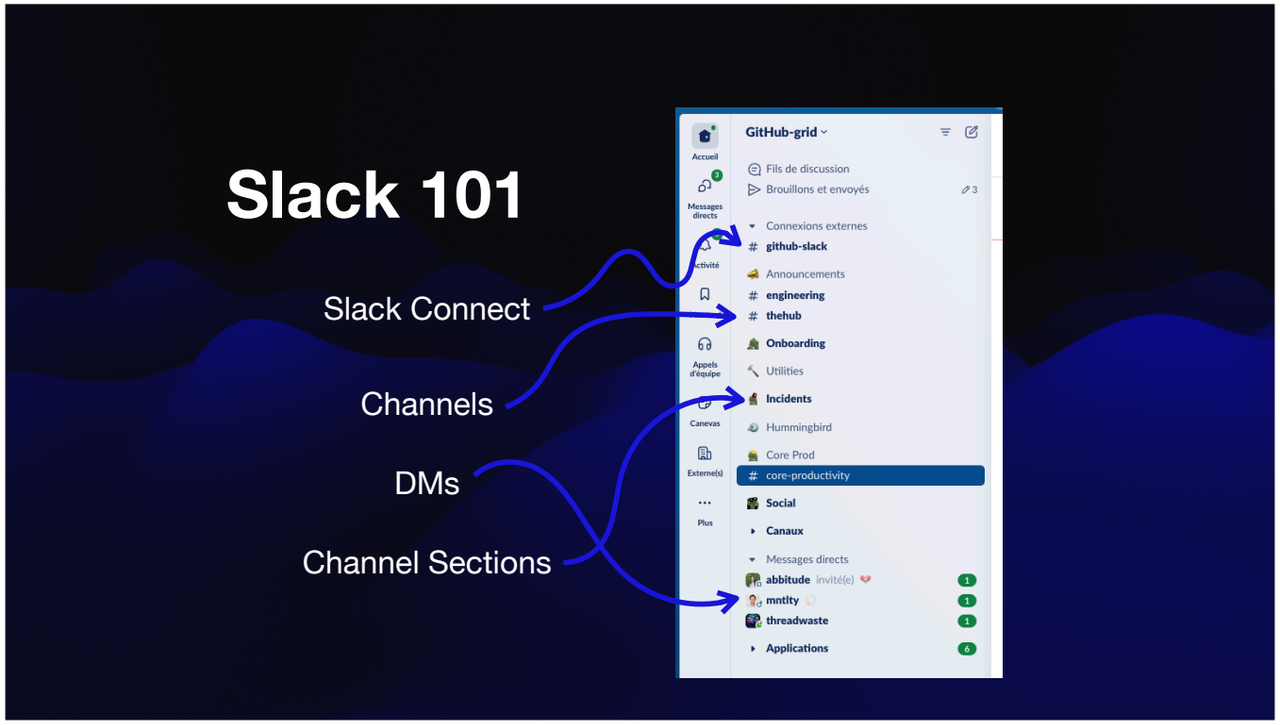
The Great Channel Membership Consolidation™

It’s April 2017. Our biggest problem at the time was IBM. They were our biggest customer in terms of daily active users (and weekly active users, and probably monthly active users, too).

They had a large east coast contingent, so on Monday morning when everyone logged into Slack at around 9 am, and everything inevitably went to – cough pile of poo – our mighty performance team had to be ready to go to tackle whatever the latest and greatest bottleneck turned out to be.

We quickly started seeing some patterns in the types of problems these customers were experiencing over and over.

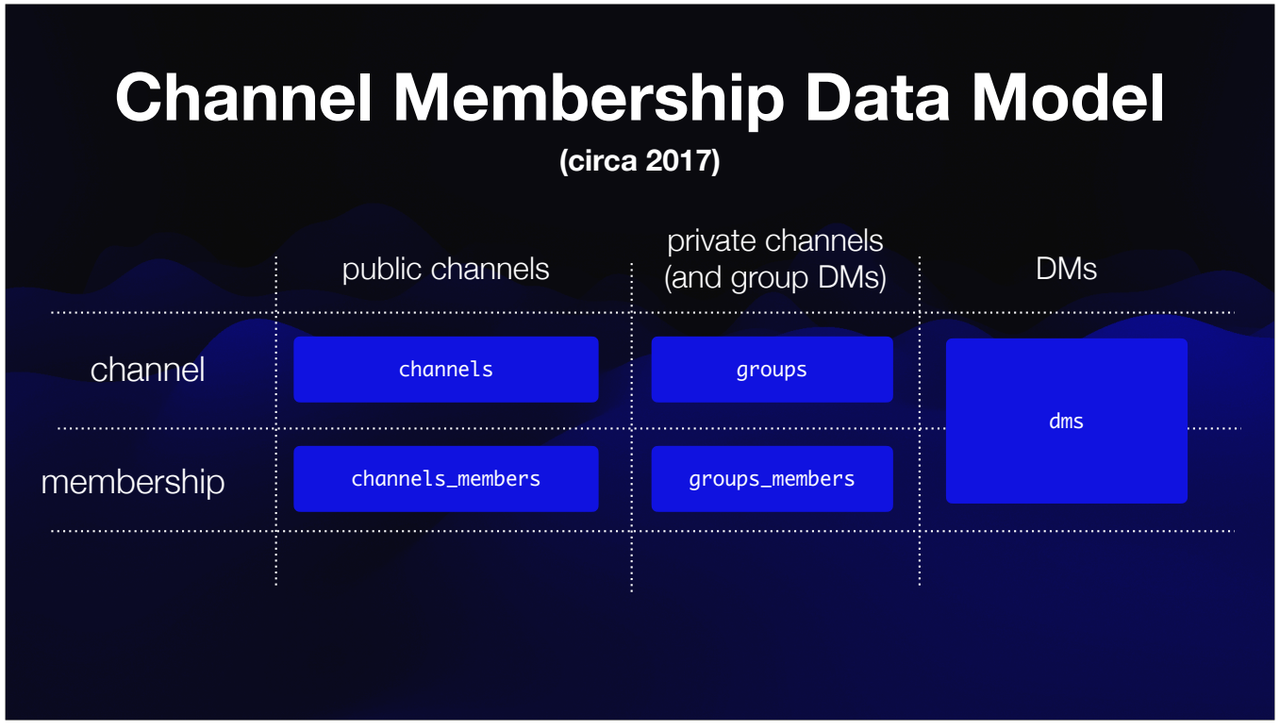
Here’s a little bit about one of those problems.



On startup, for every client, the client attempts to fetch all of your channel memberships: your public channels, your private channels, your group DMs, your DMs, everything. Slack uses this information to populate your channel sidebar, so you had someplace reasonable to land.

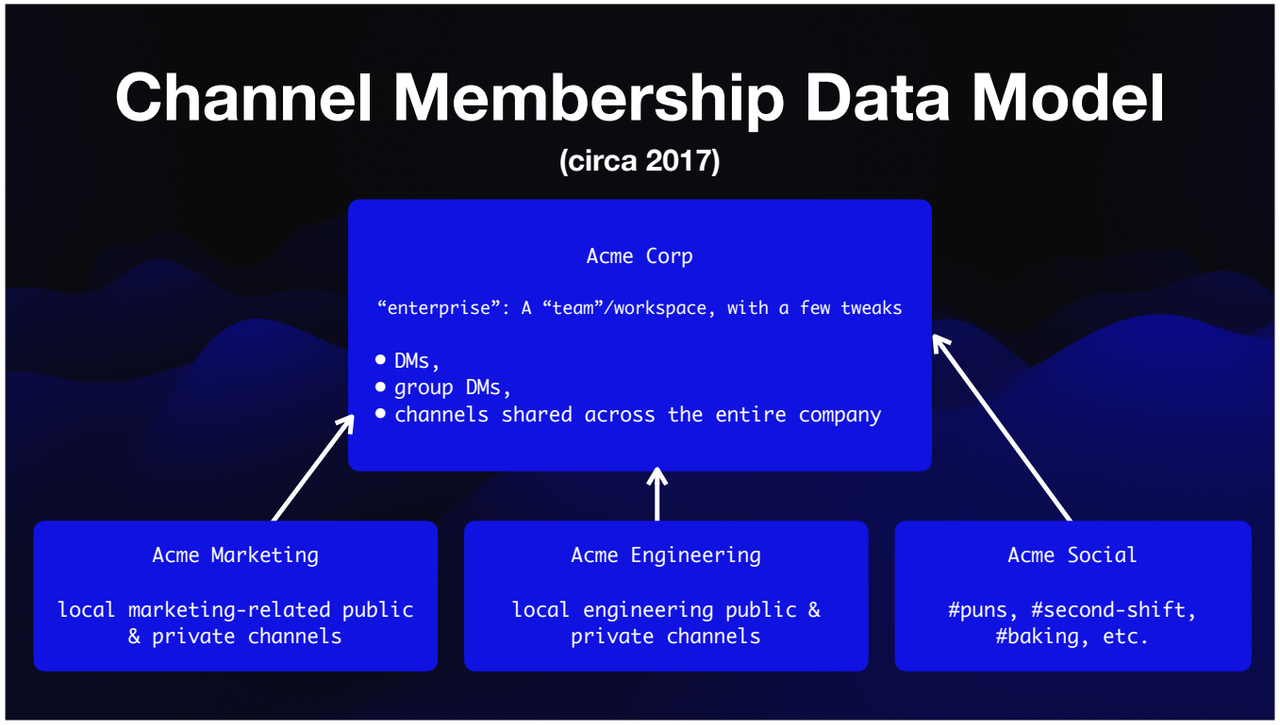
Separately, the server also has to determine which of your channels have unread messages, and calculate your mentions (that's the badge icon). It's a *whole lot* of up-front work.

By and large, the kind of logic that has to be calculated and delivered on boot hasn't changed much in the last ~7 years, but how it's done under the hood has undergone a complete overhaul.



So why was this a particularly thorny problem for IBM?

In order to calculate a user's channel membership, we had to combine information from three tables (and, in IBM's case, two distinct database shards.)



In order to calculate a user’s channel membership, we had to combine information from three tables (and, in IBM’s case, two distinct database shards.)

At the time, Slack organized its data by *team* (roughly a customer). In order to ship Enterprise as quickly as possible, they made a few shortcuts. One of those was to make an enterprise just another “team”, with sub-teams pointing to it.

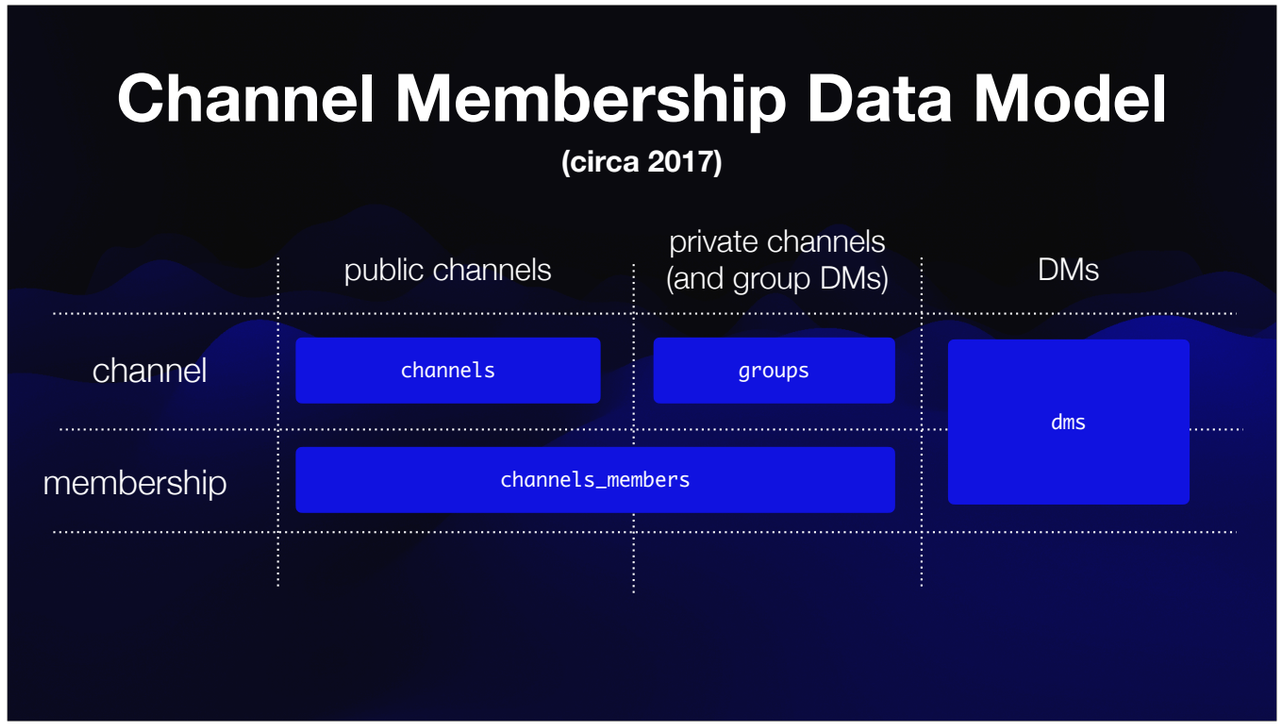
What that meant from an architectural perspective, was that we had one shard that housed data that was accessible across *all* teams in an organization, and data for the member teams spread out across other shards.

