Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy)

Steve Neely and Steve Stolt
Rally Software
3333 Walnut Street
Boulder, Colorado 80301
Email: {sneely, stolt}@rallydev.com

***Abstract*—Rally Software transitioned from shipping code every eight-weeks, with time-boxed Scrum sprints, to a model of continuous delivery with Kanban. The team encountered complex challenges with their build systems, automated test suites, customer enablement, and internal communication. But there was light at the end of the tunnel — greater control and flexibility over feature releases, incremental delivery of value, lower risks, fewer defects, easier on-boarding of new developers, less off-hours work, and a considerable uptick in confidence.**

This experience report describes the journey to continuous delivery with the aim that others can learn from our mistakes and get their teams deploying more frequently. We will describe and contrast this transition from the business (product management) and engineering perspectives.

I. Introduction

When Rally Software was founded in April 2002 the engineering strategy was to ship code early and often. The company founders had been successfully practicing Agile and Scrum, with its well-known patterns of story planning, sprints, daily stand-ups and retrospectives. They adopted an eight week release cadence that propelled the online software as a service (SaaS) product forward for more than seven years. Then we decided to change everything.

Under the influence of Anderson's book on Kanban [1], Humble's writings on continuous delivery [2] and on-site training by Don Reinertsen [3], we adopted a model of continuous flow, using Scrum-ban with the vision of releasing new code on-demand and at-will. This was our first step towards continuous delivery.

We did not know how long this journey would take nor how complex it could become. However, with careful planning and designated exit strategies we knew there was a path of *Kaizen*, or continual improvement, which would result in a higher quality organization.

This paper presents our experiences in moving from an eight week release cadence to continuous delivery. We describe and contrast this transition from the business and engineering perspectives as we provide motivation for *why* and experiential advice on *how* to adopt continuous delivery in other teams.

II. What is Continuous Delivery?

Officially, we describe continuous delivery as the ability to release software whenever we want. This could be weekly or daily deployments to production; it could mean every check-in goes straight to production. The *frequency* is not our deciding factor. It is the ability to deploy *at will*.

Covertly, we know that the engineering discipline required to support such a model must be strong. Our systems must be finely tuned and comprehensively governed by automation, with thorough monitoring and testing frameworks.

The engineering challenge to support per-commit production deployments is readily savored by technical teams. We engineers love a challenge. There may be an initial hesitation by some in the organization who hold the belief that "letting software bake" makes it somehow safer, but this fallacy is quickly vanquished as the support system for continuous delivery is realized. The improved test coverage, rapid feedback cycles, scrutiny of monitoring systems, and fast rollback mechanisms result in a far safer environment for shipping code.

But it is not free; it is not painless.

It is important to have the right level of sponsorship before you begin. A champion trumpeting the the horn for continuous delivery is a key starting point but you must garner buy-in from the executive and senior management teams. When you present the vision to these stakeholders it is important to have the mission clarified. What are you trying to *achieve* through continuous delivery and *why*? You must set clear objectives and keep progress steering toward the right direction. There are many side paths, experiments and "shiny" things to distract you on the journey towards continuous delivery. It is easy to become distracted or waylaid by these.

Equally important is setting expectations and tracking appropriate metrics along the way. This helps you demonstrate progress over time. Depending on your existing infrastructure and legacy systems the switch to a flow-based model of software delivery will take varying amounts of time. Setting objectives, "mother strategies" and "true north" goals will help guide the whole team and provide anchor points for inspection and adaptation.

A. Our starting point

Rally had been in business for just over seven years and had built a successful business deploying code to our SaaS product every eight weeks. We executed planning-sprint cycles over seven of those weeks and spent one week "hardening" the code. This week was an extended *bug bash* when everyone

978-0-7695-5076-3/13 $26.00 © 2013 IEEE
DOI 10.1109/AGILE.2013.17

in the company clicked around the application trying to find defects in the new software. As defects were logged the engineering team worked frantically to burn down the defect backlog. It was a laborious and expensive week for the company.

