Practicing Deployment

Laura Thomson
laura@mozilla.com
@lxt
Disclaimers

Not about tools
Not prescriptive
Recognize where you are, and where you want to be
Models
Maturity model

(after Capability Maturity Model, CMU)

1. Initial: “chaotic”, “individual heroics”
2. Repeatable: documented
3. Defined: standard process, some tuning
4. Managed: measured
5. Optimizing: continual improvement, innovation
Initial

Startup phase of many projects

Long term

Push code whenever you feel like it

Devs push code

Not a lot of tests, automation, or verification
Repeatable

Often after a 1.0, first non-beta ship, or first ship with a significant number of users

Some kind of documented/known process

Push when a feature is done: less often than initially, typically
Defined

Procedural documentation
Start of automation
Often done by a sysadmin
Managed Automation

Tools: packaging

Verification post-push

Measurement: How often do we push? How long does it take? How did that push affect performance?
Optimized

Take the drama out of deployment

Often - not essentially - continuous deployment

Typically a lot of test automation

Lightweight
How much do we ship?

(Size of a release)

Start with per-patch pushes

Move to features

Then to releases

Then back to features

The back to per-patch pushes
Velocity models

(Frequency of a release)

Critical mass

Single hard deadline

Train model

Continuous deployment
Critical mass

“enough stuff to release”

MVP

smallest quantum with user value
Single hard deadline

Support for X by date Y
Shipping to a marketing plan
Hard deadlines are hard
Train model

Release e.g. every Wednesday
Whatever’s ready to ship, ships
Anything else catches the next train
Continuous deployment

Ship each change as soon as it’s done

Continuous is kind of a misnomer; deployment is discrete
Tools and practices
Source control

Stable vs unstable

Branch per bug, branch per feature

“git flow” is overkill, but you need a process

If it’s not per-patch-push, tag what you push

Open source needs ESRs even if you’re high velocity
Dev Envs

Dev’s laptop is a horrible environment
VMs can be hard to maintain
Development databases are hard: fake data, minidbs
Development API sandbox
Lightweight set up and tear down VMs
“Development” staging server (unstable)
“Try” servers for branches
Staging

Staging environment MUST REFLECT PRODUCTION

Same versions, same proportions: a scale model

Realistic traffic and load (scale)

Staging must be monitored

Staging must have managed configuration
One Box Fail

Staging needs to be more than one box
If you have multiple databases or webheads or whatever in prod...you need that in staging
Continuous Integration

Build-on-commit
VM-per-build
Leeroy/Travis (PR automation)
Run all unit tests
(Auto) push build to staging
Run more tests (acceptance/UI)
Testing

Unit tests: run locally, run on build

Acceptance/User tests: run against browser (Selenium, humans)

Load test: how does it perform under prod load?

Smoke test: what’s the maximum load we can support with this build?
Deploy deployment tools

It doesn’t really matter what you use

Automate it

Do it the same way in staging and production

Use configuration management to deploy config changes and manage your platform...the same way in staging and production
QA

Feature tests on unstable
Full tests on stage
Full tests on production (verification)
Measurement

Monitoring

Performance testing

Instrument, instrument, instrument

Is it actually possible to have too much data? (Hint: yes. But only if no insight)
Postmortems

What went right

What went wrong

Blameless: scapegoats only hurt you
When things go wrong
WORKED FINE IN DEV

OPS PROBLEM NOW
Quantum of deployment
(via Erik Kastner)
“What’s the smallest number of steps, with the smallest number of people and the smallest amount of ceremony required to get new code running on your servers?”

http://codeascraft.etsy.com/2010/05/20/quantum-of-deployment/
Chemspills

Even if you have heavyweight/non-automated deployments, what does a chemspill look like?
THIS IS NOT A DRILL
Fail forward

Fail forward: the premise that Mean Time To Repair is the key measure, not MTBF
Fail

Sometimes you can’t fail forward

Example: intractable/unforeseen performance problem, hardware failures, datacenter migrations

Hit upper time limit (failing forward is taking too long)
Rollback

Going back to the last known good

Having a known process for rollback is just as important as having a known process for deployment

Practice rollbacks
Decision points

When shipping something new, define some rules and decision points

If it passes this test/performance criteria we’ll ship it

If these things go wrong we’ll roll back

Make these rules beforehand, while heads are calm
Feature switches

A nicer alternative to rollback

Turn a feature on for a subset of users: beta users, developers, n% of users

Turn it on for everybody

Turn things off if you’re having problems or unexpected load: “load shedding”
Continuous Deployment
What is CD?

Total misnomer
Not continuous, discrete
Automated not automatic, generally
Intention is push-per-change
Usually driven by a Big Red Button
Technical recommendations

Continuous integration with build-on-commit

Tests with good coverage, and a good feel for the holes in coverage

A staging environment that reflects production

Managed configuration

Scripted single button deployment to a large number of machines
People and process

High levels of trust
Realistic risk assessment and tolerance
Excellent code review
Excellent source code management
Tracking, trending, monitoring
Testing vs monitoring

Run tests against production

Continuous testing = one kind of monitoring

Testing is an important monitor

You need other monitors

You need tests too
You should build the capability for continuous deployment even if you never intend to do continuous deployment.
The only way to get good at deployment is to deploy a lot.
Questions?

laura@mozilla.com