Actual Code from 2017

```
$sql = "SELECT m1.channel_id, 'C' AS type, status, archive_date, delete_date
        FROM channels AS c
        JOIN members AS m1 ON
                m1.channel_id=c.id AND
                m1.team_id=%team_id AND
                m1.user_id=%user_id AND
                m1.delete_date=0
        WHERE
                c.team_id=%team_id AND
                m1.type=%public_chan_type AND
                m1.join_date IN (%list:join_date)
        UNION ALL
        SELECT m2.channel_id, 'G' AS type, '0' AS status, g.archive_date, g.delete_date
        FROM groups AS g
        JOIN members AS m2 ON
                m2.channel_id=g.id AND
                m2.team_id=%team_id AND
                m2.user_id=%user_id AND
                m2.delete_date=0
        WHERE
                g.team_id=%team_id AND
                m2.type=%private_chan_type AND
                m2.join_date=%private_chan_privacy";

return db_fetch_team($team, $sql, array(
        'team_id'          =>  $team['id'],
        'user_id'          =>  $user['id'],
        'public_chan_type'     =>  channel_type_t::C,
        'public_chan_privacy'  =>  [channel_privacy_type_t::PUBLIC, channel_privacy_type_t::PRIVATE],
        'private_chan_type'    =>  channel_type_t::G,
        'private_chan_privacy' =>  channel_privacy_type_t::PRIVATE,
));
```

If you were booting up your client for Acme Engineering, we had to run queries like this for both the shard associated with Acme Engineering *and* Acme Corp at a top-level. Oh, and we had to add DMs because those aren’t being considered in this query.



So this was our plan: combine the membership tables into a single table for more efficient querying. No more JOINs, UNIONs, whatever.

Unfortunately, in order to migrate these queries, we first had to do an audit of every single location in the code where we were querying the original tables.

Actual Code from 2017

```
$sql = "SELECT m1.channel_id, 'C' AS type, status, archive_date, delete_date
        FROM channels AS c
        JOIN members AS m1 ON
                m1.channel_id=c.id AND
                m1.team_id=%team_id AND
                m1.user_id=%user_id AND
                m1.delete_date=0
        WHERE
                c.team_id=%team_id AND
                m1.type=%public_chan_type AND
                m1.join_date IN (%list:join_date)
        UNION ALL
        SELECT m2.channel_id, 'G' AS type, '0' AS status, g.archive_date, g.delete_date
        FROM groups AS g
        JOIN members AS m2 ON
                m2.channel_id=g.id AND
                m2.team_id=%team_id AND
                m2.user_id=%user_id AND
                m2.delete_date=0
        WHERE
                g.team_id=%team_id AND
                m2.type=%private_chan_type AND
                m2.join_date=%private_chan_privacy";

return db_fetch_team($team, $sql, array(
        'team_id'          =>  $team['id'],
        'user_id'          =>  $user['id'],
        'public_chan_type'     =>  channel_type_t::C,
        'public_chan_privacy'  =>  [channel_privacy_type_t::PUBLIC, channel_privacy_type_t::PRIVATE],
        'private_chan_type'    =>  channel_type_t::G,
        'private_chan_privacy' =>  channel_privacy_type_t::PRIVATE,
));
```

Remember this?

There were about 400 places (from a rough audit) that called into different kinds of channel membership tables. most of these SQL queries were embedded in old, crufty code from the early days of Slack– seriously, some of the git blames were the cofounder & CTO.

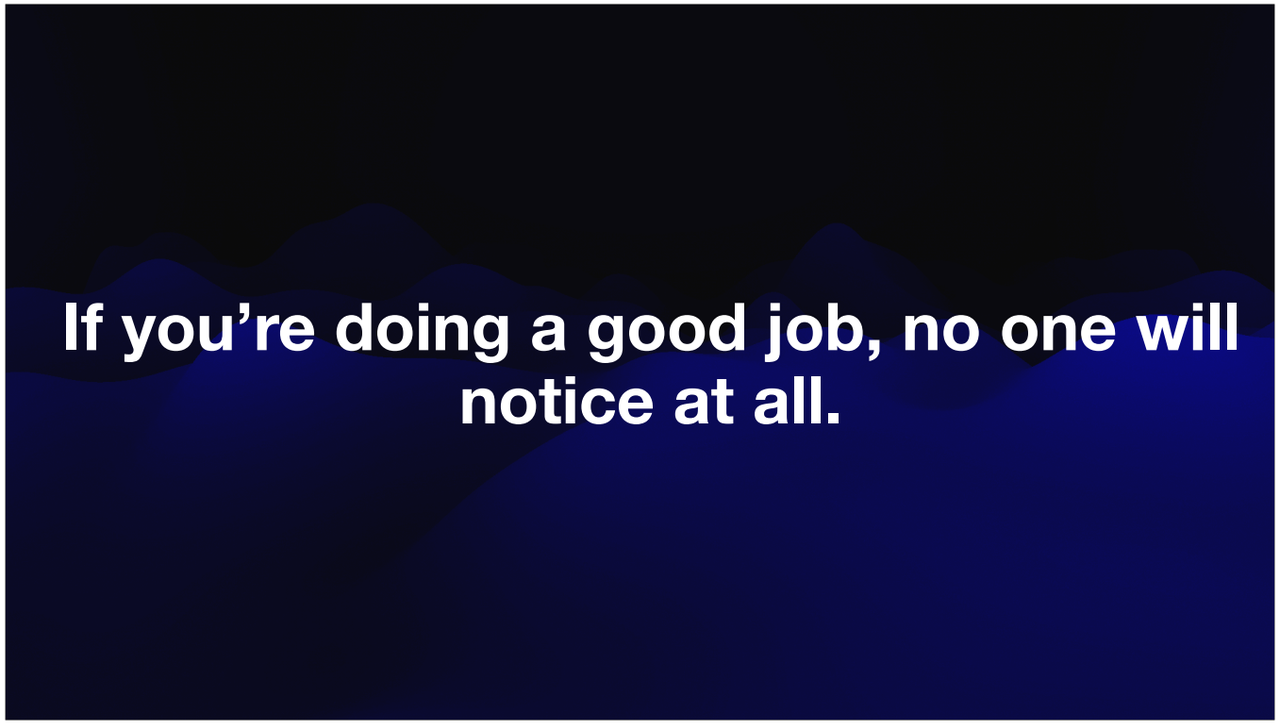
And, as expected, there was little to no unit test coverage.

This wasn’t going to be a small lift. We were going to have to alleviate some of leadership’s fears around undertaking something like this: particularly because it meant migrating data core to the Slack client.

But also, no one wanted to do it. No one wanted to touch channel membership with a 10 ft pole.

Here's a few points I brought up to our leadership:

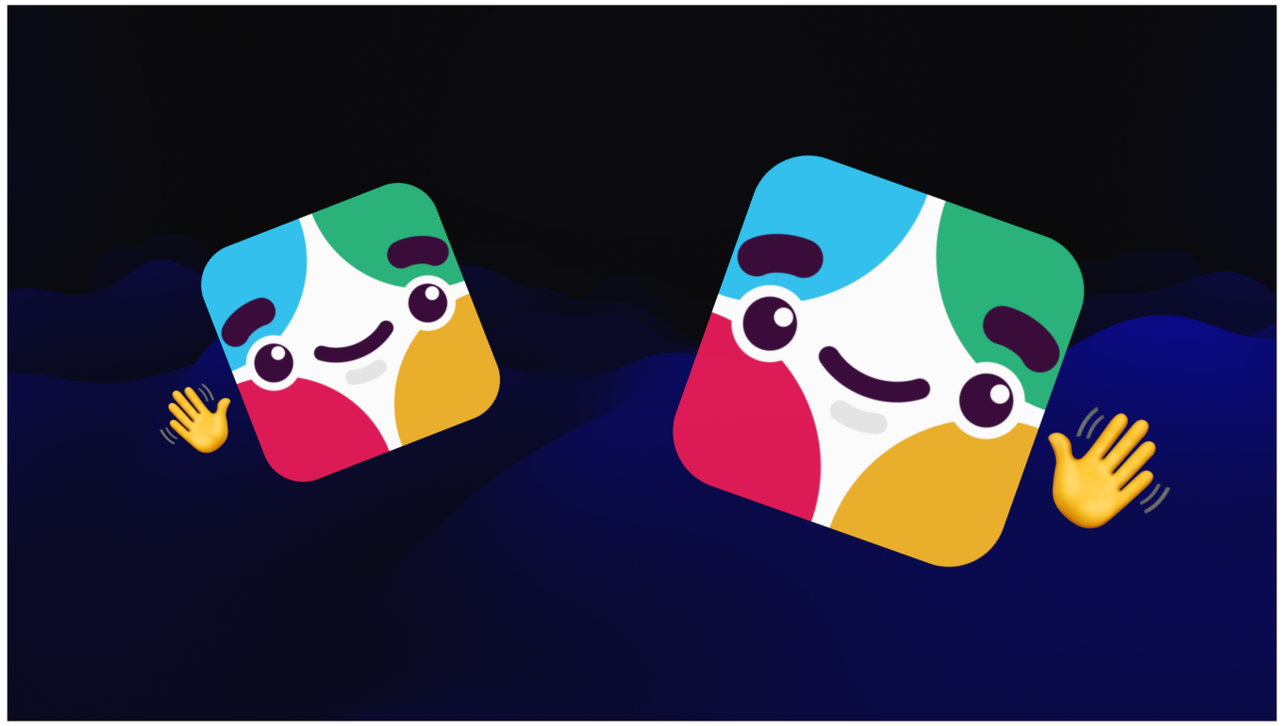
1. If we do nothing, we won't be able to log these customers in at all in a few months. There's only so much money you can throw at AWS before you have to take a hard look in the mirror and start mitigating problems at "home". We were at a critical point where we had enough of a runway to concentrate some effort on consolidating the tables, but were also incredibly close to staring down the the barrel of these customers' guns.
2. I also pointed out that while we're making changes to these tables, we'll have the opportunity to consolidate all of our callsites into a single file – and establish a prototype pattern for accessing data through a single file. I'll talk about how this played an important role later.
3. We have the opportunity to determine a pattern for doing data consolidations in the future; in this specific case, we know we may need to do something similar down the line to consolidate channel tables. More generally, it's likely this won't be the only time we have to move data around.



Refactoring work isn’t super visible. In fact, if you’re doing it well, no one should notice your work all!

