Enterprise Continuous Delivery Maturity Model

by: Eric Minick & Jeffrey Fredrick

How mature are our Continuous Delivery and automation practices? Where can we get the most improvement for our specific problems and needs? How are other organizations solving these same problems? This guide can help you answer these questions.

Automation in software development has expanded dramatically. Agile software development and Continuous Integration have collided with the realities of enterprise development projects — where large projects, distributed teams and strict governance requirements are not aberrations —resulting in increased automation efforts throughout the lifecycle. Top performing organizations have moved past team level Continuous Integration and tie their efforts together into an end-to-end solution. These Enterprise Continuous Delivery (ECD) efforts enable them to deliver changes faster with higher quality and with more control for less effort.

Despite these benefits the adoption of automation has been uneven. Many software teams struggle with manual, slow, high-risk deployments; others use release processes that are efficient and safe enough to deploy to production many times a day. There are many paths to improving your development automation efforts, but where to start?

Enterprise Diversity

A challenge creating this guide is that enterprises, and even the teams within a given enterprise, aren't a uniform set. The requirements when making medical devices are different than when making games than when adding features to an e-commerce site than creating an internal SOA application. Simply stepping up a simple, single maturity model can't work for everyone.

Instead we've chosen four attributes we find in Enterprise Continuous Delivery along which a team can measure their maturity: Building, Deploying, Testing, and Reporting. For each attribute we classify practices that we've observed into their level of maturity and why an organization would want to adopt them… or not. With this model you can understand the industry norms so you know where you're keeping up and where you're falling behind. The judgments reflected in these models are based on several years of first-hand experience with hundreds of teams and reports from the field.

To demonstrate how to put this model to use we've created three enterprise personas with different needs. We'll work through their situations and show how they use the maturity model to plan which improvements will give them the best ROI.

Levels in the Maturity Model

Throughout this paper the levels of maturity of various components of Enterprise Continuous Delivery will described in the same way. The weak starting points we commonly see are presented followed by an examination of a levels of maturity. The levels we are using are Base, Beginner, Intermediate, Advanced and Extreme.

The starting point is a process that is completely manual. An engineer must perform a long, error prone sequence of steps to accomplish a goal.

- **Base:** The base level is enough to “be on the model”. The team has left fully manual processes behind.
- **Beginner:** At the beginner level, the team is trying to adopt some ECD practices in earnest but is still performing them at a rudimentary level.
- **Intermediate:** Practices are somewhat mature and are delivering fewer errors and more efficiency. For many teams, Intermediate practices may be sufficient.
- **Advanced:** The team is doing something well beyond what most of the rest of the industry and is seeing a great deal of efficiency and error prevention as a result.
- **Extreme:** Elements within the Extreme category are ones that are expensive to achieve but for some teams should be their target. Put another way, most organizations would be crazy to implement them, while this minority would be crazy to not implement them.

Of course what is considered Advanced or even Extreme can change over time. At one point a certain lunatic fringe found it worthwhile to write robust systems to trigger automatic builds in response to the event of a source code commit. Continuous integration tooling is now commonplace enough that rolling it out no longer requires the effort to qualify it as Extreme. Basic Continuous Integration is easy enough to be an Intermediate level activity.

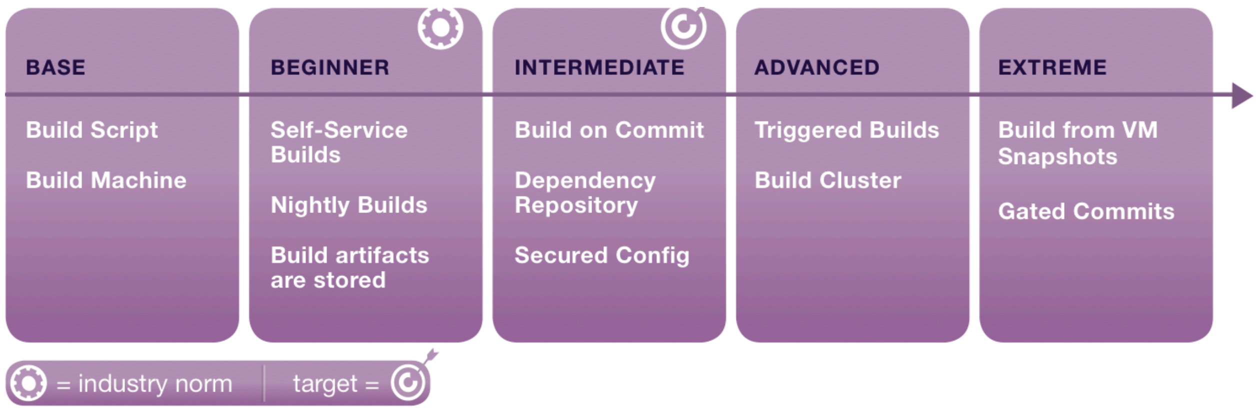
Building, Deploying, Testing and Reporting

