The DevOps Handbook – Part 3 The First Way – The Technical Practices of Flow

1. Introduction
   1. Goal – Enable & sustain fast flow of work by implementing continuous delivery
      1. Create the foundation of our deployment pipeline
      2. Enabling fast & reliable automated testing
      3. Enabling & practicing continuous integration & testing
      4. Automating, enabling, and architecting for low-risk releases
   2. Integrating objectives of QA & Operations to improve outcomes
2. Ch. 9 – Create the Foundations of Our Deployment Pipeline
   1. Enterprise Data Warehouse program by Em Campbell-Pretty - $200M, All streams of work were significantly behind schedule. Surprising discovery: only 50% of the source code in Dev/Test environments matched Prod. They fixed forward, but changes not put back into version control. Focused on version control and automated environment creation – time reduced from 8 weeks to 1 day
   2. ENABLE ON-DEMAND CREATION OF DEV, TEST, AND PRODUCTION ENVRIONMENTS
      1. Major contributing cause of issues stems from releases representing the first time we see how an application behaves in a production-like environment
      2. Don’t just document the environment specifications
         1. Create a common build mechanism to create it on demand
         2. Codified in automated environment build process
      3. Environments will be stable, reliable, consistent, & secure
   3. CREATE OUR SINGLE REPOSITORY OF TRUTH FOR THE ENTIRE SYSTEM
      1. ALL parts (code & environments) of the system are shared in a version control repository
      2. Version control is for everyone in the value stream
      3. Everything, everything, everything is checked into version control
         1. Application code & dependencies
         2. Environment scripts & creation tools
         3. DB scripts and reference data
         4. Containers
         5. Automated tests
         6. Project artifacts – documentation, procedures, etc.
         7. Application configuration files
         8. This also includes pre-production and build processes
         9. Tools
      4. 2014 State of DevOps Report – use of version control by Ops was the highest predictor of both IT performance & organizational performance
   4. MAKE INFRASTRUCTURE EASIER TO REBUILD THAN TO REPAIR
      1. Quickly building enables quickly re-creating on demand rather than repairing
      2. Bill Baker, Microsoft distinguished engineer – We used to treat servers like pets: “You name them and when they get sick, you nurse them back to health. [Now] servers are [treated] like cattle. You number them and when they get sick, you shoot them.”
      3. Ensure consistency across all environments (Prod, pre-prod(s), dev, & new)
      4. Immutable Infrastructure
         1. Once created no changes are allowed
         2. Changes must be put into version control and then new infrastructure is created
         3. To prevent variance:
            1. Disable remote logins to productions
            2. Routinely kill and replace production instances
      5. Keep developers’ environments most current
   5. MODIFY OUR DEFINITION OF DEVELOPMENT “DONE” TO INCLUDE RUNNNING IN PRODUCTION-LIKE ENVIRONMENTS
      1. In general, the longer the interval between deployment, the worse the outcome
      2. Ensure Dev & QA routinely integrate with prod-like environments at increasingly frequent intervals
      3. Demonstrated in a production-like environment
      4. Ideally, use the same monitoring, logging, and other tools in pre-production environments as in production
      5. Dev & Ops gain shared mastery of application & environment interactions
3. Ch. 10 Enable Fast and Reliable Automated Testing
   1. Undesired outcomes result from finding and fixes errors in a separate test phase.
   2. Delayed feedback fades the cause & effect link of the error; now requires firefighting and archaeology.
   3. Worst effect – lose the ability to learn from mistakes and diminish integrating that learning into future work
   4. Google Web Server (GWS) team was struggling with changes – Hard line: no changes would be accepted into GWS without accompanying automated tests. Spread Testing Grouplet team approach. Google – single, share repository of billions of files. 50% of code is changed each month. 40K code commits/day, 120K automated test suites, 75M tests cases/day
   5. CONTINUOUSLY BUILD, TEST, AND INTEGRATE OUR CODE AND ENVIRONMENTS
      1. Create automated test suites to increase frequency of integration and testing of code & environments from periodic to continuously
      2. Create build & test processes that run in dedicated environments enables:
         1. Enables build & tests to run all time, independent of developers
         2. Segregated processes so we know the dependencies – eliminates “worked on my machine”
         3. Package the application to enable repeatable installation and configuration into an environment
         4. Environments can be more production-like in a consistent and repeatable way
      3. Building a deployment pipeline
         1. Commit stage – build & package software, runs automated unit tests, and code quality (static analysis, coverage, style, etc.)