But you *can* make that work visible. You can show the value that you’ve brought to company by keeping track of everything you’ve improved. I didn’t do nearly enough of this when consolidating membership, but I came to realize how important it was when I began telling other engineers around the org about the weird, unexpected things we found (and how we addressed them). I started keeping a laundry list of every bug we fixed, and query we improved when starting to consolidate channels.

We found all sorts of problems when combing through all of these callsites – some queries that weren’t using indexes, direct messages between Slackbot and Slackbot …



… of which we found 9 million.

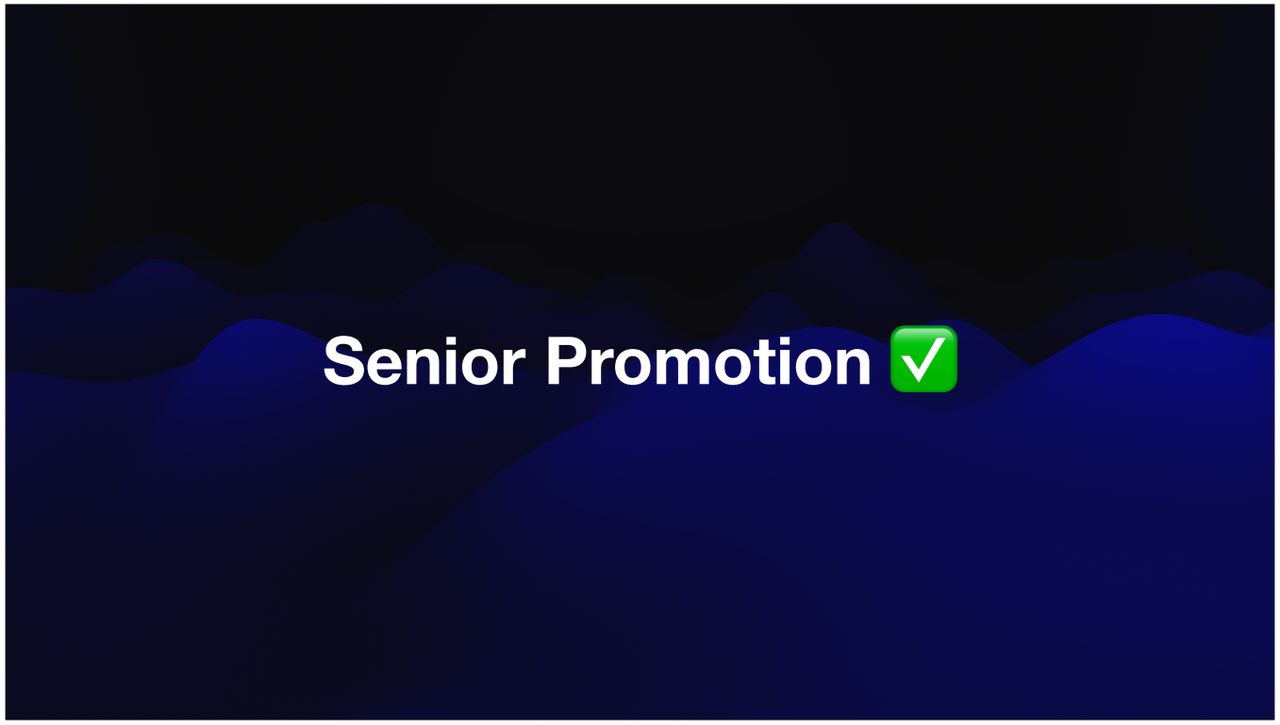
But more seriously, one of those wins was being able to introduce soft channel membership deletion. At the time of the membership consolidation, when you left a channel, you **properly left** that channel.

We issued a hard delete to your row on that table and then the only history we had of you ever being in there was either any messages you might've sent or, for two weeks while we keep backups alive, your row on the backup.

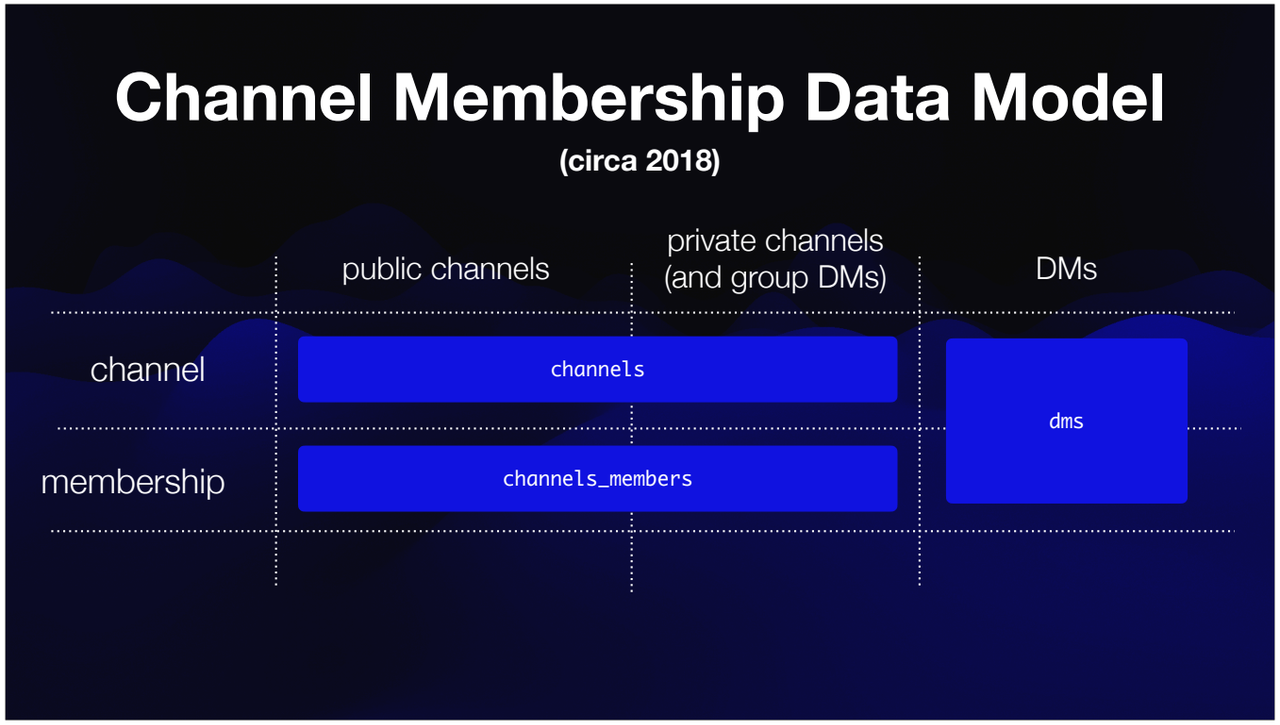
It was incredibly difficult for us to recover from mass user deletions (which unfortunately happened quite often – you might be surprised how easy it is for an admin to hit the wrong button in their identity provider).

When we deleted a user, we also deleted their channel membership (and the integrations they've installed and … a whole lot of things.)

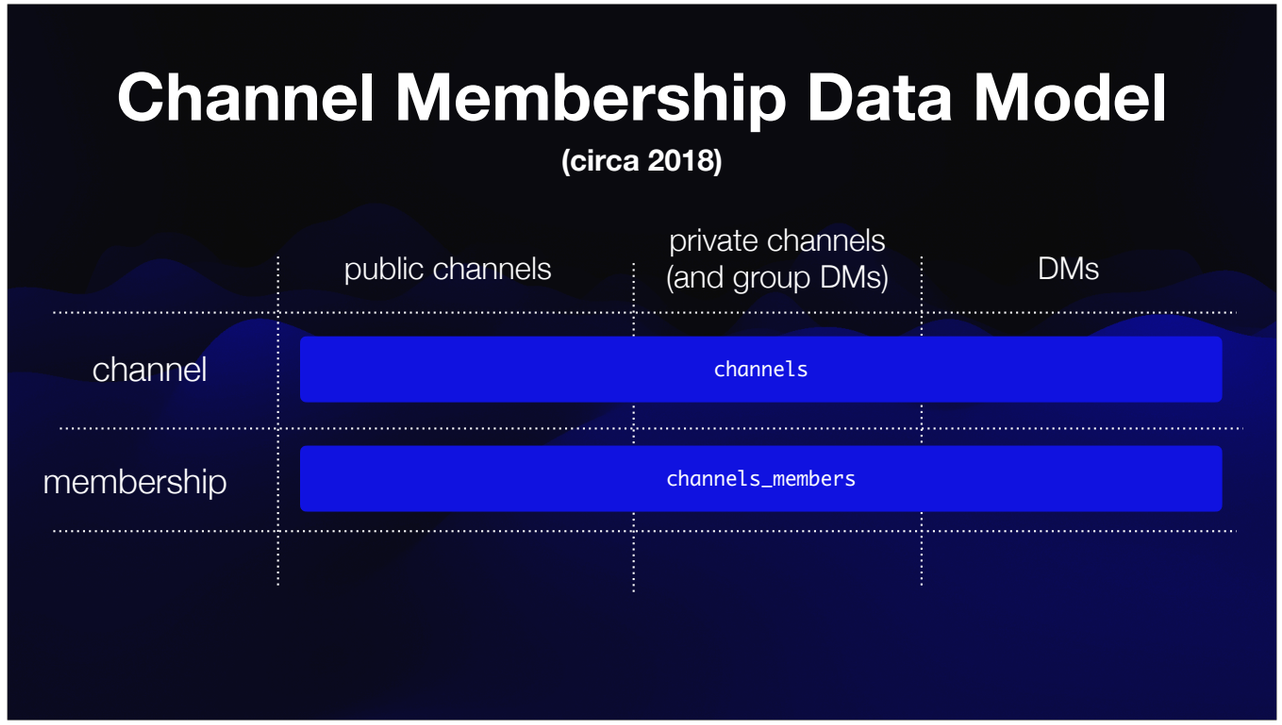
Now with all of the queries that dealt with channel membership in a single, well-tested, well-organized file, stored on a *single* table we could easily just add the new column and call it a day!



I eventually became *the* channel membership expert, which was a pretty good position to be in given how much of Slack relied on the concept.



I continued to build on the momentum, and later consolidated channels-related tables, and merged the hybrid channel + channel membership DMs table into the rest of the newly-migrated data.



I continued to build on the momentum, and later consolidated channels-related tables, and merged the hybrid channel + channel membership DMs table into the rest of the newly-migrated data.

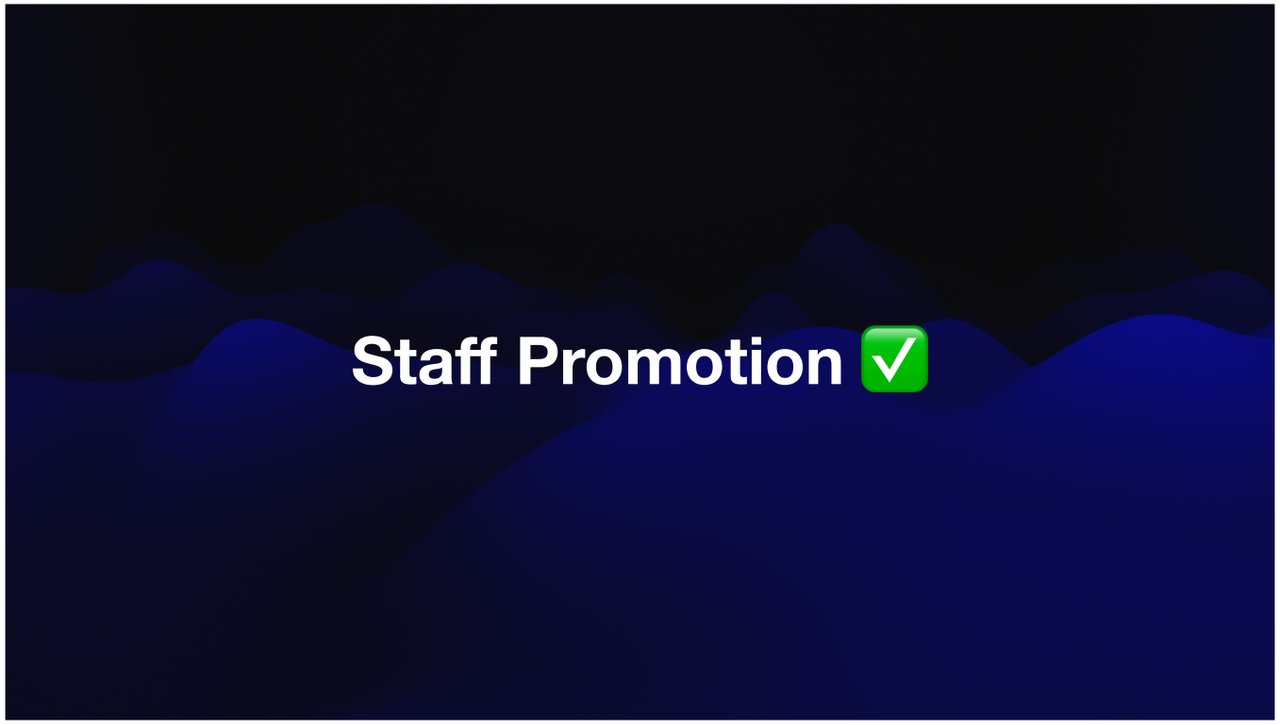
In the space of just two years, I'd gone from someone who'd never written a single line of SQL, to someone people sought out to help debug their queries. Something that had once seemed impossible was somehow now second nature to me.