Releases would go out on a Saturday morning, with scheduled application downtime for executing database migrations. Compressed WAR files would be manually copied across the network in preparation for the deployment process to begin. Engineering would be on call, with everyone watching the charts as the deployment took place via a mix of manual operations and scripts. If anything went wrong we could lose a Saturday to fixing the failure. Occasionally problems did not surface until Monday when user traffic spikes and so we would arrive early that morning as a precaution.

After a successful release Rally would celebrate. The event was recorded for history with a prized release sticker and the eight week cycle began again.

III. Why Continuous Delivery?

In our Scrum days, product management would identify major features in a release and everyone would plan to these features with a date in mind. Marketing knew what was coming up and could advertise accordingly, Sales could entice customers with the latest and greatest product features, and User Learning could prepare help documentation and tutorials in advance.

A downside of these eight week releases was that valuable shippable increments would often wait. Features completed early in the release are artificially delayed and lower priority defects (those that did not merit an special out-of-band release) would remain broken in production.

Worse, under pressure to deliver the major features identified at the beginning of the release, sometimes scope was cut back too aggressively. Features that needed one more week of work could not wait. They would be released as scheduled without the full functionality originally planned. We always had the best of intentions to return to these features but eight weeks is a long time; plans change.

A. Why would Engineering care?

Selling continuous delivery to our development team was relatively easy. The software engineers at Rally are eager to experiment with new technologies and methodologies. They understood that smaller batch sizes would lead to fewer defects in production as we limited the size of code changes and garnered fast feedback from our systems. When defects arise in a small batch system the focus on where to look is greatly narrowed since less code was shipped. An eight week corpus of code is a significant haystack to look through when problems appear.

Further, the flexibility of continuous delivery would allow Engineering to try out new ideas and roll them back if there is an issue. Waiting for long periods of time between deployments inhibits experimental behavior and encourages a more conservative mindset. Continuous delivery shortens the feedback cycle and keeps progress moving at pace.

Engineering also knew that shipping in small batches would result in fewer fire drills and hence less stress. Regular code deployment makes you practiced, comfortable and efficient. The "scary Saturdays" become a thing of the past.

Finally, Engineering had a sense of wanting to be the cool kids. We had read engineering blogs and talked to teams from other companies like Etsy, Amazon and Facebook about releasing continuously. We wanted to too. It would be a fun challenge.

B. Why would the business care?

After seven plus years of practicing and selling Agile with Scrum Rally was proud to be labeled an industry expert. Our product was successful and the company was rapidly growing. Rally has always been a company that embraces change and it was time to change.

Eight week release cycles are too long in the on-demand software world. To stay competitive you must be able to sell features that match or beat your peers. Releasing every two months is painful for a number of reasons: (1) when you miss a release you potentially have to wait another eight weeks to deliver your feature to customers; (2) technical complexity builds up over time: merging code, coordinating data migrations, and integrating long running branches is time consuming and error-prone. Rally's product managers understood that this introduces risk; (3) the "Scary Saturdays" required "all hands on deck" with *everyone* watching the charts. This included representatives from the business. Since we were only releasing code six times a year we had to carefully plan. Preparations were time consuming, detracting form other work, and we were not practiced at it.

Yet our product managers were somewhat ambivalent towards the full adoption of continuous delivery. They simply wanted to release code any time they wanted and did not care about every check-in going directly to production. However, they were very interested in trying kanban in place of Scrum and had been trying to convince Engineering to try kanban for some time. Here was a common ground here on which to build.

The business' argument was that kanban would empower the teams to work at an optimal workflow and allow them the latitude to deal with issues around technical debt. Engineering's argument to the business was that continuous delivery could allow them to release whenever, ship fixes whenever, and better support the incremental delivery of value.

Although Engineering was quick to vote in favor of a shorter release cadence the business was ultimately convinced by a more subtle argument: the process of getting to continuous delivery is *engineering driven* with a myriad of *positive side-effects*. Including: the requirement for tighter engineering discipline; fast, reliable and scalable test suites; full automation of deployments; comprehensive systems' monitoring; and fine-grained incremental delivery of value. Choosing to focus engineering effort on any of these was a clear win.