         2. Package the code only once – use the same packages throughout the pipeline
         3. Captures the history of events for a build, test results, environments deployed to, etc.
            1. Use to support audits and compliance
      4. Create out continuous integration practices
         1. Comprehensive and reliable automated tests to validate deployable state
         2. Culture that “stops the entire production line” when validation tests fail
         3. Developers working in small batches on trunk rather than long-lived feature branches
   6. BUILD A FAST AND RELIABLE AUTOMATED VALIDATION TEST SUITE
      1. Slow and periodic feedback kills – larger batch sizes compound effort needed to identify causes
      2. Unit tests – test a single method, class, or function in isolation to ensure operates as designed
      3. Acceptance tests – test application as a whole that functionality operates as designed and regressions haven’t been introduced
      4. Integration tests – ensure correct interaction with other production applications and services
   7. CATCH ERRORS AS EARLY IN OUR AUTOMATED TESTING AS POSSIBLE
      1. A test suite’s goal is to find the error as early as possible, in the fastest category as possible
      2. Later stages require more scarce and complex environment to detect and reproduce issues
      3. When an error is found, create a test in an earlier, faster stage to detect the error
      4. Ideal vs. Non-Ideal Testing Pyramids
   8. ENSURE TESTS RUN QUICKLY (IN PARALLEL, IF NECESSARY)
      1. Design tests to run in parallel
      2. Enable different test categories to run in parallel
      3. Generally, limit manual testing if automated tests haven’t passed
      4. For resource intensive suites, performance tests, IA, execute as frequently as possible
   9. WRITE OUR AUTOMATED TESTS BEFORE WE WRITE THE CODE (“TEST DRIVEN DEVELOPMENT”)
      1. TDD – Kent Beck as part of Extreme Programming
         1. Ensure the tests fail – “Write a test for the next bit of functionality you want to add”
         2. Ensure the tests pass – “Write the functional code until the test passes”
         3. “Refactor both new and old code to make it well structured”
   10. AUTOMATE AS MANY OF OUR MANUAL TESTS AS POSSIBLE
       1. Reduce the reliance on manual testing
       2. Elisabeth Hendrickson – “Although testing can be automated, creating quality cannot. To have humans executing tests that should be automated is a waste of human potential.”