After shipping migration after migration, with fewer and fewer bugs, I was eventually tasked, alongside the great @zmagg to …

Figure out how to migrate channel membership to Vitess.

This was an even *bigger* undertaking, and Maggie and I wrote about it in Chapter 12 of my book. I'll skip the details of this project because (1) you should totally ask Maggie about it and (2) I only have a half hour and really want to talk about the second half of my tenure at Slack, during which I led our load testing efforts.

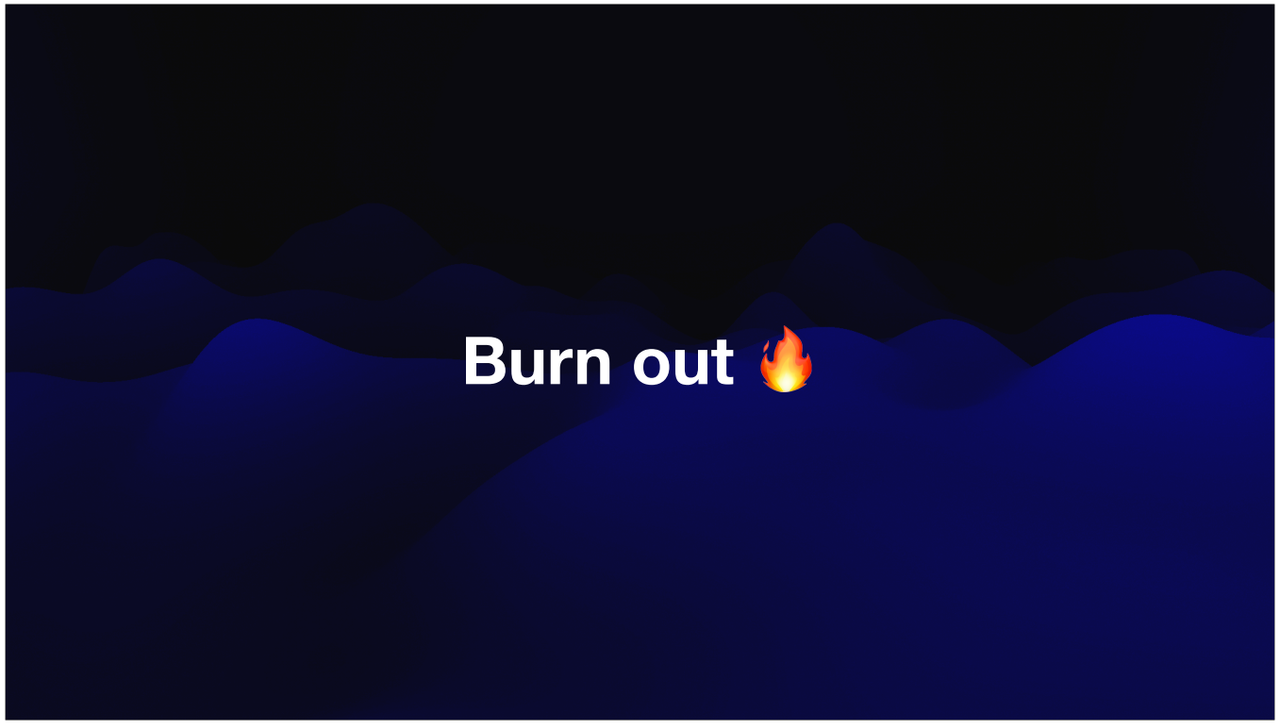
It was right around the time that the migration shipped that I got promoted to Staff.



It was right around the time that the migration shipped that I got promoted to Staff.

Backend Performance Infrastructure 🚀

The next big leap forward in my career happened when I managed to convince my director to fund a brand new team.

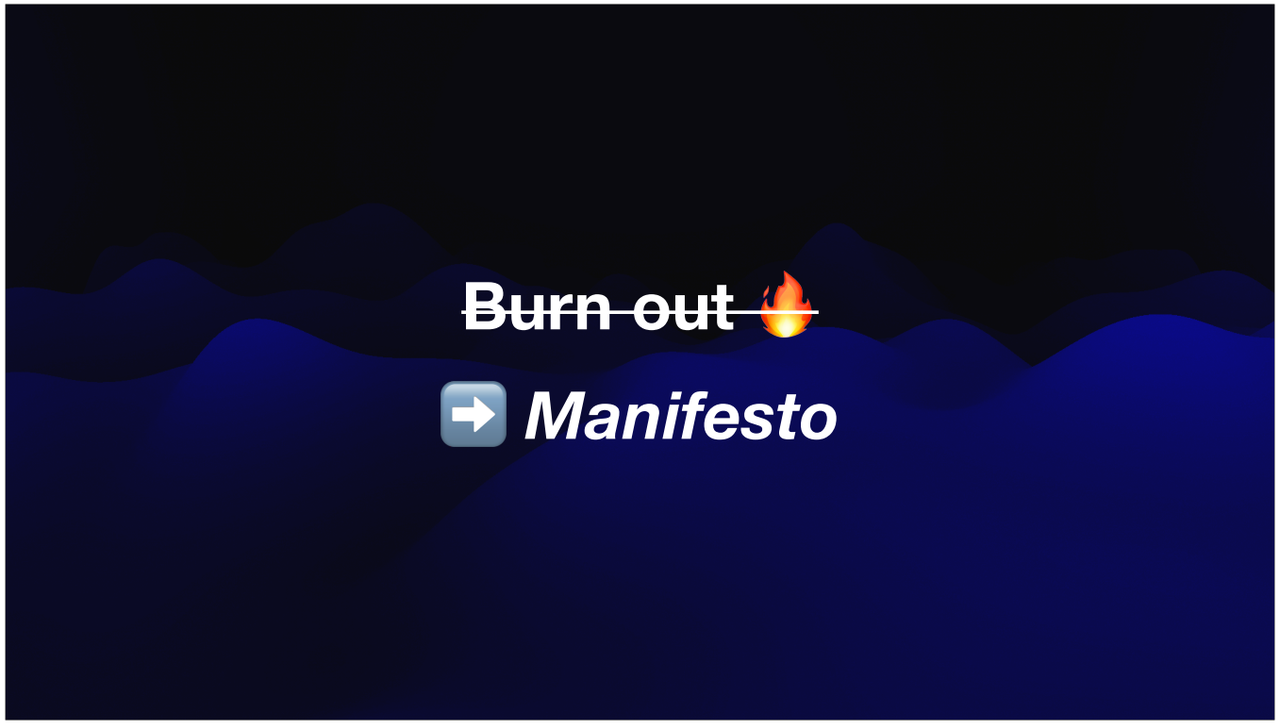


At the time, I was feeling pretty burnt out. This was about a year after the Vitess migration wrapped up, and my team was starting to tackle larger refactors within Slack's monolith, establishing stronger boundaries and detangling the spaghetti we'd spent years spinning.

I was itching to work on performance and scale problems again. The company had just gone public about 6 months prior, and we were continuing to ship features that failed to scale for our biggest customers. I thought that was embarrassing.

So, I told my manager. I talked her *ear* off about how we weren't setting great expectations with our most important customers, and how I wanted engineers to understand the impact of their changes. She encouraged me to write everything down.

Again, I was advocating we work on something that no one else wanted to work on.



I didn’t expect much when I sent her my manifesto, but she was convinced that it was compelling, complete enough vision to fund a team.

So, in February 2020, Backend Performance Infrastructure came to life with an initial headcount of just one– me!

Our first task was to take all of the scrappy, one-off load testing solutions that Slack had developed over the years, and build a robust, fully-supported offering that could be used by any engineer at the company to ensure their feature would work seamlessly for our biggest customers.

I started crafting a plan.

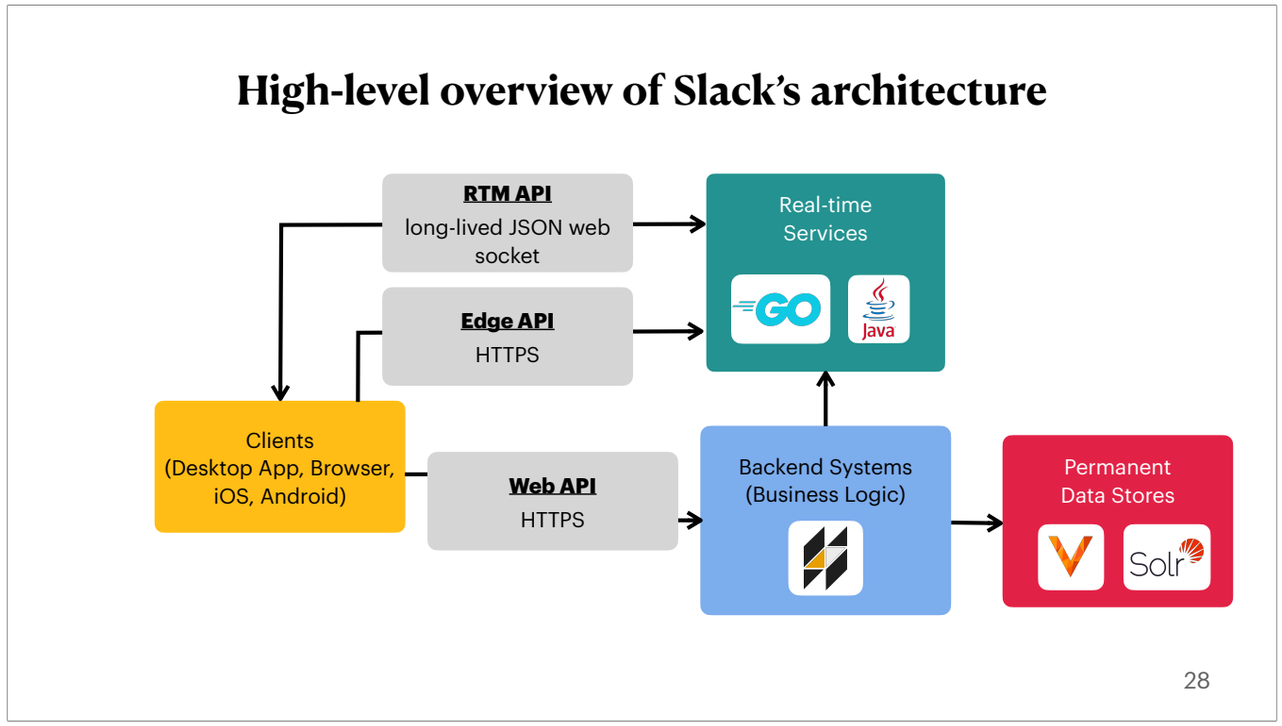
The team got its second engineer one early May. We were just getting into the early prototyping stages when the company signed its biggest ever contract– with Amazon.