So both sides were convinced. We built kanban boards, practiced continuous flow and started to shorten our release cadence.

IV. Two stories of change

A. A story of change in Engineering: "Push direct to production? What?!"

A new hire and I were pair programming on a defect when he encountered continuous delivery for the first time. We identified the root cause of the defect, wrote a failing test and implemented the code resolution. When the code was ready to be pushed to the engineering git repository the new hire asked *"what next"*.

I explained that firstly our Jenkins continuous integration server would pick up the change in the code repo, run the unit and integration tests, and deploy our service to a test server. Secondly, a health check monitoring system would interrogate the test server to ensure that the service was running. Finally, deployr (our Ruby-based deployment system) would automatically deploy the code to production and ignitr would roll the service live. *"Deploy to production? Automatically? That's scary!"*, he exclaimed.

We talked about why he felt that the push to production was scary. The test we had written proved our code change fixed the defect and the comprehensive test suite would exercise the codebase with our change. The test deploy and monitoring would verify that the service started up correctly and the deploy to production mirrored that pipeline. Reassured by our conversation the new guy pushed the code and the automatic processes took over.

That afternoon we received an email from a customer saying that they had noticed the defect fix and wanted to say a huge thank you for resolving a pain point in the application. The new hire came over to me and with a big smile said *"this continuous delivery is hot stuff!"*

B. A story of business change: ready-fire-aim

As we mentioned earlier, Rally is a company the embraces change. We had executive management very excited about using kanban. Engineering had agreed because it gave them the opportunity to focus on continuous delivery with less process and fewer meetings. So, we fired our kanban bullet. We canceled the traditional Scrum ceremonies and moved our work to web-based kanban boards. At first everything was great. We had some executives excited about Engineering using kanban and we had some engineers excited about fewer meetings.

We soon realized that there is more to implementing kanban than having your work on a kanban board. You need a shared understanding of what columns are on the board and why, and agreements on when to move things from one column to the next. You need a shared understanding of what the work in progress (WIP) limits will be for each column. You need to have shared agreement on what will be done when a WIP limit is exceeded or when a card stays on the board for too long.

In hindsight, we should have created a more thoughtful plan and aimed *before* we fired the kanban bullet. This is ironic, because we have world class coaches that train other companies on these processes, and we did not leverage them internally. Our correction steering began with training all teams on kanban, and our workflow immediately improved.

V. What we did

The transition to continuous deployment requires a significant effort from engineering. This section begins by describing some of the activities and methodologies we employed. The end of the section describes where we started with the business and discusses a key issue on defining dates for release.

A. You do not have to go from eight to zero overnight

Jumping directly from eight week releases to a push-to-production strategy is clearly not a sensible approach. We began by shrinking our release cycles down to fortnightly, weekly, semiweekly and finally at-will. These steps took weeks or even months of preparatory work to get our deploy process streamlined and automated.

The step to "at will" was exercised by our running of a *thought experiment*. We would pseudorandomly pick builds during the day (by rolling a virtual die) and hold a "Roman Vote". The vote required a short meeting of representatives from each development team and stakeholders who would review code commits. The meeting attendees would "thumbs" up or "thumbs down" the release. If anyone gave a thumbs down that would trigger a discussion on what was blocking the release and how could we make sure that similar blockers would not hold back future real releases.

At regular retrospective meetings we would discuss releases from the previous period and identify what worked and what did not work. All the time we are taking incremental steps, measuring progress, and practicing a reflect, inspect and adapt cycle.

B. Documenting the engineering process

In Engineering we spent time documenting and fully understanding our process, from pulling a story to deployment and thereafter. We tried to tease apart each step in the development pipeline and work on scripts to remove manual steps. It was time to *"automate all the things"*.

At first we tried to deploy every week (even as a thought experiment), then twice a week, and finally at-will. When you cannot release code from a given commit it is important to identify what is preventing you from deploying now. Maybe it is a cold database migration that changes the structure of a database schema. Perhaps there unknown performance implications? Does the team feel that a load and performance test run is necessary?

