The DevOps Handbook – Part 4: The Second Way – The Technical Practices of Feedback

1. Introduction
   1. Goal – Implement fast feedback loops
      1. Enable working towards shared goals
      2. See problems as they occur
      3. Enable quick detection & recovery
      4. Ensure features operate as intended and achieve organizational goals
2. Ch. 14 – Create Telemetry to Enable Seeing and Solving Problems
   1. Fact – Things will go wrong in Operations!
      1. 2001 Microsoft Operation Framework study found the best-performing organization were better at diagnosing & fixing service incidents.
         1. “Culture of Causality”
         2. Used disciplined approach to solving problems using telemetry to understand contributing factors and focus problem-solving.
      2. Telemetry – An automated communications process by which measurements and other data are collected at remote points and are subsequently transmitted to receiving equipment for monitoring
         1. Create telemetry in application & environments (to include production, pre-production, and CD pipeline)
      3. Ian Malpass, Etsy – “If Engineering at Etsy has a religion, it’s the Church of Graphs. If it moves, we track it. Sometimes we’ll draw a graph of something that isn’t moving yet, just in case it decides to make a run for it…Tracking everything is key to moving fast, but the only way to do it is to make tracking anything easy…We enable engineers to track what they need to track, at the drop of a hat, without requiring time-sucking configuration changes or complicated processes.”
      4. 2015 State of DevOps Report – high performers had MTTR 168x faster than low performers
   2. CREATE OUR CENTRALIZED TELEMETRY INFRASTRUCTURE
      1. Remove the silos of information – Developers don’t just log what’s interesting to development. Operations don’t just monitor what’s up or down.
      2. Modern Monitoring architecture
         1. Data Collection at business logic, application, & environments layer
            1. Events, logs, & metrics
            2. Common service to centralize, rotate, and delete
         2. Event router responsible for storing our events and metrics
            1. Enables visualization, trending, alerting, & other good things
            2. Enable threshold-based alerting & health checks
      3. Transform logs/events into metrics to enable statistical analysis
      4. Adrian Cockcraft – “Monitoring is so important that our monitoring systems need to be more available and scalable than the systems being monitored.”
   3. CREATE APPLICATION LOGGING TELEMETRY THAT HELPS PRODUCTION
      1. Dev & Ops create production telemetry as part of their daily work
      2. If it was important enough for an engineer to implement, it is certainly important enough to generate enough production telemetry so that we can confirm that is it operating as designed and outcomes are being achieved.
      3. Logging Levels
         1. Debug – anything that happens in the program
         2. Info – user driven actions or system specific
         3. Warn – conditions that could become an error and will likely trigger an alert
         4. Error – error conditions such as API failures, internal issues
         5. Fatal – forces a termination
      4. Examples of potentially significant events (Gartner’s GTP Security & Risk Management group)
         1. Authentication/authorization decisions
         2. System and data access
         3. System and application changes, especially privileged changes
         4. Data changes (CRUD)
         5. Invalid input, possible malicious injections or threats
         6. Resources (RAM, disk, CPU, bandwidth, and others with hard or soft limits)
         7. Health & availability
         8. Startups & shutdowns
         9. Faults & errors
         10. Circuit breaker trips
         11. Delays
         12. Backup success/failure
   4. USE TELEMETRY TO GUIDE PROBLEM-SOLVING
      1. Don’t practice “Mean Time until Declared Innocent” – culture of blame encourages limited documentation, hiding telemetry, deflection
         1. Prevents creation of institutional knowledge about incidents and learning needed to prevent them in the future
      2. Use for fact-based problem-solving
   5. ENABLE CREATION OF PRODUCTION METRICS AS PART OF DAILY WORK
      1. Create infrastructure to make it easy for Dev or Ops to create telemetry for functions built
      2. Generate graphs with overlays of production changes
      3. Tools – StatsD, JMX, codahale, New Relic, Prometheus, etc.
   6. CREATE SELF-SERVICE ACCESS TO TELEMETRY AND INFORMATION RADIATORS
      1. Spread the information – anyone who wants or needs the information can readily access it without production access or other privileged accounts.