They anticipated onboarding 500k users to Slack within 18 months. At the time, our biggest customer, IBM, had about 280k daily active users at their peak.

But within just 2 weeks of having signed the contract, they were asking us to whether we could support their 500k users in 10 weeks. (That's 88% sooner than we'd anticipated.)

10 weeks. 18 months down to 10 weeks.

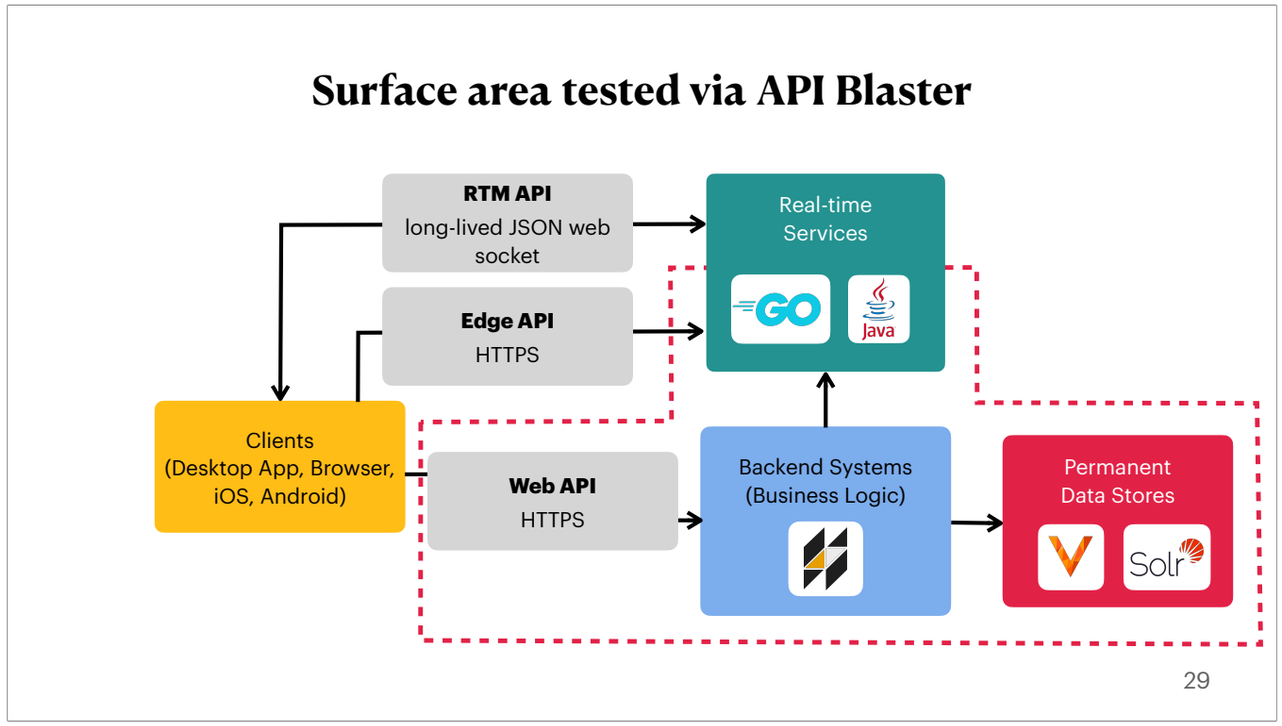
The load testing tool needed to be built *yesterday.*



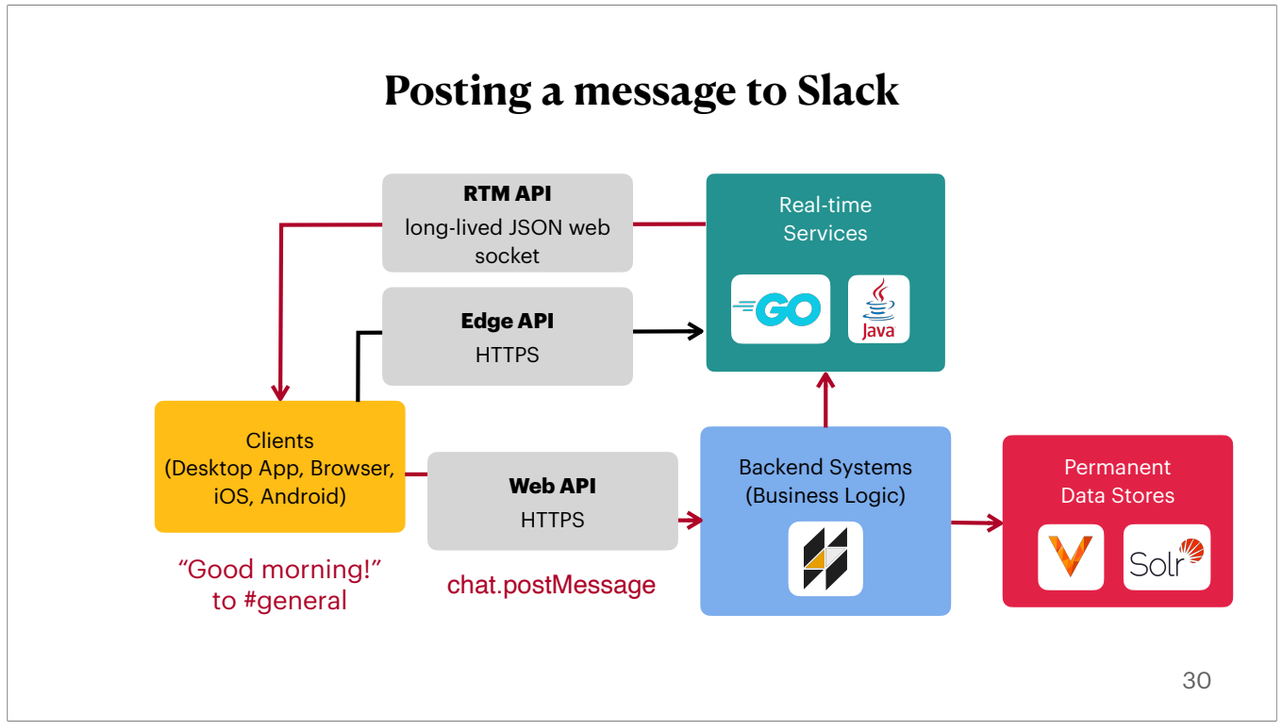
Here's a diagram I adapted from our engineering onboarding.

On the far left, we have all of our clients – desktop users, iOS users, etc. These clients send and receive information from Slack's systems via three distinct APIs:

- the Web API which queries our main backend systems responsible for most of our business logic
- the Edge API which queries our edge cache deployed across numerous points of presence throughout the world
    - a standard use case for querying the edge cache is to retrieve the your of channels when switching between them in the client
- the RTM (or real-time messaging) API



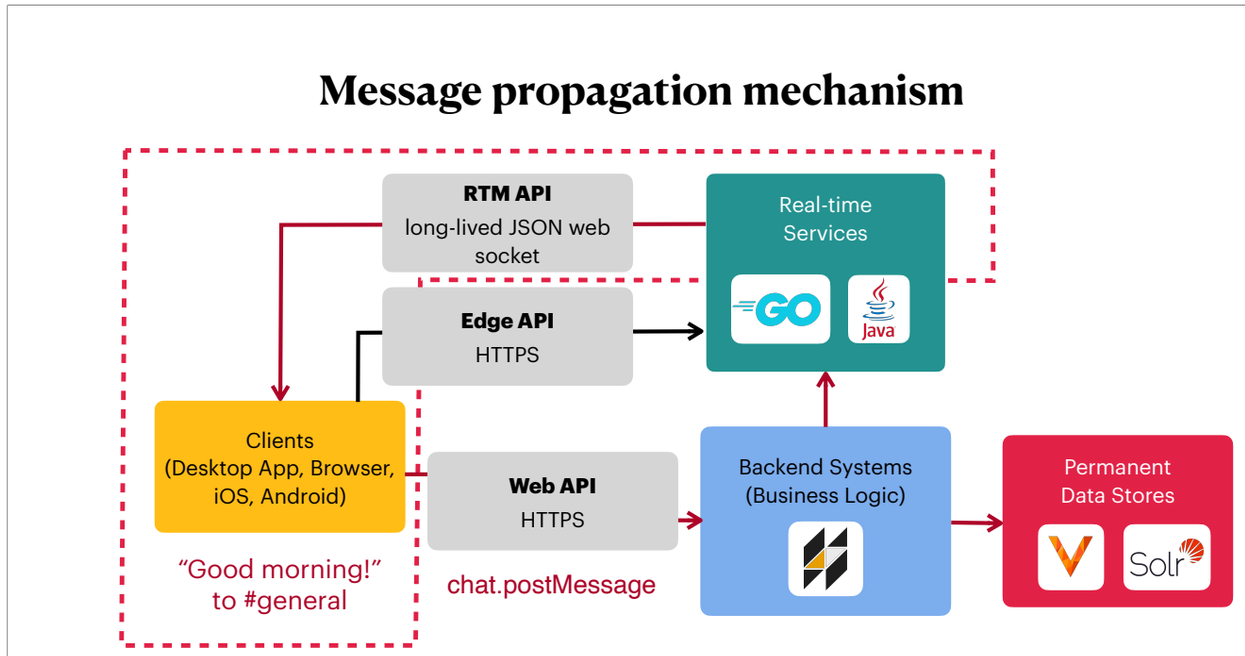
Our early attempts at load testing tools only tested load downstream of the web API. We issued barrages of requests against test environments we'd spun up with *no connected clients*, mostly to see what would happen to the monolith and downstream data stores.



I'll a bit of hand-waving here to strictly cover the necessary details; there's actually a lot more that goes on under the hood when you send a message in Slack. But if you're curious to get a more in-depth understanding of our architecture, I'm happy to answer questions during office hours!

So at a very basic level, when a user sends a message in Slack, that message will be sent to the backend systems via an HTTP request to chat.postMessage. That message will be saved to the database and propagated to our real-time services.

The real-time services – that's the teal block in the diagram – those maintain all active websocket connections with all Slack users across the world. It organizes those connections by channels. So let's say we're both connected to the #general channel when I send that "Good morning!" message, these systems are the ones that will ensure that message is properly delivered to all active clients subscribed to that channel.



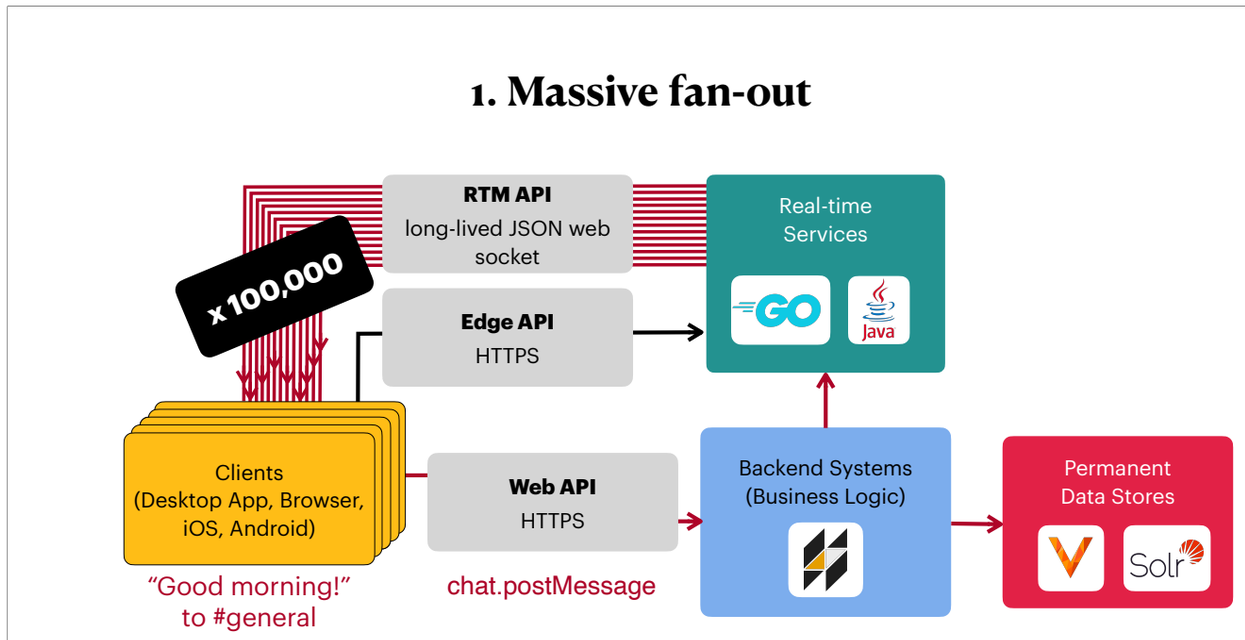
So, let's say that we're load testing chat.postMessage posting "Good morning!" to the #general channel with API blaster and there are no connected clients listening for messages.

