Deployment

SWE 432, Fall 2016
Design and Implementation of Software for the Web
Today

• Big picture: from ideas to awesome products
  • How do we structure the process that gets us those products?

• Buzzwords:
  • DevOps, Continuous Integration, Continuous Deployment, Continuous Delivery, and how we got there

• No specific technologies!

For further reading:
Chuck Rossi (Facebook) on Continuous Mobile Release
http://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/
What is a software process?

• A structured set of activities required to develop a software product
  • Specification
  • Design and implementation
  • Validation
  • Evolution (operation and maintenance)
• Goal: Minimize Risk
  • Falling behind schedule
  • Changes to requirements
  • Bugs/unintended effects of changes
Software Specification

• The process of establishing what features and services are required, as well as the constraints on the system’s operation and development.

• Requirements engineering process
  • Feasibility study;
  • Requirements elicitation and analysis;
  • Requirements specification;
  • Requirements validation.
Software Design & Implementation

• The process of converting the system specification into an executable system.

• Software design
  • Design a software structure that realizes the specification;

• Implementation
  • Translate this structure into an executable program;
  • The activities of design and implementation are closely related and may be inter-leaved.
Software Validation

• Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the customer(s).

• Involves checking and review processes, and acceptance or beta testing.

• Custom software: Acceptance testing involves executing the system with test cases that are derived from the real data to be processed by the system in the customer’s environment.

• Generic software: Beta testing executes the system in many customers’ environments under real use.
Software Evolution

• Software is inherently flexible and can change.
• As requirements change due to changing business circumstances, the software that supports the business must also evolve and change.
• Although there has historically been a demarcation between development and evolution, this is increasingly irrelevant as fewer and fewer systems are completely new.
Process Models

• If we say that building software requires:
  • Specification
  • Design/Implementation
  • Validation
  • Evolution
• How do we structure our organization/development teams/tasks to do this most efficiently?
Code-and-Fix

- Really Bad
- Really Common
- Advantages
  - No Overhead
  - No Expertise
- Disadvantages
  - No means of assessing progress
  - Difficult to coordinate multiple programmers
- Useful for “hacking” single-use/personal-use programs: start with empty program and debug until it works
Waterfall Model

- Widely used today
- Advantages
  - Measurable progress
  - Experience applying steps in past projects can be used in estimating duration of “similar” steps in future projects
  - Produces software artifacts that can be re-used in other projects
- Disadvantages
  - Difficulty of accommodating change after the process is underway: One phase has to be complete before moving onto the next phase.
• Agile results in an *iterative* model, where each iteration is several weeks long and results in several features being built
• Recognize that requirements ALWAYS evolve as you are trying to build something
• Plus, maybe you can get useful feedback by delivering a partial app early
Continuous Development

• Like agile, but…
  • We are always working on different features
  • We have a formal mechanism for deploying new versions of code and validating (test/staging/production)
The value of the Staging Environment

• As software gets more complex with more dependencies, it's impossible to simulate the whole thing when testing

• Idea: Deploy to a complete production-like environment, but don't have everyone use it

• Examples:
  • “Eat your own dogfood”
  • Beta/Alpha testers

• Lower risk if a problem occurs in staging than in production
Test-Stage-Production

Developer Environments

Testing Environment

Beta/Dogfooding

Staging Environment

User Requests

Production Environment

Revisions are “promoted” towards production
# Operations Responsibility

- Once we **deploy**, someone has to monitor software, make sure it’s running OK, no bugs, etc.
- Assume 3 environments:
  - Test, Staging, Production
- Whose job is it?

<table>
<thead>
<tr>
<th></th>
<th>Developers</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waterfall</td>
<td></td>
<td>Test</td>
</tr>
<tr>
<td>Agile</td>
<td>Test</td>
<td>Staging</td>
</tr>
<tr>
<td>DevOps</td>
<td>Test</td>
<td>Production</td>
</tr>
<tr>
<td></td>
<td>Staging</td>
<td>Production</td>
</tr>
<tr>
<td></td>
<td>Production</td>
<td>Production</td>
</tr>
</tbody>
</table>
DevOps Values

- No silos, no walls, no responsibility "pipelines"
- One team owns changes "from cradle to grave"
- *You* are the support person for your changes, regardless of platform
- Example: Facebook mobile teams
DevOps Values

• No silos, no walls, no responsibility "pipelines"
• One team owns changes "from cradle to grave"
• You are the support person for your changes, regardless of platform
• Example: Facebook mobile teams
Continuous X

• Continuous Integration:
  • A practice where developers automatically build, test, and analyze a software change in response to every software change committed to the source repository.