       3. Aggressively resolve unreliable tests and false positives
       4. Focus on automating tests that genuinely validate business goals
   11. INTEGRATE PERFORMANCE TESTING INTO OUR TEST SUITE
       1. Discovering performance issues in production is too late
       2. Validate performance with automated tests across the entire application stack as part of the deployment pipeline
       3. Creating performance testing environments can easily be more complex than the application itself
       4. Log performance results and evaluate compared to previous results
   12. INTEGRATE NON-FUNCTIONAL REQUIREMENTS TESTING INTO OUR TEST SUITE
       1. Incorporate automate tests to validate the “-ilities” that are important (availability, capacity, security, etc.)
       2. Incorporate security hardening testing and evaluation
   13. PULL OUR ANDON CORD WHEN THE DEPLOYMENT PIPELINE BREAKS
       1. Keep the build in a green state
       2. When a change causes the build or automated tests to fail:
          1. No new work is allowed to enter the system until the problem is fixed.
          2. Bring in whatever help is needed to resolve the problem
       3. Prioritize organizational/team goals over individual goals
          1. The value stream is a working application, not just working people
       4. Later stages may not necessarily stop all new work as long as responsibility for immediately resolving the issue is addressed
       5. Changing human behaviors and incentives is more challenging than the technical aspects
   14. WHY WE NEED TO PULL THE ANDON CORD
       1. If not, it becomes increasing difficult to get back to a deployable state
       2. If not, undoes the work done to get to a known workable state
4. Ch. 11 Enable and Practice Continuous Integration
   1. Branching – Pros & Cons
      1. Pro – Allows work in parallel while minimizing potential breaking impacts to trunk
      2. Con – Branch work, in isolation, becomes increasingly difficult to integrate and merge into trunk
      3. Integration problem cause significant rework and often cause additional delays due to integrating late in the lifecycle. Downward spiral of pain
   2. HP LaserJet Firmware –
      1. Before CI: 2 releases per year. 5% of effort supporting new features, 20% on detailed planning, 25% on porting code amongst branches, 10% integrating, 15% manual testing.
      2. Implemented Trunk-based development & CI
      3. After CI: 40% effort on new features, 40% decrease in development costs, 140% increase in programs under development, 78% reduction in cost per program
   3. SMALL BATCH DEVELOPMENT AND WHAT HAPPENS WHEN WE COMMIT CODE TO TRUNK INFREQUENTLY
      1. Branching Strategies
         1. Optimize for individual productivity – everyone works independently, merging and collaboration extremely difficult
         2. Optimize for team productivity – everyone works in common area, simpler commits, but each commit can break the entire project
      2. When merging is difficult, less able to refactor due to potential rework and merge problems
         1. Cross-cutting dependency improvements often provide high-payoffs
   4. ADOPT TRUNK-BASED DEVELOPMENT PRACTICES
      1. CI and Trunk-based development (TBD) are countermeasures to large batch size
      2. TBD enables:
         1. Frequent commits
         2. Runs of automated tests of the integrated system as a whole
         3. Work towards single-piece flow
      3. Trunk & state of Deployment Pipeline represents & communicates the current state of the system
      4. Modified Definition of Done – “At the end of each development interval, we must have integrated, tested, working, and potentially shippable code, demonstrated in a production-like environment, **created from trunk using a one-click process, and validated with automated tests.**”
      5. 2015 State of DevOps Report – trunk-based development predicts higher throughput and better stability, and even higher job satisfaction and lower rates of burnout.
5. Ch. 12 Automate and Enable Low-Risk Releases
   1. Just like we reduce batch size and increase frequency of feedback during development and testing, we now apply same concepts to releasing to production.
   2. Longer times between deployments accumulate larger differences and increased batch sizes
   3. Focus on reducing the friction with production deployments to be performed frequently and easily
   4. AUTOMATE OUR DEPLOYMENT PROCESS
      1. Begin with documenting the steps for current deployment process
      2. Simplify and automate through the steps
         1. Packaging code in method suitable for deployment
         2. Create pre-configured VM, containers, etc.
         3. Automate deployment & configuration of middleware
         4. Copying packages to production servers
         5. Restarting VMs, containers, applications, etc.
         6. Generating configuration files
         7. Run automated smoke tests
         8. Running test procedures
         9. Scripting & automating DB migrations
      3. Focus on areas to reduce lead times and handoffs
      4. New requirements for the deployment pipeline
         1. Deploying the same way to every environment (begin practicing how to deploy to production in development)
         2. Smoke testing our deployments – test connections to supporting services and systems, run sample data/transaction tests, fail deployment if needed
         3. Ensure we maintain consistent environments – continually ensure all environments are synchronized and consistent
   5. ENABLE AUTOMATED SELF-SERVICE DEPLOYMENTS
      1. As a result of compliance, oversight, and control needs separate Operations groups have emerged
      2. Widely accepted practice to “reduce the risk of production outages and fraud”
      3. DevOps goal – shift reliance from separate groups to other control mechanisms to mitigate risks more effectively using
         1. Automated testing
         2. Automated deployment
         3. Peer reviews
      4. 2013 State of DevOps Report – no statistically significant difference in change success rates between organizations where Development deployed code and those where Operations deployed code
      5. Enable fast flow through common promotion process anyone can run without manual steps and handoffs
   6. INTEGRATE CODE DEPLOYMENT INTO THE DEPLOYMENT PIPELINE
      1. Ensures packages created by CI are suitable for deployment
      2. Shows readiness of production environments at a glance
      3. Provides push-button, self-service for any suitable version into production
      4. Record automatically for auditing who and what commands were run
      5. Run smoke test to validate system behavior
      6. Provides fast feedback
      7. 2014 State of DevOps Report – high performers had deployment lead times measured in minutes or hours; low performers had deployment lead times measured in months.