That means that this entire outlined section isn't being exercised. We're not able to witness the impact that propagating those messages will have on the real-time services, and most importantly on our clients – our users on the receiving end of any downstream action taken within Slack.

We were *okay* with this tradeoff for a while. Our early tools gave us *some* peace of mind about how a good portion of our systems might handle load and that was fine for a while. Until it wasn't, and we needed greater assurance that we could survive specific kinds of load events.

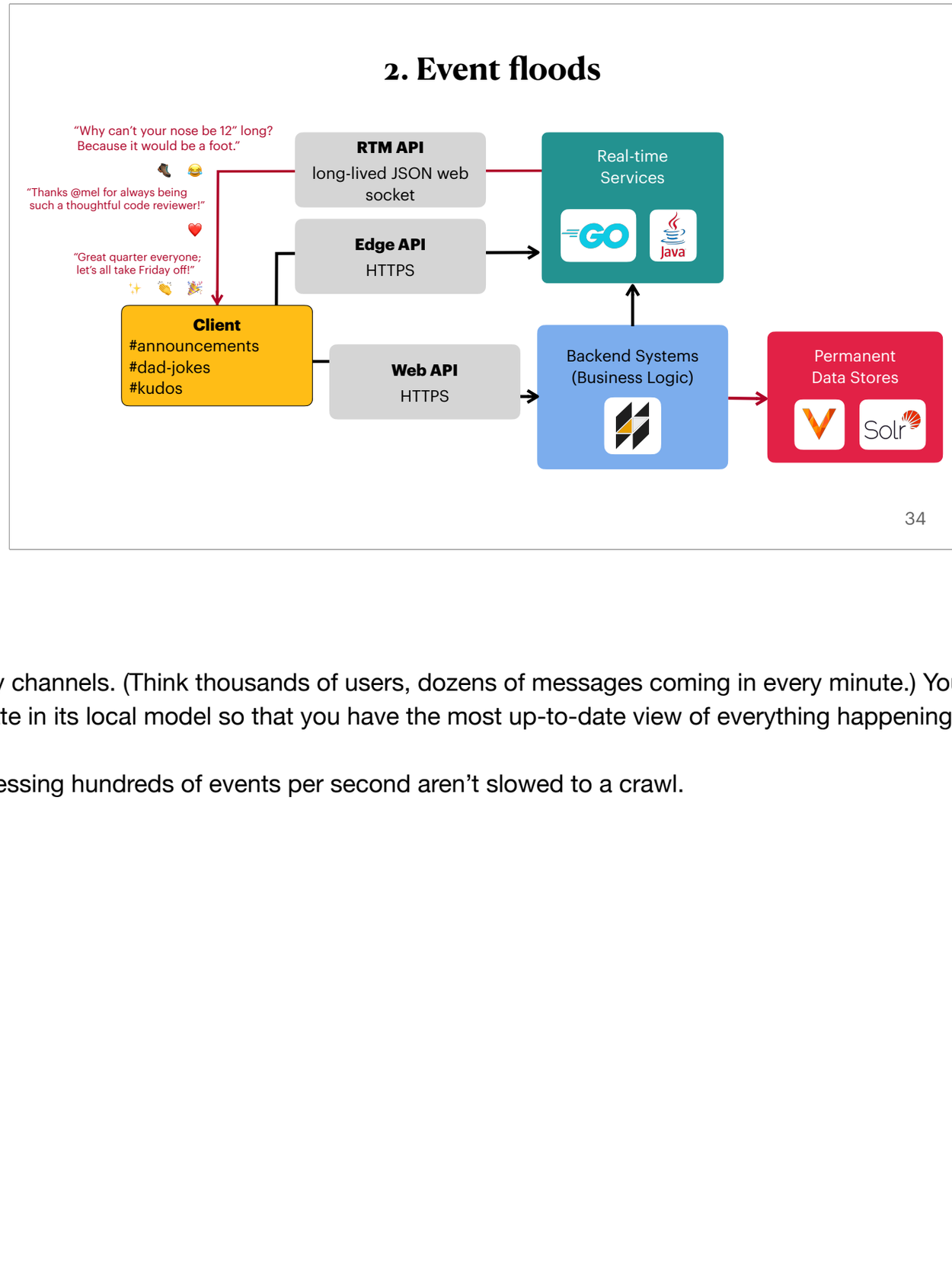
Slack is susceptible to load-related performance problems in 3 key ways

The 3 key ways that Slack is susceptible to load-related performance problems.



Let's say you have 100,000 active users in the #general channel. A single "Good morning!" message would be fanned out to all 100,000 of those clients *at once.*

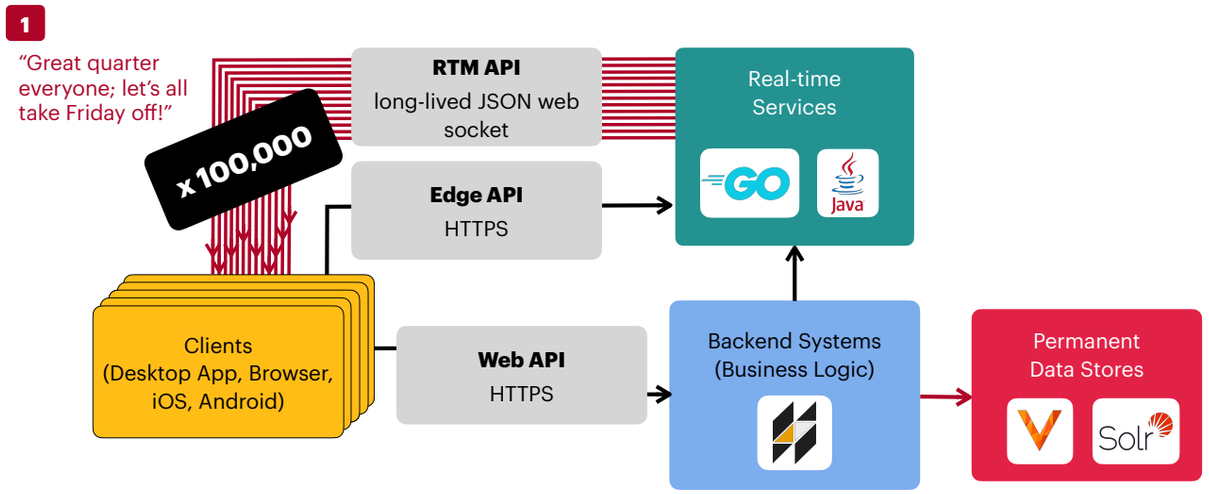
This, as you can imagine, is not an atypical use case.

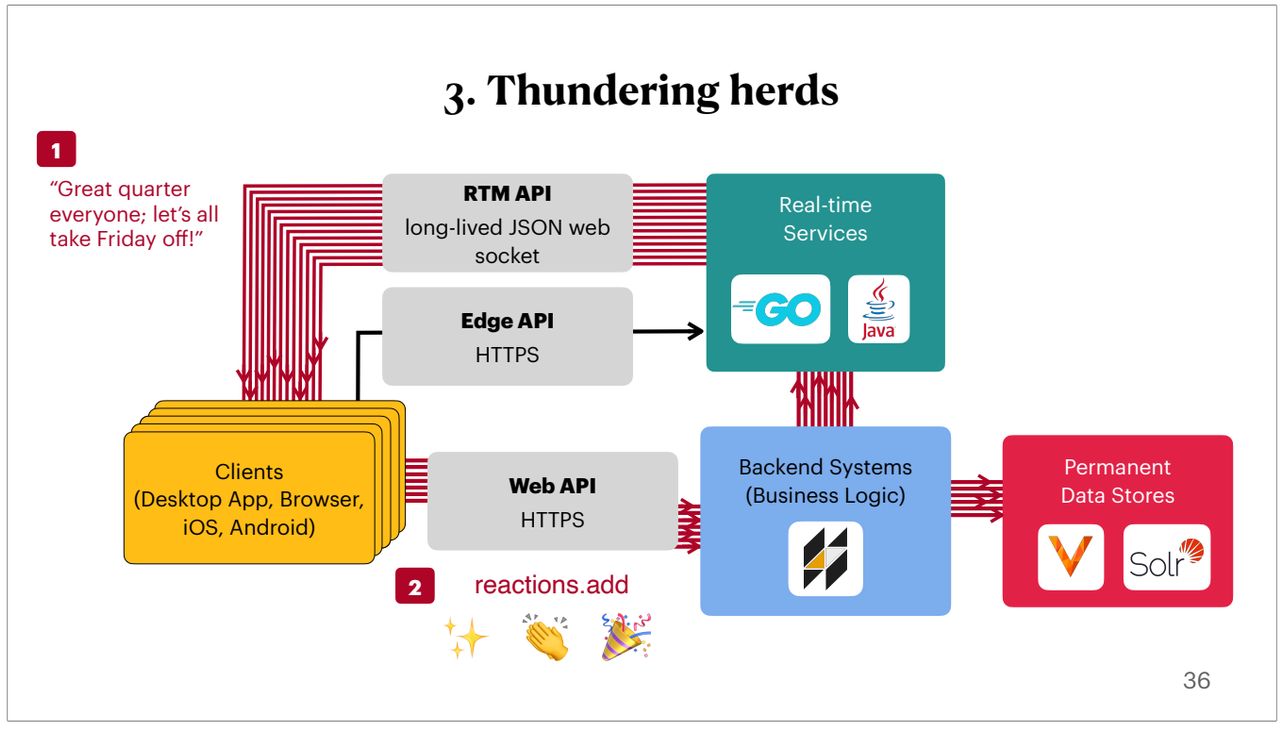


Now say you're in a handful of big, busy channels. (Think thousands of users, dozens of messages coming in every minute.) Your client is processing *all* of these events at once, deciding what it needs to update in its local model so that you have the most up-to-date view of everything happening across Slack.

We need to make sure that clients processing hundreds of events per second aren't slowed to a crawl.

