TestOps: Continuous Integration when infrastructure is the product

Barry Jaspan
Senior Architect, Acquia Inc.

This talk is about the hard parts.

Rainbows and ponies
have left the building.

Intro to Continuous Integration

- Maintain a code repository
- Automate the build
- Make the build self-testing
- Everyone commits to the baseline every day
- Every commit (to baseline) should be built
- Keep the build fast
- Test in a clone of the production environment
- Make it easy to get the latest deliverables
- Everyone can see the results of the latest build
- Automate deployment

Intro to Acquia Cloud

- PaaS for PHP apps
  - Multiple environments: Dev, Stage, Prod, …
  - Continuous Integration environment for your app
  - Special sauce for Drupal
- Obligatory Impressive Numbers, ca. 03/2014
  - 27 billion origin hits per month
  - 422 TB data xfer/month
  - 8000+ EC2 instances
- Release every 1.6 days on average
  - Each release alters the infrastructure under thousands of web apps that **we do not control**
- Our customers REALLY hate downtime

Server configuration is software

- Puppet, Chef, and similar tools turn server config into software
  - Hurray!
  - Deserves the same best practices as app development
- TestOps == test-related CI principles for infrastructure software
  - Make the build self-testing
  - Test in a clone of the production environment
  - Keep the build fast
- “If it isn’t tested, it doesn’t work.”

Unit tests vs. System tests

- Unit tests isolate individual program modules
  - Injection mocks out external systems
- Problem: You can’t mock out the real world and get accurate results
  - Server configuration interacts with the OS, network, and services
  - “puppetd –noop” doesn’t tell the whole story
- Sample failure
  - crontab and cron daemon race condition

Unit tests vs. System tests

- System tests are end-to-end
  - Apply code changes to real, running servers
  - Exercise the infrastructure as the app(s) will
- Problem: Reality is very messy!
  - Launch failures
  - Race conditions
  - Vendor API scheduled maintenance
  - Cosmic rays

System tests FTW

- For infrastructure, system tests are essential

- Making Acquia Cloud’s system tests reliable is the hardest engineering challenge I’ve ever faced.
- We would be totally dead without them.

Test in a clone of production

- No back-doors to make tests “easier”
- Ex: HIPPA, PCI, etc. security requirements
  - Access only from a bastion server using two-factor authentication
  - No root logins, even from the bastion
  - Any failure here locks Ops out from all servers!
- Tests operate just as admins do
  - Most tests operate “from a bastion”:
    ssh testadmin@server ‘sudo bash -c “cmd”’
  - Ensures the code works in production

Server build strategy

- Always build from a reference base
  - e.g. “Ubuntu 12.04 Server 64-bit”
  - No incrementally evolved images
  - Puppet makes this natural
  - Sidebar: Docker gets this **totally wrong**!
- Puppet can take a while
  - Makes tests and MTTR slow
- More on this later…

Basic build tests

- Launch VMs, run puppet
  - Replicate a functional production environment
  - Isolated from production
- Scan syslog for errors
- Test config files, daemons, users, cron jobs…
- Sample failure
  - Incorrect Puppet dependencies work while iterating on development instances but not on clean launch

Functionally test the moving parts

- Backup and restore
- Message queues
- Worker auto-scaling
- Load balancing with up and down workers
- ELB health check and recovery
- Database failover
- Monitoring & Alerting
- Self healing

Application test

- Install and verify application(s)
  - Real site code
  - Real site db (scrubbed)
- Cause app to exercise the infrastructure
  - Write to database, message queues, etc.
  - Verify success on the back end
- Operate app on degraded infrastructure
  - Failed web nodes
  - Database failover

Reboot test

- Reboot all test servers
- Re-run build tests
  - Filesystems mounted?
  - Services restarted?
- Re-run functional and application tests
- Sample failure
  - Database quota daemon starts via /etc/init.d before MySQL daemon, then aborts

Relaunch test

- Relaunch all test servers from base image
  - Simulates server crash and recovery
  - Persistent data retained?
  - Server rejoins services? (e.g. MySQL replication restarts)
  - Unexpected issues, ex: tmpfs
- Re-run build, functional, and application tests
- Sample failure
  - Non-deployable customer application prevents relaunch from completing normally

Upgrade test

- So far we've talked about new servers
  - Use case: Your service is growing.
- Also need to test upgrading existing servers
  - Use case: You add a feature.
- The Upgrade Test Dance
  - Launch servers in test environment on current production code
  - Run smoke tests to ensure system is operating
  - Upgrade servers to latest development code
    - Requires a fully automated upgrade process
  - Run build, functional, and application tests from development code

Upgrade release process

- Puppet cannot orchestrate all upgrades
  - Rolling upgrade across HA clusters
  - Server type upgrade order requirements
  - Post-release tasks
    - ex: Uninstall package X once all servers are upgraded
  - Devs document release procedure with the commit
  - Different devs run the release on an internal-use installation
- Sample failure
  - nginx failed to restart after version upgrade because prod server has more domain names than test

Server builds: continuous imaging

- Remember: Always launch from a reference image. No evolved images!
- Building servers from scratch can be slow
- Automate pre-built images from development branch
  - Speeds intra-day tests, reduces MTTR in prod
- You will hit unexpected bumps in the road
- Sample failure
  - MySQL server, EBS, and init.d

Continuous Imaging image tests

- Create development images nightly
- Create per-branch images at release
- Run system tests on *both* base and pre-built images
- Test upgrade from per-branch to development pre-built images

Testing in parallel

- Infrastructure system tests are slow
- Run them in parallel
  - Workers may alter server-wide behavior (e.g. kill Apache)
  - Each worker needs an isolated set of servers
  - Workers that break their servers need to self destruct, or they will cause false failures
- Optimize running time
  - Add more workers
  - Reduce setup time
  - Run the slower tests first

Management issues

Who writes the tests?

- Our tests are as, or more, complex than the product
  - Tests often take longer to write
- Subtle cases require white-box testing
  - Triggering specific failure scenarios requires understanding OS and code details together
- First try: QA department
  - Did not work, they could not keep up or go deep
- Now: Engineering
  - Every dev writes unit and system tests for their own code

Who fixes the tests?

- Infrastructure system tests are fragile
  - The damn things break for every little bug!
  - … and every race condition imaginable
  - … and every cosmic ray
- Code reviews require a “passing” run
  - Author must analyze any failures, confirm they are unrelated, and refer to or open a ticket for it
- Bugs often only occur post-commit
- Permanent, rotating team handles failures
  - Authority to revert any commit causing a failure
  - Usually it is easier to fix it instead

Who invests in the tests?

- Management *must* accept that infrastructure system tests are
  - hard
  - time-consuming
  - essential
  - worth it
- Under-investing will bite you badly
  - “If it isn’t tested, it doesn’t work.”
  - It *will* fail, at the worst possible time

Questions?

- Barry Jaspan, barry.jaspan@acquia.com

- Please evaluate this session!

- Acquia is hiring!
  - Boston, New York, Portland
  - Europe! Australia!!!
  - wherever you are