C. Feature toggles

We use git for or source code management. Everyone works off a master branch and we rarely employ long running branches. Not even feature branches. Instead we use *feature toggles*.

The no-branch policy encourages small sized stories and frequent commits. We have felt the pain of merging long running branches too many times. Merge conflicts can take hours to resolve and it is all too easy to accidentally break the codebase. Git does an amazing job of automatic, even three-way or octopus, merging of branches but it does not always get it correct. This has introduced bugs in our application before and they can be difficult to track down.

Instead of feature branches we use in-code feature toggles [4]. The concept is fairly simple: you add a framework of conditional logic that hides code changes not ready for general release. We built an administration interface that allows these toggles to be switched on at different levels in the application. A feature can be toggled on for all users, an organization or an individual user (or users).

Feature toggles allow us to stage switching on code and monitor its effect on the system (behind closed doors in Engineering we liken it to "testing in production"). If the database load suddenly jumps through the roof or requests start to back up then we can toggle off the feature and return the system to its previous state. We can try out features in quieter periods and rollback if needed.

Testing feature toggles has been criticized as producing a combinatorial explosion of tests. The argument posits that you must factorially test with every combination of on/off for every toggle. In our experience this is not the case. Features behind toggles tend not to cross boundaries or interfere with each other. Further, the combinatorial criticism would be the same problem with feature branches, and with feature toggles it is simpler as you toggle on and off in test code. The only additional complexity in testing toggles is that you must be cognizant of what will be toggled on in production at the same time.

When a toggled story is ready for general release we write a following story to remove the toggles. If you forget to retire toggles your codebase becomes a mess of feature specific logic. This follow up story keeps our codebase clean and is specifically called out in our working agreements.

D. Dark and canary deployments

Dark deployments are services in production that process real traffic but have no impact on the running system. This is a mechanism for phased deployments and detecting problems in production before customers would be effected. By testing your service with actual traffic you reduce the risk of missing bugs that test traffic may not discover.

The method for standing up a dark deployment is to find a place in the code path where you can *tee* the incoming requests. Asynchronously send a copy of the requests to your dark service and log interactions. Care must be taken to ensure that the tee'd traffic is truly asynchronous and does not impact regular request flow. Use of programming constructs such as Futures or Actors with a library like Akka [5] will help with this.

Another technique you can use for testing with production traffic are Canary deployments. These are a single node in your multi-cluster system running new code that can be taken back down if it misbehaves. The metaphor is borrowed from *the canary in a coal mine*. If it dies, then get out and save the rest of the system. If you deploy a "canary" node you must ensure any modifications it makes to shared data are backwards compatible with the rest of the system.

E. Test planning

If you want to release code frequently you must remove manual steps. We used to rely on manual testing as a step before code could be shipped to our master branch for a production release. Quality Assurance (QA a.k.a. "test") had to check a box indicating that the code had been manually verified and had their stamp of approval for release. This is slow, error prone and not methodological.

But tester expertise and mindsets are essential. We like to think of the mind of a QA as a weird and wonderful place. Their professional experiences enable a different analysis of code and testers have an uncanny ability to discover edge case bugs in applications. As a developer working deep in the code it is easy to miss angles for test that QA can spot early on.

We continue to utilize tester expertise by changing our process to include a phase for test enumeration. This phase is *pre-development*, before an IDE has been opened or line of code written. *Test planning* is a pair (or tri-) exercise in which QA works with developers to document a comprehensive list of automated tests.

For stories that require modification of existing code we may go to the lengths of documenting existing tests before pulling the story for test planing. This provides visibility into the test coverage of the code, allows QA and developers to close gaps and gives the team a level of confidence that that part of the application has been exercised by the automated test suite.

The result of test planning and obtaining a fuller understanding of our automated test suites is that QA no longer needs to manually test the majority of our application for regression bugs.