   7. DECOUPLE DEPLOYMENTS FROM RELEASES
      1. Deployment & Release are distinct with different purposes
      2. Deployment – installing a specified version to a given environment
      3. Release – when a feature is made available to customers.
         1. Release of functionality should not require changing our application code
      4. By separating the actions, release becomes a business decisions, not a technical decision
      5. Environment-based Release Pattern – multiple environments available, only one is receiving live traffic. Release performed by moving traffic to the new environment, i.e. blue-green deployments
      6. Application-based Release Pattern – modify application to selectively release functionality by small configuration changes. Feature flags, dark launching
   8. ENVIRONMENT-BASED RELEASE PATTERNS
      1. Enables deployments during normal business hours
      2. Blue-Green Deployment
         1. Strategy
            1. Multiple environments with only 1 serving traffic
            2. Deploy to inactive environment, test,
            3. Swing traffic to new environment, Rollback by swinging traffic back
         2. Database changes
            1. Create two DB – put one in read only mode, backup & restore to new environment, swing traffic
            2. Decouple DB changes from application changes

Make only additive changes, never mutate existing

Remove application assumptions about DB version

* + 1. The Canary & Cluster Immune System Release Patterns
       1. Canary Strategy
          1. Automates the release process of promoting to larger and more critical environments
          2. Successful application behavior moves to the next group; failure stops or rolls back
       2. Cluster Immune Strategy
          1. Expands upon Canary. Links production monitoring with release process to automatically roll back when metrics deviations, outside of expected ranges, are detected
  1. APPLICATION-BASED PATTERNS TO ENABLE SAFER RELEASES
     1. Implement Feature Toggles
        1. Selectively enable/disable features without code deployments
        2. Wrap UI or application logic in conditional statements or strategy patterns
        3. Support easy roll back
        4. Support graceful performance degradation
        5. Increase resilience through SOA
     2. Perform Dark Launches
        1. Deploy features to production without releasing to users
        2. Perform background testing and risk-reduction evaluation with production loads
        3. Enable progressive rollouts
        4. Eliminates big bang, have already evaluated for considerable time before releasing

1. Ch. 13 Architect for Low-Risk Releases
   1. Principle of Evolutionary Architecture – Jez Humble – “any successful product or organization will necessarily evolve over its life cycle.”
   2. AN ARCHITECTURE THAT ENABLES PRODUCTIVITY, TESTABILITY, AND SAFETY
      1. Loosely-coupled
      2. Well-defined interfaces that enforce how modules connect with each other
   3. ARCHITECTURAL ARCHETYPES: MONOLITHS VS. MICROSERVICES
      1. Monoliths are often best early in a product life cycle
      2. Need to evolve based upon new constraints, goals, and functionality
      3. What works at scale 1X rarely works at scale 10X or 100X
   4. USE THE STRANGLER APPLICATION PATTERN TO SAFELY EVOLVE OUR ENTERPRISE ARCHITECTURE
      1. Coined by Martin Fowler in 2004
      2. Strangler Application
         1. Put existing functionality behind an API
         2. New functionality implemented in new services with new architecture
         3. Make calls to old system when necessary
         4. Repeat for next critical areas of the enterprise