      2. Make it highly visible for everyone in the value stream
      3. Promotes responsibility
         1. The team has nothing to hide from its visitors (customers, stakeholders)
         2. The team has nothing to hide from itself: it acknowledges and confronts problems
   7. FIND AND FILL ANY TELEMETRY GAPS
      1. Expand metrics from business, application, infrastructure, client software, and deployment pipeline levels
         1. With every production incident identify missing telemetry that could help detect and recover faster
      2. Application & Business Metrics
         1. Measure extent application is helping achieving organizational goals
         2. Business metrics typically represent part of the customer acquisition funnel
         3. Goal – Actionable metrics that inform changes. If not actionable, likely a vanity metric.
         4. Reinforces that analyzing customer usage is part of daily work and helps understand how work contributes to organizational goals
      3. Infrastructure Metrics
         1. Quickly determine whether infrastructure is contributing to the cause of the issue
         2. Focus on dynamically registered & discovered supporting tools rather than static
         3. Graph business metrics with infrastructure metrics to provide deeper context
      4. Overlaying other relevant information onto our metrics
         1. All production deployments should be visible on graphs
         2. Similarly useful are maintenance, backups, and other activities.
3. Ch. 15 – Analyze Telemetry to Better Anticipate Problems and Achieve Goals
   1. USE MEANS AND STANDARD DEVIATIONS TO DETECT POTENTIAL PROBLEMS
      1. Easy to filter to identify when a metric is significantly different from its norm.
      2. Minimizing false positive is critical to prevent alert fatigue
      3. Statistical measures help set thresholds without need to statically define each one.
   2. INSTRUMENT AND ALERT ON UNDESIRED OUTCOMES
      1. An approach
         1. Analyze most severe incident in recent past.
         2. Create list of telemetry that could have enabled earlier detection and diagnosis as well as confirmation
         3. For each metric, configure alerts when metric deviates significantly from mean
         4. Repeat for weaker signals
   3. PROBLEMS THAT ARISE WHEN OUR TELEMETRY DATA HAS NON-GAUSSIAN DISTRIBUTION
      1. Many operational data sets are non-Gaussian (Bell curve) so standard deviations will likely over or under alert
   4. USING ANOMALY DETECTION TECHNIQUES
      1. Anomaly detection is “the search for items or events which do not conform to an expected pattern.”
      2. Smoothing – suitable for time series data. Using moving averages to reduce short-term fluctuations and highlight longer-term trends.
      3. More exotic - Fast Fourier Transforms or Kolmogorov-Smirnov
4. Ch. 16 – Enable Feedback So Development and Operation Can Safely Deploy Code
   1. USE TELEMETRY TO MAKE DEPLOYMENTS SAFER
      1. Actively monitor production telemetry whenever anyone performs deployment; focus on the metrics associated with the feature.
      2. If negative results, ideally feature toggle off and fix-forward or use other Blue-Green, Canary release patterns
   2. DEV SHARES PAGER ROTATION DUTIES WITH OPS
      1. Complex systems will inevitably have unexpected problems
      2. Prevent upstream work from locally optimizing at the expense of the entire value stream – Everyone (development, managers, architects, ops, etc.) in the value stream shares responsibility for handling operational incidents and gets visceral feedback on upstream decisions. (Pedro Canahuati, Facebook Director of Production Engineering)
      3. Find the proper balance between fixing production issues and new features development – feature is only “done” when it’s performing as expected in production
   3. HAVE DEVELOPERS FOLLOW WORK DOWNSTREAM
      1. Contextual Inquiry – product team watches customer use the application in their natural environment. Helps identify compensatory behaviors and workarounds driven by un/usability
      2. Same benefits applies with internal customers (infrastructure, operations, etc.)
      3. Gene Kim – “One of the worst moments of my professional career was in 2006 when I spent an entire morning watching one of our customers use our product. I was watching him perform an operation that we expected customers to do weekly, and, to our extreme horror, we discovered that it required sixty-three clicks. This person kept apologizing, saying things like, “Sorry, there’s probably a better way to do this.” Unfortunately, there wasn’t a better way to do that operation.”