F. Where did we start with the business?

We fell in love with toggles. When we were releasing every eight weeks, features would be mostly done when we released. When we started releasing much more often, this was not the case. How should we deal with 5% of a feature's software being released? In some cases 5% is not really visible to the user, and in most cases 5% of a feature is not worth making noise about. We knew we did not want to blast the users every time part of a feature got released. The solution was feature toggles. Building features behind a toggle provides us with the ability to toggle features on an off at will and on and off for specific customers. This gives the business control in the following ways:

- Stage a rollout - internal, beta customers, general availability
- Rollback - an unforeseen negative side-effect is encountered, turn the new feature off

- Market release - I want to enable a set of features together for a larger rollout timed to a conference

We use the term *"enablement"* internally to refer to the education around new features (both internally and externally). The Product Marketing team handles the bulk of enablement activities. They struggled in the beginning as we moved to continuous delivery.

The struggle centered around the fact that a consolidated enablement push every eight weeks was no longer an option. With eight week releases there was a lot to talk about. Large internal meetings were scheduled with Sales, Technical Account Managers, and Support teams to educate everyone about new features. However, when features are being released every few weeks, the value of these big meetings rapidly diminished. We needed a leaner approach. Big meetings got replaced by crisp emails. The information about new features is posted to internal wikis and blogs. The responsibility to learn about new features was pushed out to the Sales, Technical Account Managers, and Support Teams.

Externally, we used to hold a webinar every eight weeks describing the new features in a release. With continuous releases this has largely replaced by blogs. We also use more small callouts or "New" badges in the product to draw attention to new features. Today we enable in a more subtle ways, to not overload the customer. We save our voice for the larger features and larger venues.

G. Where are my dates?

Although we engaged executives with the push to kanban we made a crucial mistake by not performing a deeper analysis across all departments. The two most important stakeholders we skimmed over were Sales and Marketing.

The Sales teams had come to rely on our eight week releases. These gave them time to prepare for the new features and set expectations with existing customers. Marketing was used to ramping up to message the eight-week release, then starting work to prepare for the next release in eight more weeks. With continuous flow, Sales and Marketing were not quite sure when anything would be released.

This was particularly painful for the "product owner" role. Part of the product owner's role has been to buffer the team from distractions. Most of the Sales and Marketing teams respect that. So, since Sales and Marketing could not rely on feature *a, b* and *c* coming out with the next release in *x* weeks, they asked the product owner. But the product owner did not really know *exactly* when the next feature was going to come out. They would give vague and unsatisfying answers to Sales and Marketing when asked. As you can imagine this was not great for their relationships.

We eventually learned, that we need to increase the level of transparency with Sales and Marketing. We could not tell them what we were going to release in eight weeks then go away, pop up eight weeks later and say "ta-da, here are the features we promised." We had to provide mechanisms where they could track the status of work in real time, so they can plan accordingly.

We found it easier to maintain a calendar communication cadence with stakeholders. As a company we have annual planning and quarterly planning. Each product line has a monthly council to gather stakeholder feedback and to keep them current on what is being being built and what is coming next.

VI. Lessons Learned

We have implemented many changes to our processes with varying degrees of success and failure. In this section we highlight lessons learned — some from experiments and some from pain.

A. A story of warning — the day we broke production

Circumventing the checks and gates to release code is a *very* bad idea. We discovered this the hard way when we locked out all a small number of *beta testers* for about four hours one morning. This is a story of what *not* to do.

We were having problems deploying a significant code change because our test deployment kept failing in the pipeline. We decided that it was because there was bad data on the test system. This was incorrect. The actual problem was that our new code was incompatible with the old data format.

To get the tests to pass we deleted all the historical test data and the new code made it out to production. Then everything stopped working... the new code was not compatible with the production data. Fortunately, since we were in early *pre-release* for this particular service our mistake only affected our select beta customers (who were still a little mad at us).

The lesson learned from this is that *your tests, gates and checks are there for a reason*. Always run them with old data and never circumvent the process without a clear understanding of what you are doing. The ability to release quickly does not mean you should rush without full understanding.