• Continuous Delivery:
  • A practice that ensures that a software change can be delivered and ready for use by a customer by testing in production-like environments.

• Continuous Deployment:
  • A practice where incremental software changes are automatically tested, vetted, and deployed to production environments.
Continuous Integration

Developers

Check code in

Build agent listens for changes ...

Repository

and notifies team if there's a problem.

Error
Continuous Integration

• Commit Code Frequently
• Don’t commit broken code
• Fix broken builds immediately
• Write automated developer tools
• All tests and inspections must pass
• Run private builds
• Avoid getting broken code
Deployment Pipeline

1. Local Dev/Test
2. Commit to Version Control
3. Build & Run Tests
4. Deploy to Staging
   - Monitoring
5. Deploy to Production
   - Monitoring
Deployment Pipeline

- Even if you are deploying every day, you still have some latency
- A new feature I develop today won't be released today
- But, a new feature I develop today can begin the release pipeline today (minimizes risk)
- **Release Engineer**: gatekeeper who decides when something is ready to go out, oversees the actual deployment process
Deployment Example: Facebook.com

- Developers working in their own branch
- When feature is ready, push as 1 change to master branch

master branch

- ~1 week of development
- Weekly
- All changes from week that are ready for release

release branch

- 3 days
- Stabilize
- 4 days
- Release Branch

production

- Twice Daily

Your change doesn’t go out unless you’re there that day at that time to support it!

“When in doubt back out”
Continuous Integration & Continuous Deployment

- Thousands of changes coming together at once
- To isolate problems:
  - Every time that every change is potentially going to be introduced, the entire system is integrated and tested
  - Facebook does 20,000-30,000 complete integrations PER DAY for mobile alone
- General rule:
  - Cost of compute time to run tests more often is way less than the cost of a failure
Blue-Green Deployment

- Always have 2 complete environments ready:
  - One that you’re using now
  - One that you’re just about ready to use
- Easily switch which is handling requests
A/B Testing

- Ways to test new features for usability, popularity, performance
- Show 50% of your site visitors version A, 50% version B, collect metrics on each, decide which is better
Monitoring

- **Hardware**
  - Voltages, temperatures, fan speeds, component health
- **OS**
  - Memory usage, swap usage, disk space, CPU load
- **Middleware**
  - Memory, thread/db connection pools, connections, response time
- **Applications**
  - Business transactions, conversion rate, status of 3rd party components
When stuff goes wrong

- Automated monitoring systems can notify “on-call” staff of a problem
- Triage & escalation
Monitoring Dashboards
Canaries

Monitor both:
But minimize impact of problems in new version
Making it happen

- Build Tools
- Test Automation
- Build Servers
- Deployment Tools
Build Tools

• Need to be able to automate construction of our executable software… Example:
  • “Install d3 with bower with grunt with npm with brew.”
    *phew*
• We talked about it getting a little better with npm (which installs modules), but need something more general to handle build tasks:
  • Minify my code
  • Run my tests
  • Generate some documentation
  • Deploy to staging
• Ensure that builds are repeatable, reproducible and standard
Build tools example: Push to gh-pages

```javascript
grunt.initConfig({
  'gh-pages': {
    options: {
      base: 'dist'
    },
    src: ['**']
  }
});
```
Build Servers

• Once we have a standard mechanism for describing how to build our code, no reason to only build it on our own machine
• Continuous Integration servers run these builds in the cloud
  • Bamboo, Hudson/Jenkins, TravisCI
• Easy to use - typically monitors your source repository for changes, then runs a build
• Really helps with organizing tests and results
• Can scale the build server independently of the rest of your processes
TravisCI

GitHub

Commits code to

Developer

Checks for updates

TravisCI

Runs build for each commit
TravisCI

- Can see history and status of each branch
TravisCI

- Can also see status per-commit
Summary

• DevOps: Developers as Operators
• Continuous Integration & Deployment: Techniques for reducing time to get features out the door
• Staging environments reduce risk
• Build Systems and Services help automate CI