With our model of maturity in place we are ready to address our topics in Enterprise Continuous Delivery: Building, Deployments, Testing, and Reporting. These topics cover the essential elements of the end-to-end Build Lifecycle, the journey from source code to software in production.

BUILDING

Garden variety, developer-centric continuous integration is about fast feedback from building the software. When continuous integration meets enterprise concerns, build management and a controlled build process, become a critical element. A controlled build process regulates the steps that retrieve the source code, build it, package it, and store it.

Most new projects start out with builds performed on developer machines and without a standard process. One developer builds in their IDE while another uses a build script. The least mature teams use these builds for testing or even for releases to production. The problems resulting from this lack of control are quickly apparent to most teams, starting the search for better alternatives



The first steps for maturing the build are standardizing the build process and performing official builds off developers' machines. Using a non-developer machine for the build means changes in one developer's environment don't pollute the build in unpredictable ways. Because the build is no longer done in a developer's workspace part of standardizing the build process is determining how the source is retrieved from source control to use in builds. This may be the latest from the tip of a branch, or labeled source code, or something else. The important part is the convention is consistently applied. With these practices in place the team has reached an Base level of build competence.

Any team using standard continuous integration will take this a step further and automate execution of the build steps as part of a self-service or nightly build. The build server will dictate machines, source code rules and run those build steps, providing a Beginner level of controlled build process. Typically these automated builds are scheduled daily, though some teams build twice or more a day at scheduled times.

At the Intermediate level, teams start managing dependencies on other software – both subprojects and 3rd party libraries – more explicitly. Replacing a convention of well-known locations, the Intermediate level employs a dependency management repository to trace these libraries and provision them at build time. Similarly, builds that will be consumed by other builds are put into the repository for access by the dependency management tool. All builds are stored (be it on a network drive, or just the CI server) cleaned up regularly and numbered for easy identification.

With this level of control in place automatic builds are easy to achieve and provide valuable feedback. Teams at the Intermediate level adopt continuous builds, running builds automatically upon each developer commit or when a dependency has changed. Larger teams will use distributed build infrastructure to handle a large number of builds executing concurrently.

More regulated organizations will control their build processes. In this environment the team will track changes to the build process as well as changes to source code and dependencies. Modifying the build process requires approval, so access to the official build machines and build server configuration is restricted. Where compliance is a factor or where Enterprise Continuous Delivery becomes a production system Intermediate controlled build processes should be the target; for other teams the secured configuration may be optional.

Larger organizations or those looking for slightly faster integration tests will look to the Advanced category. As the number of builds per day grows or the organization becomes more diverse (needing both Linux and Windows builds machines for instance), a single build machine becomes inadequate. A cluster is required with automatic machine selection and load balancing. At scale, traditional polling of source control systems looking for changes may place a performance strain on the ECD server as well as the source control system. In this case polling should be replaced by events from the SCM triggering builds in the ECD server, probably via webservice.