B. Work life balance

An immediate observation in moving to a continuous delivery model is that everyone in Engineering becomes practiced at releasing code and has a safety net of comprehensive automated test suite with gates, checks and balances between a developer and production. Our development and the operations teams are happy that they no longer needed to work Saturdays to release software. The business is happy because they were not worried about an all-eggs-in-one-basket Saturday release failing or having to wait eight weeks for another release of functionality.

C. Move fast?

If you want to move fast, deploy with high frequency and delivery code continuously it is imperative that you know the state of your system. Monitor everything. We use a variety of monitoring tools like Splunk, Ganglia, Nagios and GDash. Everyone on the team watches when a deployment goes out because you need to quickly identify if your commit affects the production systems in a negative way.

The default approach to problems in production is to fix and roll forward. Since we are deploying small batches of code this is usually fairly easy to manage. Even at scale.

Rolling back a commit should be the last resort. Especially because other commits may have gone out on top of that commit and have muddied the codebase.

D. On-boarding

The Engineering organization discovered that it is easier to on-board new employees and get them up and running faster because we have built a safe-to-fail environment. That is, if existing software has good test coverage, new people can be more fearless and be productive earlier.

E. Livelihood

Be aware: your teams will find this scary (you might too!). The transition to a continuous delivery model can make stakeholders uncomfortable. The ability to push code directly to production does come with added responsibility — developers must be diligent in writing tests and monitoring the system. Everyone in the team adopts a operational aspect to their daily roles.

Members of our QA team were particularly worried about not having time to test code before it shipped. They needed to switch to a mindset that trusts the automatic tests to perform this task. This naturally led to the fear that they would be automated out of employment. This is absolutely not the case.

As mentioned earlier, before a story is coded we execute a "test planning" phase. This needs a professional QA mindset. QA does not just appear blindly at this stage. They will have spent hours reading stories in the backlog, performing research into the background of the story and brain storming testing angles.

With the decreased load from test automation some of our QA team have adopted a *TestOps* role. TestOps monitor our application with tools like Splunk and Ganglia to discover changes deep within the system. DevOps is a commonly talked about role in our community and we suspect that TestOps will become a familiar role in engineering teams of the future.

Operations teams also have to alter their mindsets to accommodate continuous delivery. There will no longer be scheduled deployment dates when the operations team copy code around data centers and execute startup scripts. Instead, they will be working to support the continuous delivery pipeline with SSL tunnels, firewall gateways and monitoring solutions.

F. Speed is difficult

At one time our GUI test suite took nine hours to run. Since a passing test run is a prerequisite for a production release this mean the fastest we can ever release is *nine hours*.

When test runs take this long people begin to ignore them. To resolve this one must break down the suite, optimize long running tests and parallelize. Our advice here, learnt from pain, is to *thread early and often*. Adding concurrency *de facto* is far more time-consuming than working on it up front.

G. Must trust tests

If tests are nondeterministic people will stop listening. Tests that pass or fail based on race conditions or test execution ordering are essentially useless. We call these *"flaky"* tests. Flaky tests cannot be tolerated.

We fell foul of a set of flaky tests when making a cross cutting change to improve performance of our application. The GUI tests were still slow at this time and failed on an nine hour overnight run. We ignored the failing GUI test run because they regularly flaked out. After a day of high stress fixing the defective production systems we ran a "PER" retrospective and realized that the GUI tests were indicating this problem. This lesson encouraged us to invest heavily in our GUI test framework.

H. Cause pain — get things done

When something is painful we react to that something. Firing of pain receptors invokes immediate remedial action. Want action? Cause pain.

Rally Engineering spaces host collections of colored LED lights that track our builds. If all tests pass then the lights go green; if any test fails then the lights go red. We have a working agreement that says: *when the lights are red you cannot push code*. [1]

The team is always trying to move faster with faster builds, tests, releases, and value delivered to our customers. To make tests execute more quickly we parallelize, multithread, cache browser profiles on SSDs, implement grid solutions, and optimize our test code all over the place. Sometimes this breaks things. Sometimes this sends us into red light hell.