   4. HAVE DEVELOPERS INITIALLY SELF-MANAGE THEIR PRODUCTION SERVICE
      1. Google has development groups self-manage their services in production before they become eligible for Ops group to manage.
         1. Define launch requirements (Launch Readiness Reviews, LRR) that must be met before interaction with real customers
            1. Validates effective monitoring in place, reliable deployments, and architecturally suitable to fast, frequent change
            2. Identifies any regulatory or contractual compliance objectives
         2. Ops engineers consult with teams to help make production-ready services (Hand-Off Readiness Review, HRR)
      2. Operations has a service handback mechanism to the development team when production services become fragile
5. Ch. 17 – Integrate Hypothesis-Driven Development and A/B Testing into Our Daily Work
   1. INTRODUCTION
      1. Jez Humble, “The most inefficient way to test a business model or product idea is to build the complete product to see whether the predicted demand actually exists.”
      2. How to build a feature:
         1. Ask, “Should we build it, and why?”
         2. Perform cheapest, fastest experiment possible to validate whether the feature will achieve desired outcomes.
         3. Repeat
      3. Intuit’s rampant innovation culture – went from 7 experiments/year to 165 experiments during the 3 month US tax season in 2010 with website conversion rates up 50%
   2. A BRIEF HISTORY OF A/B TESTING
      1. Pioneered in direct response marketing
      2. Early efforts required sending thousands of postcards/flyers and asking consumer for follow-up action (phone call, return postcard, etc.)
         1. Campaigns would modify text, layouts, packaging, etc.
         2. Very expensive
   3. INTEGRATING A/B TESTING INTO OUR FEATURE TESTING
      1. Random subset of users shown differing versions. User groups define a cohort. Statistical analysis of cohort’s outcomes from differing versions used to identify statistically significant difference in behaviors and identify a causal link
      2. Ronny Kohavi, Distinguished Engineer & General Manager of the Analysis and Experimentation group at Microsoft: “evaluating well-designed and executed experiments that were designed to improve a key metric, only about one-third were successful at improving the key metric!”
      3. A/B testing helps reduce zero or negative value add features to the baseline that also increase maintenance costs and identify opportunity costs.
   4. INTEGRATE A/B TESTING INTO OUR RELEASE
      1. A/B testing requires fast CD to support
      2. Use feature toggles to control experiments, cohort creation, etc.
      3. Use telemetry to measure outcomes
      4. Etsy open-sourced their experimentation framework – Feature API
   5. INTEGRATE A/B TESTING INTO OUR FEATURE PLANNING
      1. Product owners should consider each feature as a hypothesis and use real users to prove/disprove the hypothesis
         1. Barry O’Reilly, Lean Enterprise describes as: **We Believe that** <some feature> **Will Result in** <business metric improvement>. **We Will Have Confidence To Proceed When** <expected change in business metric>
6. Ch. 18 – Create Review and Coordination Process to Increase Quality of Our Current Work
   1. Goal – enable Development and Operations to reduce risk before production changes are made. Keep the reviews and approvals close to those whom are knowledgeable, not far removed groups.