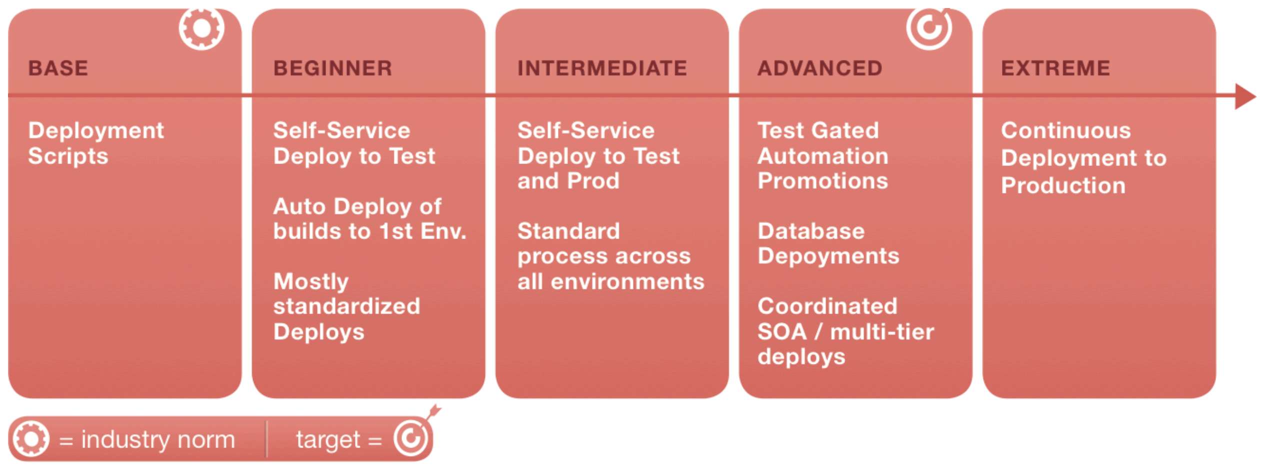
Some regulatory rules are more severe and dictate that the organization must be able to perform perfect rebuilds of previous releases. These organizations will use a variety of techniques to ensure exact replication of environments. Some have carefully versioned scripts that prepare the build machine from the operating system up prior to running the build. Others use snapshot based virtual machines, which are instantiated and have their clock modified prior to running a versioned build process. We categorize this level of controlling the build process at the Extreme level. At some levels of regulation, the team would be Extreme not to take these steps. However, these steps would be a painful burden on most teams without providing much value.

Other companies endeavor to have some development streams never be “broken”. They use a gated commit strategy to perform build and test of any chance submitted for source control. If there change fails, they reject the commit. This Extreme strategy can provide a stable area to check out from, but slows integrations, and introduces commit queues or race conditions.

DEPLOYING

Deploying is moving software to where it can be tested, accessed by a user, or ready to be sent to a customer. For a web application, this may mean installing the application on a collection of web services and updating databases or static content servers. For a video game targeting the console, one deployment would be to install the game on test machines and a production deployment may involve stamping a gold ISO to deliver to the publisher.

Deploying starts as mostly manual processes. The location of the build is sent to a deployment engineer who moves it to the target machines and run through the installation procedures. This leads to slow deployments as well as high rates of deployment failures. Engineers are often forced to work evenings and weekends to perform risky deployments of production systems or test systems that can’t be disturbed while testers are using them. Worse yet, deployments in each environment may use different processes, providing little assurance that a successful deployment in one environment indicates that the deployment will be successful in the next environment.


BASE
Deployment Scripts
BEGINNER
Self-Service Deploy to Test
Auto Deploy of builds to 1st Env.
Mostly standardized Deploys
INTERMEDIATE
Self-Service Deploy to Test and Prod
Standard process across all environments
ADVANCED
Test Gated Automation Promotions
Database Depoyments
Coordinated SOA / multi-tier deploys
EXTREME
Continuous Deployment to Production
= industry norm
target =

A team that has moved away from a fully manual process to using a number of Helper Scripts or a fully scripted process has made important improvements. Across the industry, most teams have some helper scripts but fall short of fully scripted deployments – especially to controlled environments.

Beginning ECD teams are good at deploying for testing purposes. They will have put their fully scripted deployments behind tooling that allows for push button deployments to some or all of their test environments. This takes a great deal of load off the deployment engineers and reduces the downtime of test teams waiting for a deployment. Continuous builds are a characteristic of Intermediate build teams and automatic deployment to the first test environment is a hallmark of Intermediate maturity in deployment. Depending on team dynamics, this should happen at the end of any successful continuous build or at regular

intervals through the day when testers are unlikely to be interrupted.