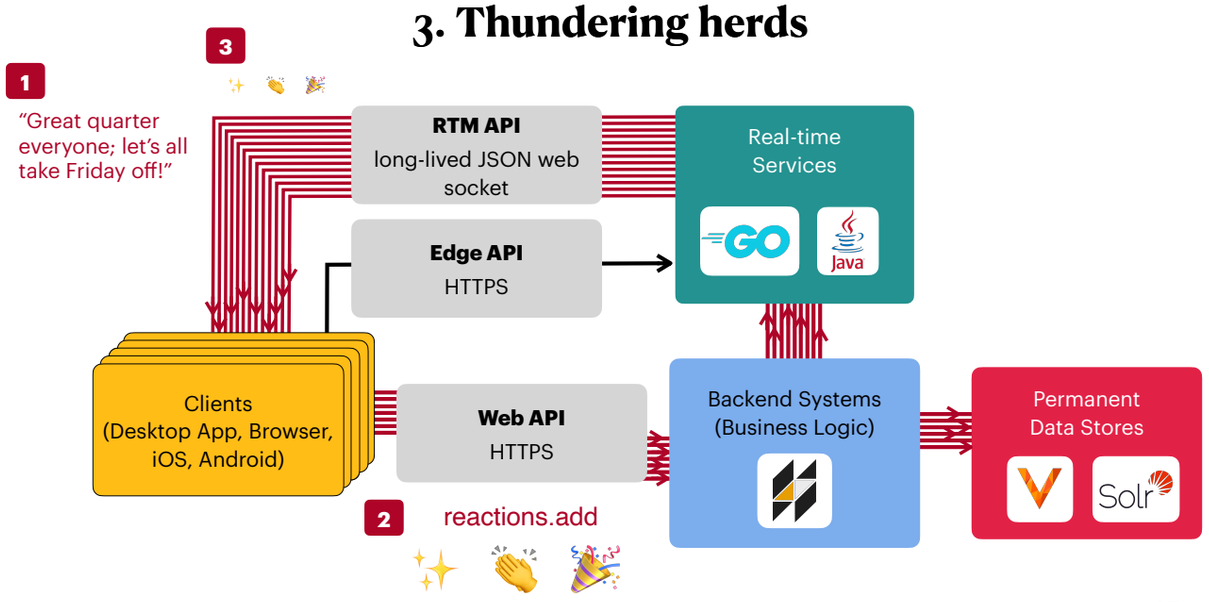
3. Thundering herds

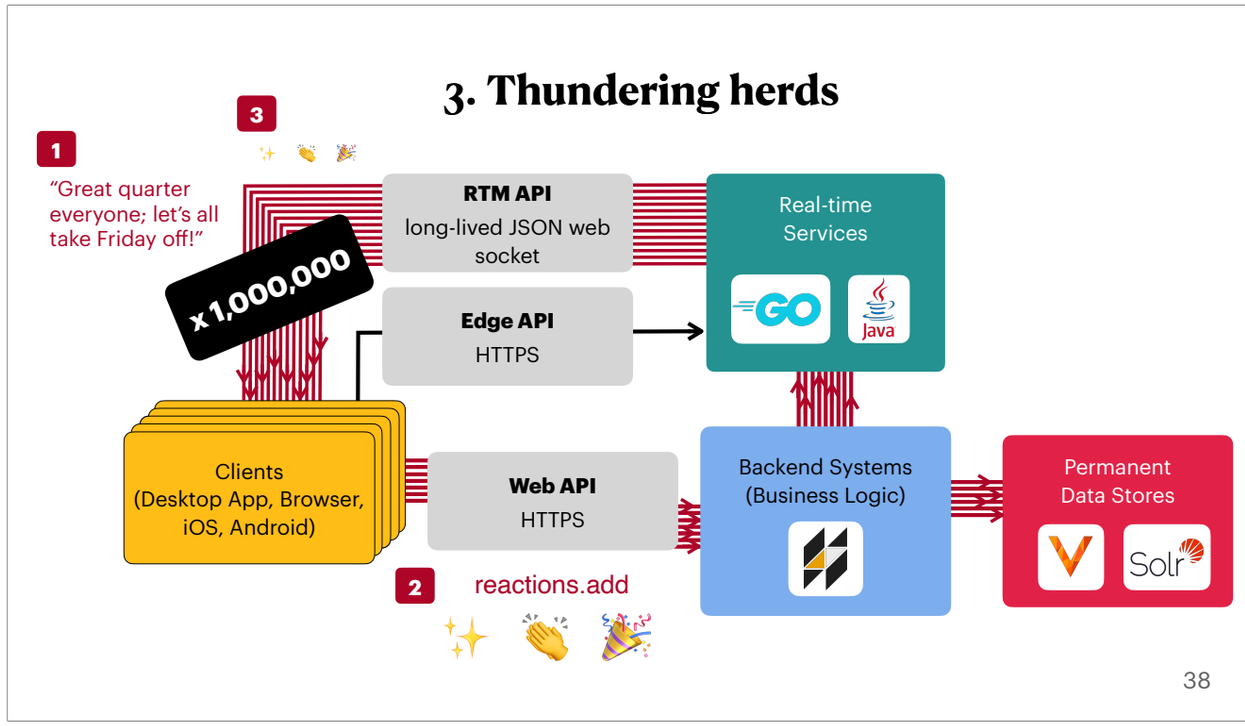




Next up, we had thundering herds.

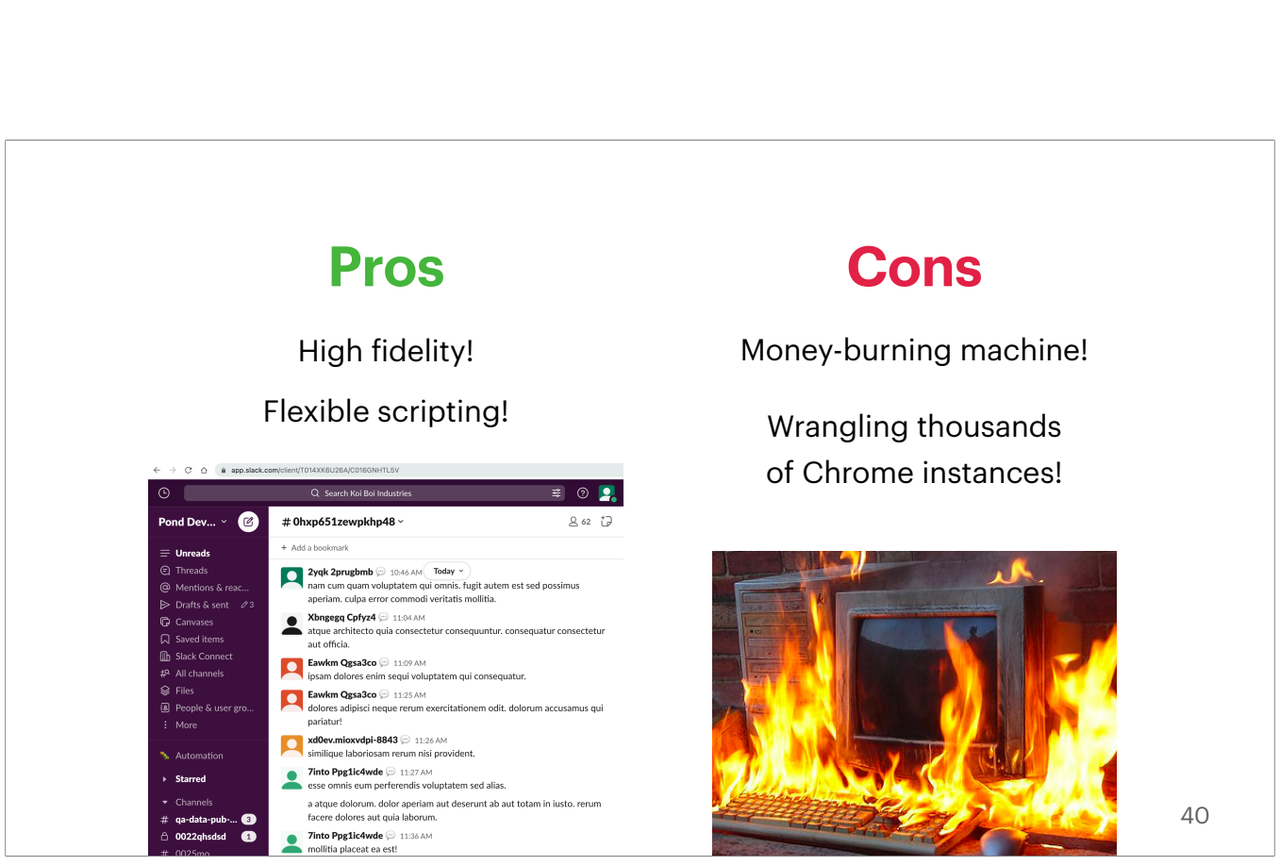
3. Thundering herds





If just 10 people send ONE reaction, that's going to turn into 10 reactions being propagated to 100,000 clients– a total of 1,000,000 websocket events. Realistically, there's no way only 10 reactions are going to land on that message, so you can image how quickly it can all balloon.

When Amazon signed, we’d already built a load testing solution that enabled us to test the entire ecosystem. But it wasn’t up to the task of load testing for 500k potential users.



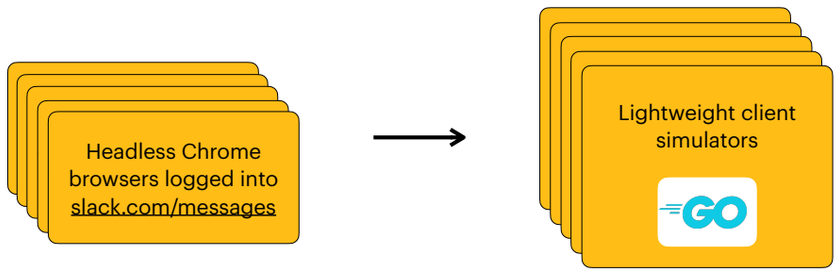
Why? Because we were spinning up headless Chrome instances pointing at slack.com, and it was prohibitively expensive.

This gave us incredibly high fidelity (we were logging into slack.com, after all) and it enabled us to test the entire infrastructure, but running headless Chrome instances was *prohibitively expensive* and difficult to manage. We spent a bunch of time tuning the resources on each pod powering these instances, but ultimately each Puppet was costing us about 0.37$ per day to run.

It took us several days to spin up over 100,000 instances, and once spun up, each pod (within the greater Kubernetes cluster) could crash partway through a test scenario due to high memory and CPU usage. If you think *your* machine struggles to run VS Code, Slack, Chrome, and iTerm at the same time, imagine running 1,000 Slacks, all logged in as different users, on the same machine.

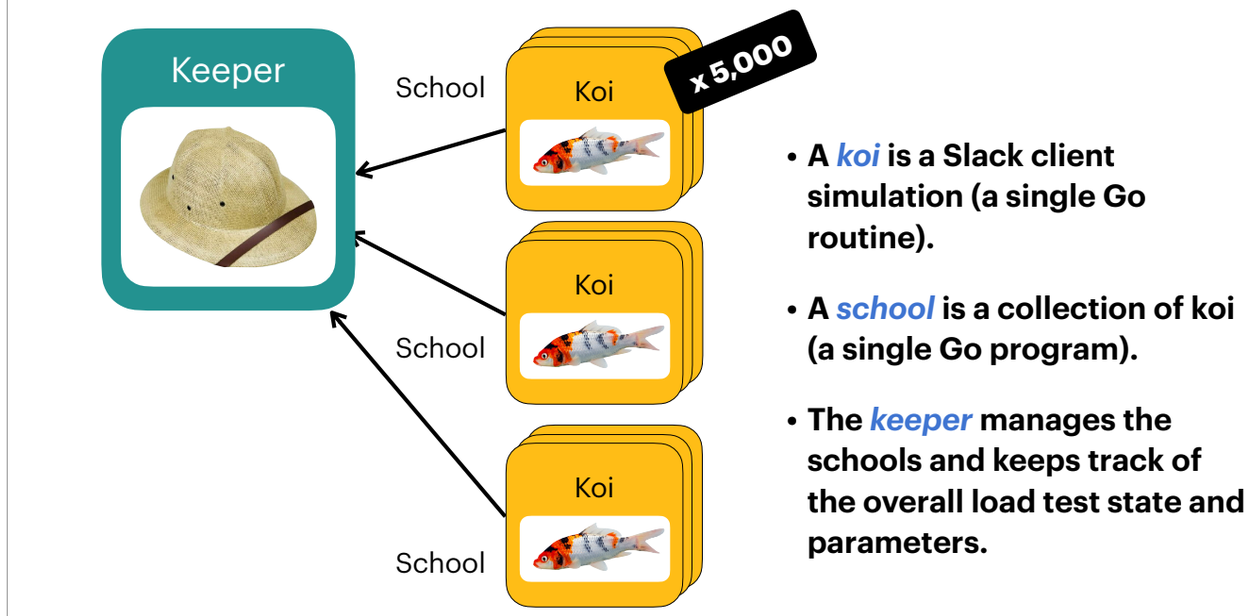
We used Puppet Show *once*. We ran through the handful of test cases we'd agreed to verify for the customer, but that was about it. It was simply too expensive to make it a viable option for regular load testing, never mind the Amazon use case.