   2. Shift away from periodic inspections and approvals; move to integrated peer reviews performed continually as part of daily work
   3. THE DANGERS OF CHANGE APPROVAL PROCESSES
      1. Knight Capital - $440M trading loss
      2. Counterfactual Narratives
         1. Accident due to change control failure – seems valid since we can imagine better control practices could have detected the risk earlier prevent the issue or could have taken steps to detect and recover faster
         2. Accident due to testing failure – seems valid since better testing practices could have mitigated
      3. In low-trust, command & control cultures, change control often results in worse outcomes
   4. POTENTIAL DANGERS OF “OVERLY CONTROLLING CHANGES”
      1. Traditional change controls
         1. Lead to long lead times and diminish effectiveness of feedback from the deployment process
         2. Increase friction by adding more steps/approvals which tends to increase batch size
      2. Toyota Production System – “people closest to a problem typically know the most about it.”
      3. 2014 State of DevOps Report – high performing organizations relied more on peer review and less on external approval of change. The more organizations relied on change approvals, the worse their IT performance for stability (MTTR, change fail rate) and throughput (deployment lead time, deployment frequency)
   5. ENABLE COORDINATION AND SCHEDULING OF CHANGES
      1. Loosely coupled architectures enable greater team independence and can likely succeed with more informal coordination approaches like chat rooms
      2. For deliberate change scheduling, gather representatives, not authorize changes, but to schedule and sequence.
   6. ENABLE PEER REVIEW OF CHANGES
      1. Require peer reviews, not external board approval
      2. Peer review help improve overall quality, provide cross-training, learning, and skill improvements.
      3. Keep batch sizes of reviews small
      4. 4 simple Peer Review Guidelines
         1. Everyone must have someone review their changes
         2. Everyone should monitor the commit stream of their fellow team members to identify issues
         3. Define changes which qualify as high-risk and require SME review
         4. If a change is too large to reason about, split it up
   7. POTENTIAL DANGERS OF DOING MORE MANUAL TESTING AND CHANGE FREEZES
      1. When testing failures occur, adding more testing could backfire, *especially if the testing is at the end of a project and/or manual*
         1. Manual testing is slower and more tedious
         2. Tends towards increasing batch sizes
      2. Fully integrate testing into daily work – enable flow, more automated testing
   8. ENABLE PAIR PROGRAMMING TO IMPROVE ALL OUR CHANGES
      1. Pair Programming – two engineers working together at the same workstation. Popularized by XP & Agile.
         1. One engineer is the driver – actually writing code
         2. Other engineer is the navigator/observer/pointer – reviewing work, considers strategic direction, generating ideas for improvement and the next steps
         3. Supports ad-hoc cross training
      2. Dr. Laurie Williams, 2001 study – “paired programmers are 15% slower than two independent individual programmers, while ‘error-free’ code increased from 70% to 85%. Since testing and debugging are often many times more costly than initial programming, this is an impressive result. Pairs typically consider more design alternatives than programmers working alone and arrive at simpler, more maintainable designs; they also catch design defects early.”
   9. EVALUATING THE EFFECTIVENESS OF PULL REQUEST PROCESSES
      1. Review production outages and examine the peer review process for the relevant changes
      2. Sampling and examining pull requests from larger population or those relevant to production incidents
         1. Bad pull requests – typically doesn’t have enough context for the reader, little documentation of the changes intended outcome, or explanation of the thought process behind the change
         2. Good pull requests – sufficient detail on why the change is made, how the change was made, and identified risks and resulting countermeasures
   10. FEARLESSLY CUT BUREAUCRATIC PROCESSES
       1. Adrian Cockcroft – “A great metric to publish widely is how many meetings and work tickets are mandatory to perform a release – the goal is to relentlessly reduce the effort required for engineers to perform work and deliver it to the customer.”
       2. Continuously question and examine existing processes to streamline and improve
   11. CONCLUSION
       1. Create the conditions that enable change implementers to fully own the quality of their change
       2. Create ever-safer systems of work
       3. John Allspaw – A newly hired junior engineer asked permission to deploy a small HTML change. Allspaw responded, “I don’t know; is it? Did you have someone review your change? Do you know who the best person to ask is for changes of this type? Did you do everything you absolutely could to assure yourself that this change operates in production as designed? If you did, then don’t ask me – just make the change!”