The last marker of a Beginner team is that they have made good progress in standardizing their deployments across environments. There may still be some environmental variation, or two types of deployments, but successful deployments earlier in the build's lifecycle are good predictors of success in later deployments. Reaching this level of maturity is a good target for many teams.

Intermediate teams turn their attention to controlled or production environments. Deployments to production (releases) are push-button and successful production releases automatically trigger matching releases to disaster recovery. Any team that is deploying to the production release to internal infrastructure should consider targeting the Intermediate level: having a consistent deployment process across all environments dramatically reduces the likelihood of last minute failures when moving to production.

Advanced teams are characterized by fully automated deployments to some testing environments based passing quality gates. For example, an approval from a test manager stages a build to be automatically installed in the stress test environment. Moreover, they have tackled some of the harder parts of their deployment processes, such as Database deployments, updating content in their content management system, or promoting reports from business intelligence tools.

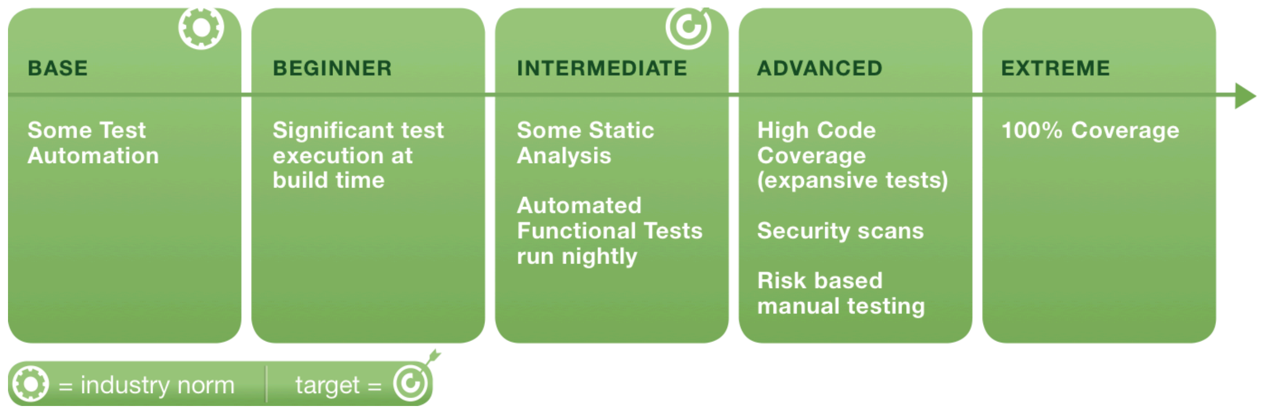
Another key characteristic of Advanced teams is that they coordinate their deployments. Rather than simply deploy a "build" they perform a deployment of inter-related builds that represent multiple tiers of an application or a collection of related web services. This helps the teams ensure that the full set of components for their system that they tested is the same set that is deployed to production. A push-button "System Deploy" replaces the push-button "Component Deploy".

Teams at the Extreme level practice continuous deployment, automatically deploying to production without any manual intervention. A build is created, deployed through testing environments in turn, tested automatically at each one, and, having passed appropriate quality gates, immediately deployed to production. Some serious dot-com applications release changes within an hour of it reaching source control. Obviously, the automated testing must be very mature, along with roll-back and application monitoring. But in a fast paced environment, extremely fast deployments of new functionality are a key competitive advantage and mitigate the risk of big bang functionality changes.

TESTING

Continuous Integration and Delivery have been long associated with some level of automated testing. This is true both in the seminal article by Martin Fowler or in the older practice described by Steve McConnell of the Daily Build and Smoke Test. In fact, it is chiefly because we want to rapidly and provide feedback on quality problems that we perform regular builds and deployments. Within the scope of Enterprise Continuous Delivery, multiple types of automated tests and manual testing are considered.

Ironically, many teams that are good at building and deploying are weak at testing. They perform a build, put it through some basic exercises manually and then release. The same parts of the application break repeatedly and new functionality is lightly tested. As teams mature in testing they detect problems sooner, increasing both productivity and confidence.