We designed and began building Koi Pond.



We needed to build something *fast* that addressed Puppet Show's biggest pain points: it needed to be easy to operate and it needed to be cheap.

We decided to take Puppet Show and replace the headless Chrome browsers with very simple, lightweight client *simulators*.

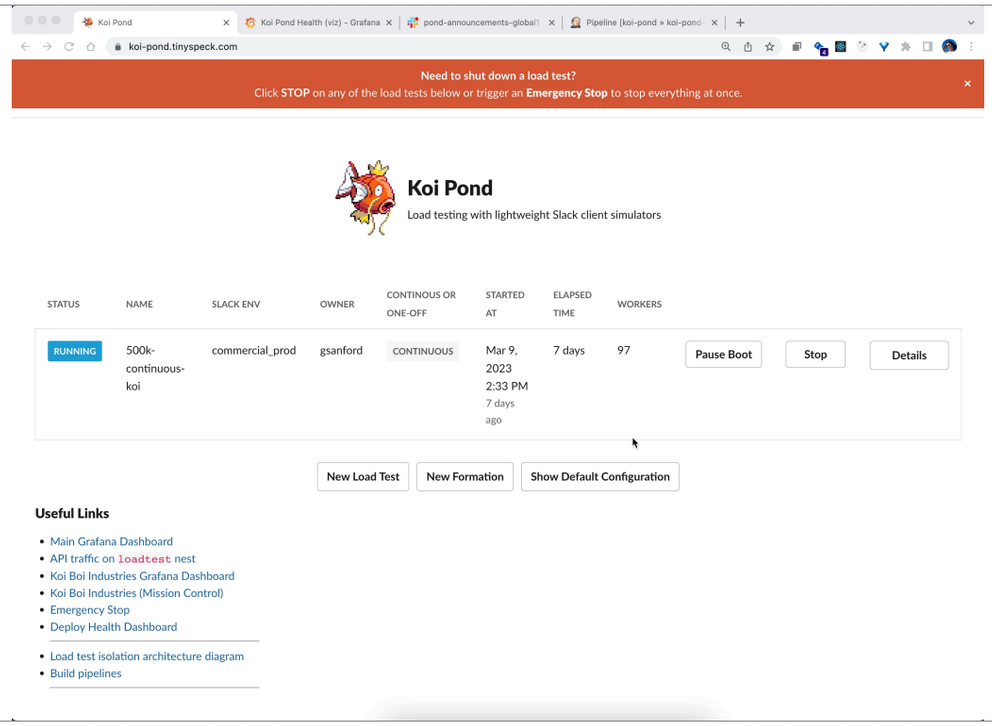


Here's what it all looks like: so on the right we have koi, each is a single Slack "client", executed as part of a single Go routine.

Koi are grouped into schools. A school is a single Go program, running on a single pod in a Kubernetes cluster.

The keeper oversees the entire ecosystem (or pond, as we like to call it). It keeps track of the overall load test state and the work distributed to each of the schools.

The only wrinkle is that we had to teach these simulators how to behave like a Slack client– how to log into Slack, establish a websocket connection, and make requests to both the web and edge APIs. Easy!


Koi Pond
Koi Pond Health (viz) - Grafana
pond-announcements-global
Pipeline [koi-pond » koi-pond
koi-pond.tinyspeck.com
Need to shut down a load test?
Click STOP on any of the load tests below or trigger an Emergency Stop to stop everything at once.
Koi Pond
Load testing with lightweight Slack client simulators
STATUS
NAME
SLACK ENV
OWNER
CONTINOUS OR ONE-OFF
STARTED AT
ELAPSED TIME
WORKERS
RUNNING
500k-continuous-koi
commercial_prod
gsanford
CONTINUOUS
Mar 9, 2023 2:33 PM
7 days ago
7 days
97
Pause Boot
Stop
Details
New Load Test
New Formation
Show Default Configuration
Useful Links
Main Grafana Dashboard
API traffic on loadtest nest
Koi Boi Industries Grafana Dashboard
Koi Boi Industries (Mission Control)
Emergency Stop
Deploy Health Dashboard
Load test isolation architecture diagram
Build pipelines

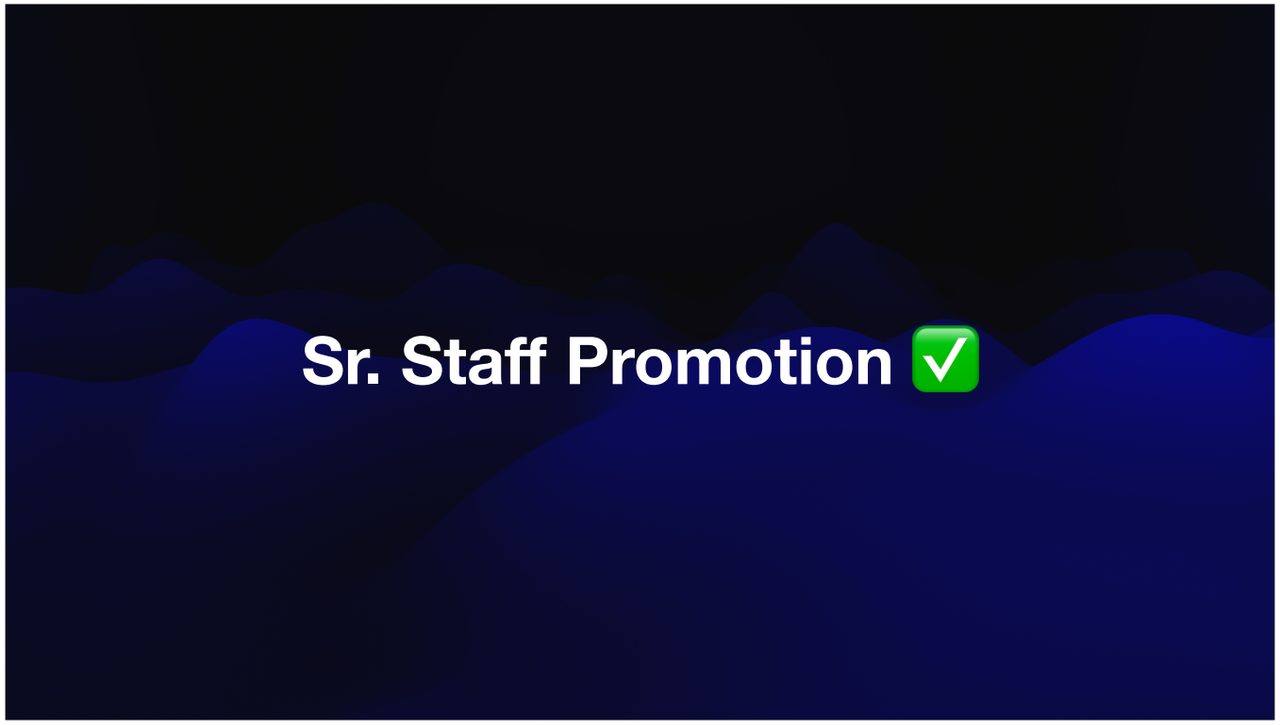
Were we successful?

Per koi cost:
0.001$ per day

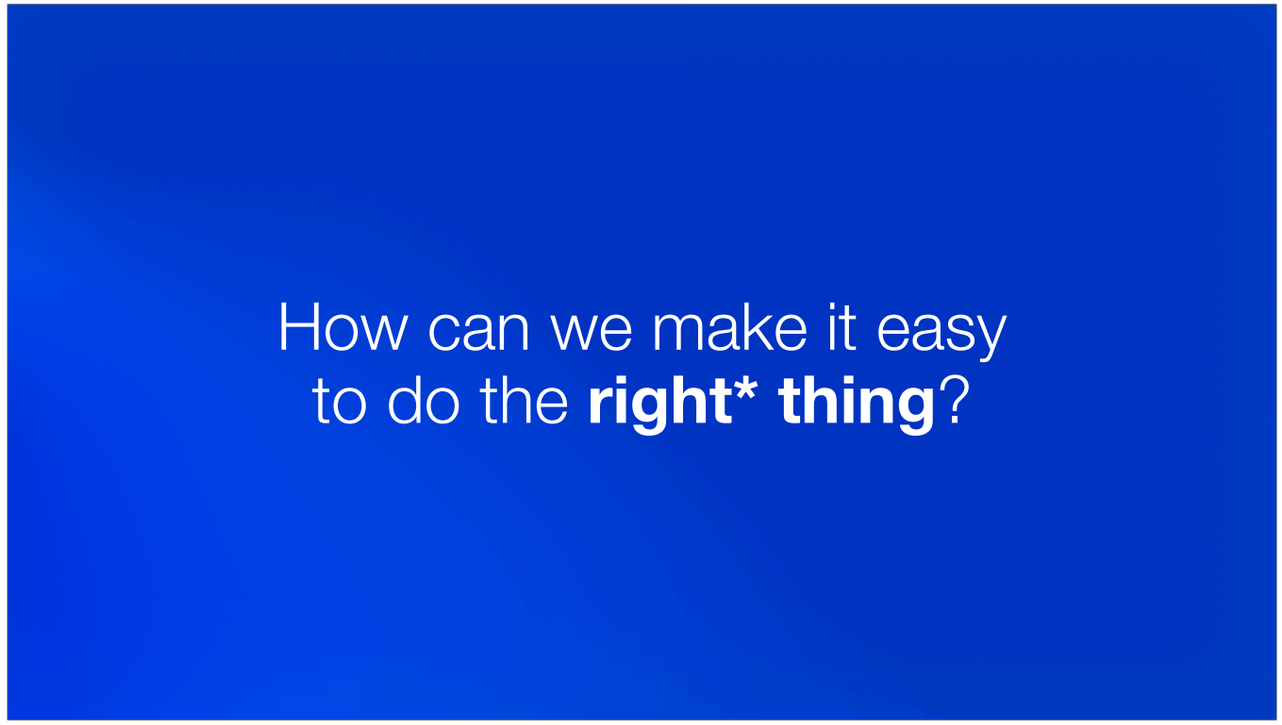
In short, we were!

We ran dozens of load tests with 500,000 simulated users in the weeks leading up to Amazon's anticipated launch date. It was a massive, coordinated effort– with someone from *nearly every engineering team* contributing.

Not only that, we didn't spend absurd amounts of money! It cost us 0.37$ per day to run a puppet, and it cost us POINT ONE cents per day to run a koi. That's around 400 times cheaper.



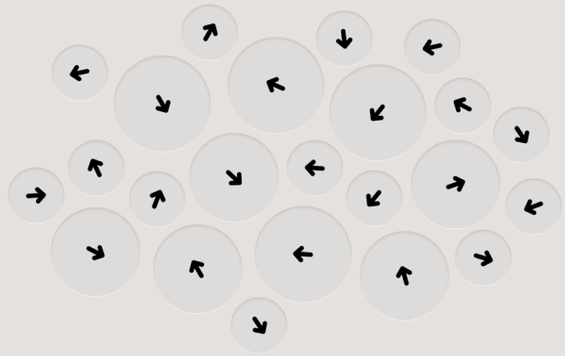
It was right around the time that the migration shipped that I got promoted to Staff.



Right here can mean just about anything– scale, performance, resiliency, availability, accessibility, uniformity. What are *you* passionate about that you can help others follow your lead on?

1. Either the business **has to want it**, or you have to **make the case for it**. 💰
2. Dig a **pit of success**, and the **moats leading into it**. ⛳
3. Scale and performance work **isn't only for wizards.** 🧙

When you're not aligned it doesn't matter how much energy you put in



You're not going anywhere. It all cancels itself out.