What Powers Instagram: Hundreds of Instances, Dozens of Technologies - Instagram Engineering

instagram-engineering.tumblr.com

One of the questions we always get asked at meet-ups and conversations with other engineers is, “what’s your stack?” We thought it would be fun to give a sense of all the systems that power Instagram, at a high-level; you can look forward to more in-depth descriptions of some of these systems in the future. This is how our system has evolved in the just-over-1-year that we’ve been live, and while there are parts we’re always re-working, this is a glimpse of how a startup with a small engineering team can scale to our 14 million+ users in a little over a year. Our core principles when choosing a system are:

- Keep it very simple
- Don’t re-invent the wheel
- Go with proven and solid technologies when you can

We’ll go from top to bottom:

OS / Hosting

We run Ubuntu Linux 11.04 (“Natty Narwhal”) on Amazon EC2. We’ve found

previous versions of Ubuntu had all sorts of unpredictable freezing episodes on EC2 under high traffic, but Natty has been solid. We've only got 3 engineers, and our needs are still evolving, so self-hosting isn't an option we've explored too deeply yet, though is something we may revisit in the future given the unparalleled growth in usage.

Load Balancing

Every request to Instagram servers goes through load balancing machines; we used to run 2 nginx machines and DNS Round-Robin between them. The downside of this approach is the time it takes for DNS to update in case one of the machines needs to get decomissioned. Recently, we moved to using Amazon's Elastic Load Balancer, with 3 NGINX instances behind it that can be swapped in and out (and are automatically taken out of rotation if they fail a health check). We also terminate our SSL at the ELB level, which lessens the CPU load on nginx. We use Amazon's Route53 for DNS, which they've recently added a pretty good GUI tool for in the AWS console.

Application Servers

Next up comes the application servers that handle our requests. We run Django on Amazon High-CPU Extra-Large machines, and as our usage grows we've gone from just a few of these machines to over 25 of them (luckily, this is one area that's easy to horizontally scale as they are stateless). We've found that our particular work-load is very CPU-bound rather than memory-bound, so the High-CPU Extra-Large instance type provides the right balance of memory and CPU.

We use http://gunicorn.org/ as our WSGI server; we used to use mod_wsgi and Apache, but found Gunicorn was much easier to configure, and less CPU-intensive. To run commands on many instances at once (like deploying code), we

use Fabric, which recently added a useful parallel mode so that deploys take a matter of seconds.

Data storage

Most of our data (users, photo metadata, tags, etc) lives in PostgreSQL; we've previously written about how we shard across our different Postgres instances. Our main shard cluster involves 12 Quadruple Extra-Large memory instances (and twelve replicas in a different zone.)

We've found that Amazon's network disk system (EBS) doesn't support enough disk seeks per second, so having all of our working set in memory is extremely important. To get reasonable IO performance, we set up our EBS drives in a software RAID using mdadm.

As a quick tip, we've found that vmtouch is a fantastic tool for managing what data is in memory, especially when failing over from one machine to another where there is no active memory profile already. Here is the script we use to parse the output of a vmtouch run on one machine and print out the corresponding vmtouch command to run on another system to match its current memory status.

All of our PostgreSQL instances run in a master-replica setup using Streaming Replication, and we use EBS snapshotting to take frequent backups of our systems. We use XFS as our file system, which lets us freeze & unfreeze the RAID arrays when snapshotting, in order to guarantee a consistent snapshot (our original inspiration came from ec2-consistent-snapshot. To get streaming replication started, our favorite tool is repmgr by the folks at 2ndQuadrant.

To connect to our databases from our app servers, we made early on that had a huge impact on performance was using Pgbouncer to pool our connections to

PostgreSQL. We found Christophe Pettus's blog to be a great resource for Django, PostgreSQL and Pgbouncer tips.

The photos themselves go straight to Amazon S3, which currently stores several terabytes of photo data for us. We use Amazon CloudFront as our CDN, which helps with image load times from users around the world (like in Japan, our second most-popular country).

We also use Redis extensively; it powers our main feed, our activity feed, our sessions system (here's our Django session backend), and other related systems. All of Redis' data needs to fit in memory, so we end up running several Quadruple Extra-Large Memory instances for Redis, too, and occasionally shard across a few Redis instances for any given subsystem. We run Redis in a master-replica setup, and have the replicas constantly saving the DB out to disk, and finally use EBS snapshots to backup those DB dumps (we found that dumping the DB on the master was too taxing). Since Redis allows writes to its replicas, it makes for very easy online failover to a new Redis machine, without requiring any downtime.

For our geo-search API, we used PostgreSQL for many months, but once our Media entries were sharded, moved over to using Apache Solr. It has a simple JSON interface, so as far as our application is concerned, it's just another API to consume.

Finally, like any modern Web service, we use Memcached for caching, and currently have 6 Memcached instances, which we connect to using pylibmc & libmemcached. Amazon has an Elastic Cache service they've recently launched, but it's not any cheaper than running our instances, so we haven't pushed ourselves to switch quite yet.

Task Queue & Push Notifications

When a user decides to share out an Instagram photo to Twitter or Facebook, or when we need to notify one of our Real-time subscribers of a new photo posted, we push that task into Gearman, a task queue system originally written at Danga. Doing it asynchronously through the task queue means that media uploads can finish quickly, while the 'heavy lifting' can run in the background. We have about 200 workers (all written in Python) consuming the task queue at any given time, split between the services we share to. We also do our feed fan-out in Gearman, so posting is as responsive for a new user as it is for a user with many followers.

For doing push notifications, the most cost-effective solution we found was https://github.com/samuraisam/pyapns, an open-source Twisted service that has handled over a billion push notifications for us, and has been rock-solid.

Monitoring

With 100+ instances, it's important to keep on top of what's going on across the board. We use Munin to graph metrics across all of our system, and also alert us if anything is outside of its normal range. We write a lot of custom Munin plugins, building on top of Python-Munin, to graph metrics that aren't system-level (for example, signups per minute, photos posted per second, etc). We use Pingdom for external monitoring of the service, and PagerDuty for handling notifications and incidents.

For Python error reporting, we use Sentry, an awesome open-source Django app written by the folks at Disqus. At any given time, we can sign-on and see what errors are happening across our system, in real time.

You?

If this description of our systems interests you, or if you're hopping up and down

ready to tell us all the things you'd change in the system, we'd love to hear from you. We're looking for a DevOps person to join us and help us tame our EC2 instance herd.

instagram-engineering.tumblr.com ➔