BASE
Some Test Automation
BEGINNER
Significant test execution at build time
INTERMEDIATE
Some Static Analysis
Automated Functional Tests run nightly
ADVANCED
High Code Coverage (expansive tests)
Security scans
Risk based manual testing
EXTREME
100% Coverage
= industry norm
target =

Most teams have some form of automated testing in place. Perhaps a small unit test suite or some scripted tests ensure that the basic functionality of the application works. These basic automated regression tests provide easy, early detection of fundamental problems. Teams at the Base stages of Enterprise Continuous Delivery are generally just becoming acclimated to automated testing.

To reach Beginner level maturity there should be a set of fast tests that are run with every build.

These tests provide the team confidence that the application will basically work virtually all the time. When the tests fail, the development team is notified so that they can fix the problem before their memories of the changes that broke the tests become too foggy. It is responding to the failure notifications that are important for this level of maturity; a team that has failing tests that they don’t respond to are below the Beginner level of testing maturity.

Teams at the Intermediate level expand on these fast, build time tests. Mature testing in ECD is characterized by a diverse set of tests. An intermediate team not only has the fast build time tests and manual testing, but they also have some additional automated functional tests. Intermediate level teams often employ some static source code analysis that is run by the continuous integration system. The static analysis may not be run on every build but is run periodically and any serious violations are expected to be fixed prior to release.

The Advanced stage is characterized by the thoroughness of the testing. Each test type is nearing the limit for the information it can provide. Unit tests provide coverage of all the most complex code in the system. Functional tests cover all the important functionality in the system. Boundary and randomized testing may be present as well. Static code analysis is run frequently and complemented with instrumented run-time analysis and security scans to detect likely problems the tests are missing. To support their range and scope tests are distributed across multiple systems so that they can be run in parallel and provide faster feedback.

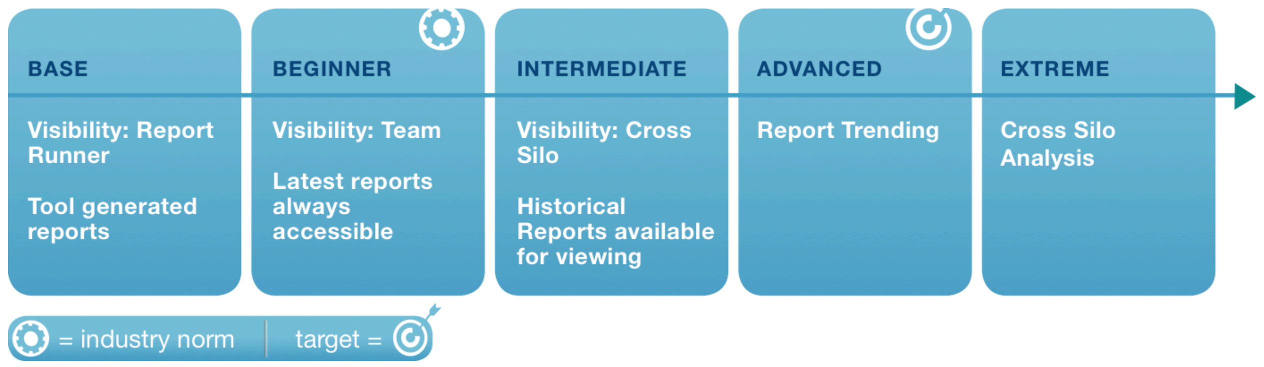
Reaching the Advanced stage requires a significant investment, but is a worthy goal for teams where the cost of defects are high and ability to sustain a fast pace is important. For projects without these attributes the Intermediate level may be the more appropriate target.

At the Extreme, some teams pursue 100% test coverage. While the definition of 100% coverage varies, it reflects at least every line of code being covered by some test. In most applications there is a point of diminishing returns, where the value of automating a test for some obscure line of code is less than the cost of creating it. Teams pursuing 100% coverage tend to accept that they will do some wasteful testing knowing that by setting the bar at the extreme they remove any excuse for creating valuable, but difficult tests. Meeting and keeping the 100% coverage target can also be a source of pride and motivation. For advanced teams that find themselves skipping some important tests, pushing for 100% coverage may be practical. For most it would be a waste.