When the red light blocks the entire Engineering team for a day that is painful. Their reaction is to fix the issue and make the pain go away.

Fixing strategies have included spawning off a group that met every time the light goes red, and holding a weekly meeting to "protect the build lights". One of our engineers was interested in learning more about the selenium driver and spent his two week hackathon attempting sub-five-minute run GUI tests.

To obtain better visibility into the tests we wrote a Splunk connector to ingest metrics from Jenkins logs and a Splunk engineering application. We wrote custom apps in Rally to track build health that would gather metrics on how long a build usually remains broken and the ratio of green to red lights.

We wrote a flaky finder application that hammers new test to see if they contain determinism or concurrency bugs and, more recently, we built Flowdock integrations with bots that report build health and can be queried for statistics on the test suites.

I. Transparency and managing stakeholders

The business still runs on dates and dollars. We set annual goals, quarterly goals and target market events. But if the

[1]This was an extension of an older agreement: *when the lights are red you cannot release to production.*

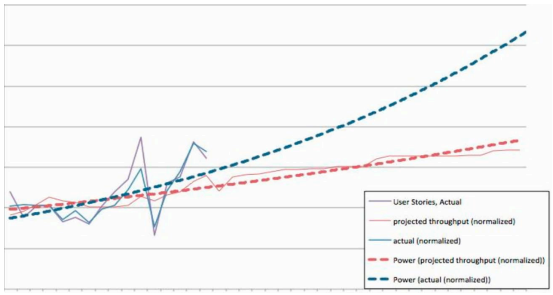

Fig. 1. Monthly Story Throughput.

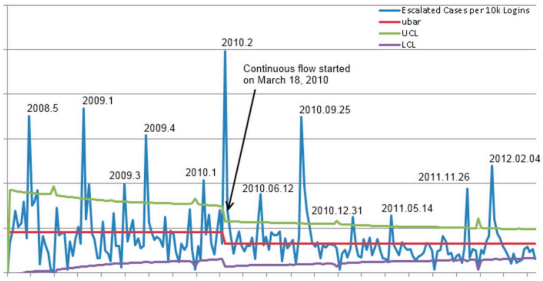

Fig. 2. Escalated Case U Chart.

delivery teams are running continuous flow how do you reconcile this?

The product team still maintains an annual roadmap but the future is less defined now. By this we mean, the roadmap is detailed for the current quarter, defined for the next and has candidate initiatives for the rest of the year.

We still have monthly product council meetings where we discuss recently released work, work in progress and take input on the near-term roadmap.

All our planning and in progress work is organized using Rally Software's product. We have built dashboards that make it easy for stakeholders to check in on progress and provide vision to what features are coming soon. Using historical data on team throughput we can estimate with a reasonable accuracy when a feature will be shipped.

Finally, the individual development teams have planning meetings on an *ad-hoc* basis. Usually when the queue of work to pull from is low.

VII. Impact

We use our own software product to manage, track and review engineering teams. Extracting data from pre- and post-continuous flow days resulted in some interesting data.

Fig. 1 shows our monthly story throughput. The raw values are represented by the lines that spike up and down on the left of the graph. The smoothed out dotted lines are the power projected throughput. The lower power line is the derived story throughput given historical data, accounting for staffing increases. The upper dotted line is the actual power line as recorded when teams complete work.

The takeaway from this figure is that our throughput increased per developer over time as our efforts toward continuous delivery progressed. This is likely due to the faster feedback cycles and improved testing frameworks.

Our second figure, Fig. 2, describes the number of customer reported defects per 10K logins over time. The regular spikes on the left of the graph are produced by eight week releases. At those times large batches of code were shipped, customers would find defects and call our support team.

The arrow, just left of center of the graph, points to the day we switched to continuous flow. From that point onwards the spikes become smaller in amplitude as fewer cases are reported at a given time.

In this figure the upper and lower control lines have been drawn (the upper and lower horizontal lines) and you will notice that they are trending closer together. This indicates that the defect reporting rate is narrowing and becoming more predictable.

