RetailNext
Comprehensive In-Store Analytics

JANUARY 2014

Continuous Delivery at RetailNext

Today I'm going to talk about continuous delivery and how we do it at RetailNext. Continuous delivery is a somewhat new technique and largely limited to cloud services. Lots of large companies use this. Places like Facebook, Twitter, Etsy and Netflix. It sounds crazy at first but it's real and you can do it responsibly.

Product Requirement

Can’t you fit one more feature/bugfix into this release?

This is the product requirement we’re trying to address. The normal way to deal with this is with yelling. We’re going to put our product manager hats on and figure out what they really want.

What Do They Really Want?

Customers shouldn’t have to wait to get new features and bugs fixed.

Sure! I agree. So what does this mean for us?

Engineering Requirement

We need to be able to release at any time

In practical terms there's not much middle ground between being able to release at any time and releasing on some schedule, no matter how frequent those scheduled releases are. One week releases suffer from the same problems and pressures as one year releases. No matter how frequent your releases are you will always be pressured to lengthen them. If you want to release quickly you need to be able to release code at any time.

Technical Requirements

- Zero-downtime upgrades
- No more then 20 minutes between checkin and release
- Only one install of the software
- Operations is part of the product

So how does this translate into technical requirements?

If we're upgrading randomly in the middle of the day there can't be any downtime. You can't stop the world, do an extended upgrade and turn things back on. You can't even wait 30 seconds while your web server restarts.

You need to be able to go from a diff to a release quickly. No magic number. 20 minutes is my threshold. I don't think you can reliably do faster then ten, 30 is too long.

There can only be one install of the software. It may span multiple machines but you need to start thinking of it as one large distributed system.

Finally, operations needs to be part of the product. When you add new features you need to figure out how that will run in production and make all the tools we'll need. There's no time for an ops team to test out the release and make their own tooling.

Zero-Downtime Upgrades

- Services must restart quickly
- Multi-tier systems need to tolerate outages
- Need to handle heterogeneous environments
- The upgrade process can’t disrupt operations

Zero downtime upgrade are one of the trickier requirements. Not all services can do this. We have some systems that we upgrade less frequently during off hours.

In general this boils down to “restart quickly and buffer while other people are restarting”. Services need to start up quickly, and while they are any consumers need to wait and hold on to requests. Lots of techniques for doing this.

You need to handle heterogeneous code environments. We’re not going to shut everything off during the upgrade so one machine will be first and one will be last. In our system there can be months in between. The main impact is that multi-tier services need to be rolled out in stages. Also carefully.

Finally, the upgrade process can’t disrupt normal operations. You can’t use all the CPU or disk I/O. Upgrades (especially data migrations) are sometimes slower and machines need more resources then they would otherwise.

20 Minutes to Release

- QA has to happen before the merge to master
- All regression tests must be automated
- Full test runs need to be fast
- Need to fix problems by rolling forward

This one is mostly process solutions. If we want to roll code out quickly QA needs to be done during development and not afterwards. Once it's checked in we need to be confident that the tests will catch any regressions. You'll get a big test suite but you need to make sure that it still runs fast. Lots of engineering challenges there.

The last one is somewhat unintuitive. If we're want to merge stuff into master and deploy quickly it means we won't be able to roll back if something goes wrong. Downgrade tests are very difficult to automate. Since code is going to get merged to master frequently and in an arbitrary order you won't be able to write or perform downgrade tests in advance.

Only One Install

- Only one install of a large distributed system
- Only one team responsible for running it
- Only in the cloud

We can do continuous delivery on managed systems. The cloud is great. We do it with our managed on-premise systems also, but we put different requirements on our customer then the RN support team. We need to have direct access. You can’t do this over a live meeting. It’s a tradeoff and it takes work.

Operations Are The Product

- Normally dev writes the code and ops figures out how to run it
- There's no time. One engineering org responsible for both.

This should be obvious by now. Since there's little time between a checkin and release there's no time for an ops group to learn about the changes and write tools. The engineering team is now responsible for operations. If something goes wrong with your code you will be responsible for investigating and fixing the production systems. You are responsible for resource planning and tooling. We may have a team called "ops" eventually but this will never change.

Culture Impact

You are ultimately responsible for the quality of your code

This is the ultimate impact of all of this. Individual engineers are closer to production and the customer and they need to take on more responsibility.

Big Takeaway

Everyone Gets What They Want

- Customer: Faster features and bug fixes
- Product: Faster feedback from the customer
- Engineering: Replace arguing with exciting technical challenges

The big takeaway is that everyone gets what they want. The customer gets code faster. Product management gets better customer feedback and engineering gets to trade arguments for exciting technical challenges.

How We Do it!