REPORTING

Historically, continuous delivery tools have focused on reporting the state of the most recent build. Reporting is a critical element of Enterprise Continuous Delivery but with wider concerns. In ECD reporting covers information about the quality and content of an organization’s software as well as metrics relating to the Enterprise Continuous Delivery process.

A team without reporting is flying blind. All the testing in the world is useless if no one can review the results. Likewise reams of data that has not been distilled into digestible information can be so difficult to learn from as to be useless. Maturing teams have reports that are increasingly visible, and expose increasingly useful information.



Many of the tools used in the build lifecycle will generate reports. A team is acting at the Base level when individuals read the reports generated by their tool and take action based on those reports. At this level if someone else on the team needs to act based on information coming from source control, test tools, source analysis, bug tracking or other tools, the report executor contacts him and passes along the relevant information.

As teams move up to the Beginner, each group – development, testers, release engineers or operations – will publish snapshot information. A build server may provide reports about new changes to the code, source code analysis, unit test results and compilation errors. The testing team may publish results of recent automated and manual test runs from their own tools. Release engineers may track uptime for the builds in production, defects logged, as well as information about the speed of their releases. Each team is working mostly within their own silos and passing information between silos is still limited and manual. This level of reporting is the most typical baseline in the industry, although many organizations have some silos demonstrating capabilities at different levels.

At the Intermediate level, there are two big changes. The first is that a critical subset of the reports from each silo is made accessible to the rest of the software team. Testers gain insight into information from development like which features and defects were addressed since the previous QA drop or the results of developer testing. Developers gain insight into which build QA is testing and the results of those tests. Everyone who participates in the release lifecycle has access to at least summary information about any reports that could be useful to them. With broader visibility, less effort is spent communicating basic data and more effort is spent acting on the information in reports.

The second marker of an Intermediate team is Historical Reports. The organization has information not just about its most recent activity, but also about past activity as well. It can retrieve test results prior releases and compare the results to the current release. Not only does the team know that their most recent build passed 95% of tests, they know how many new tests were added, which tests were removed, and how many tests the previous build passed. Is 95% better than yesterday, or worse? Should the team be pleased by progress or spring into action to address a problem?

An Advanced team takes historical reporting information and applies Trending Reports to it. Where the Intermediate team may record each test failure, the Advanced team has reports making them aware of the tests the break most frequently. Further, they can review how many tests are being added over time, to ensure appropriate increase. Many testing initiatives die a quiet death when after several weeks or months of adding tests, no more tests are added and the old ones simply hang around. Regular reviews of the trends to check for stagnation can help an organization stick to its aims of expanding testing.

As the team matured from Beginner level to Advanced, it supplemented feedback about its most recent work, with historical information that puts the current feedback in context. However, while each report was provided across silos, the contents rarely spanned ideas from multiple silos. At the extreme level, data from multiple silos is cross-referenced for additional information. Do some areas of the code base when changed have a higher incidence of causing functional tests to fail? Is that because the functional tests are limited in scope, or does it indicate fragile code? To answer these questions, reports would need access to source control changes, functional test results and matching code coverage data. At the Extreme, teams are regularly asking these types of questions and mining their data for the results. As tooling improves, we expect this type of Cross Silo Analysis to become increasingly commonplace.

Personas

It is unlikely that a team will mature uniformly across Building, Deploying, Testing and Reporting. Different situations put a premium on different aspects of Enterprise Continuous Deliverys. To illustrate this point we offer three “personas”, fictional companies that value a different mix in their approach to Enterprise Continuous Delivery.

El Emeno Investments: Balancing Agility and Control

The team at El Emeno Investments writes trading systems for securities traders. Getting new features implemented quickly can give them key competitive advantages. However, legal requirements require tight control and auditability around releases.

Prior to ECD, the team has found themselves in a bind between the demands of traders looking for an edge and slow processes put in place to satisfy auditors. No matter how quickly a developer could implement a feature the traders needed, actually releasing it required numerous sign-offs, handheld deployments and a lengthy manual test cycle. To compensate, release engineers were asked to work late into the night and over weekends to get deployments done and documentation prepared. Tired release engineers were making too many mistakes and turn over was disturbingly high.