VIII. Where Do You Start?

A. Engineering

Begin by *pruning all manual steps in your process of deployment to production*.

To do this you must identify what your process is. Although this may seem unimaginable for some readers there are parts of the deployment process that are opaque to people in your organization. Perform a Value Stream Mapping [6] with a sticky note exercise where a team writes down the steps from development to deployment and arranges the stickies on a wall. The results can be surprising — people may not even agree.

After documenting your processes try to identify the slowest manual step. This might be waiting for operations to repurpose a test machine or have your QA team check your app for regression bugs. Try to automate this step. Remove the manual processes.

B. Invest in your tests

This cannot be emphasized enough: *invest in your tests.*

Your automated tests must be fast, solid and reliable. If your test suite takes nine hours to run then the fastest you can ever deploy is: *nine hours*. A slow feedback loop that takes hours is painful, frustrating and easily ignored. If your test suite is slower today than it was yesterday then you must fix it — parallelize.

As discussed earlier: *destroy flaky tests*. If your team stops paying attention to the tests they are useless. Never click the "run" button again to get the test suite to pass because of some ordering or concurrency bug. Prioritize fixing that bug.

Heavy investment in your test suite is expensive but the payoff is huge. It cannot be overlooked.

C. Mimic production

The expense of the engineering effort for continuous delivery can be mitigated by making your test systems mimic production. This will allow you to reuse metrics gathering, monitoring, charting, and logging. There is an added bonus

in that as you scale your test suite it will uncover bugs in production that only manifest themselves under load. Our experiences show that our GUI test suite is a leading indicator of production bugs that are six months from appearing — our GUI tests run with a higher traffic rates than production, and our production request rate traditionally grows to match that number within six months.

D. Business

Your business team should use and embrace toggles. These enablers were key to our success in moving to a deploy at will scheme. Feature toggles can be used to schedule general release of features giving Sales and Marketing the dates they need to communicate with customers. Toggles can be used for a variety of other experiments, such as A/B and User Experience (UX) testing.

Experiment often and explore your tooling options. Build trust and create transparency (transparency to the status of the work) across the organization.

IX. Conclusion

The concept of continuous delivery is easy to understand but its implementation may require that you change everything.

From your first steps you must have buy-in from your organization and the enthusiasm from key stakeholders to carry through. Begin by understanding all your process, not just the technical requirements in Engineering but also your Sales, Marketing, Support, Technical Account Managers, and User Learning teams. Everyone will be effected. The process must be *transparent* and *clearly communicated*.

When you have a complete understanding of your deployment process begin to *automate the slowest parts* of it. These are usually the manual steps, which are also the most error prone.

Invest heavily in your automated test suite, make *tests fast, reliable* and *approachable*. Your QA team should have a comprehensive understanding of what tests coverage exists and where there are gaps. They should *test plan* and guide developers in test coverage. Do not underestimate their expertise.

Finally, have confidence. It is not free and it is not easy, but the journey towards continuous delivery will result in a *best of class engineering discipline* and velocity that will be the envy of your competition.

References

[1] D. Anderson, *Kanban: Successful Evolutionary Change in Your Software Business*. Blue Hole, 2010. [Online]. Available: http://books.google.com/books?id=RJ0VUkfUWZkC

[2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, ser. Addison-Wesley Signature Series. Pearson Education, 2010. [Online]. Available: http://books.google.com/books?id=6ADDuzere-YC

[3] D. Reinertsen, *The principles of product development flow: second generation lean product development*. Celeritas Publishing, 2009. [Online]. Available: http://books.google.com/books?id=1HlPPgAACAAJ

[4] M. Fowler. (2010, Oct.) Feature toggle. [Online]. Available: http://martinfowler.com/bliki/FeatureToggle.html

[5] Typesafe. (2013) Akka library. [Online]. Available: http://akka.io/

[6] Q. Lee and B. Snyder, *The Strategos Guide to Value Stream & Process Mapping*. Enna Products Corporation, 2006. [Online]. Available: http://books.google.com.my/books?id=vshUVrKdS90C