El Emeno Investments' first priority adopting ECD was automated, traceable deployments. Secure, push button deployments would speed the deployment process while providing the security and auditability. El Emeno worked to reach Beginner deployment capability as quickly as possible to relieve pressure on the release engineers by taking the most common deployments - to early test environments - off their hands.

The next step was to expand this automation to production releases. Due to the sensitive nature of these releases, adoption of the ECD system for production releases was initially slow. Given the speed and auditibility demands of the business, and a system that limited production deployments to people in the proper roles, the initial resistance from operations was overcome. Moving to Intermediate deployment maturity proved well worth the trouble, as there were both fewer mistakes and fewer delays when putting releases into production.

To support the deployments, El Emeno Investments adopted a build artifact repository to provide the required traceability. Other Intermediate level Building practices such as a dependency repository, a dedicated build cluster and continuous builds were outside the critical path and were put off for later consideration. Beginner level build maturity proved an adequate start.

While the quality of Testing is not carefully scrutinized by auditors or the traders, the team knows that errors can be expensive. Once the deployments were streamlined, the team prioritized building out its testing infrastructure. Basic automated regression testing run automatically after each build freed testers from their manual smoke testing. This allowed them to perform more exploratory testing which allowed them to find more bugs in less time.

With the ECD system in place the team recognized the opportunity to start addressing software security earlier in their cycle. Prior to ECD all releases would have a security review late in the cycle. After seeing how the automated functional tests allowed them to catch bugs earlier in the development cycle they decided to do the same with their security tools. Now a security scan using their static analysis tools happens daily. Although security scans are generally considered an Advanced technique they are appropriate for El Emeno given the sensitive nature of financial instruments.

For El Emeno, maintaining historic records of deployments to provide a clear audit log as well as historic records of change history and testing results is their top priority. Prior to ECD this information was available but spread across several tool-specific data silos. With their ECD system providing cross silo data aggregation El Emeno find their compliance efforts more effective for less effort. In their first post-ECD audit they were able to show their auditor the full logs for an arbitrarily selected deployment, along with the change history for that build, automated test results and a clear security scan. The auditor actually smiled!

El Emeno Investments was challenged trying to provide agility while maintaining control. For them, Enterprise Continuous Delivery was about integrating their end-to-end build lifecycle into a fully traceable solution. This didn't require great sophistication in build automation but did require mature deployment and reporting capabilities.

All-Green United Systems: Large Scale Scrum

All-Green United Systems (AGUS) is implementing Scrum across the enterprise. All-Green isn't a software company, but the organization has a large global IT group to write and manages various business critical applications. Developers, analysts and testers participate in cross-functional Scrum teams, with a separate QA team testing application integration and coordinating with release engineering to manage releases.

Prior to ECD All-Green Scrum teams reported that releases were a bottleneck. The release process was designed to slowly and carefully handle the big bang releases produced by the previous traditional development process. Sprints were not synchronized across applications and the release team had a backlog of changes from completed Sprints that were waiting to be deployed.

A single Scrum team of five to ten people is too small to manage many of the larger systems used within All-Green. Development of these larger systems is handled using the "scrum of scrums" approach with each scrum team co-located but the project as a whole distributed. These distributed teams found it was difficult to keep the codebase working because the changes of one team would often disrupt another. Some traditional applications broke at build time when changes from each team were integrated. Teams working on Service Oriented Architectures found their conflicts when the services were finally deployed together. With each team keeping their own metrics it was difficult to get feedback on the progress made in the sprint to help make better decisions about what stories will be completed in the Sprint and which will need to be pushed off.

In evaluating Enterprise Continuous Delivery All-Green had two main priorities. The first is to improve the coordination between Scrum teams. The second was to improve the speed of the release process so that changes could be released with every Sprint and get new functionality into use more quickly.

When it comes to Building, every Scrum team in All-Green United Systems had their own build system performing team level continuous integration with their own changes. The first change AGUS performed from the Building perspective was to unify all the teams into a single build system that can be shared. Some teams still run their local CI systems but the enterprise system produces the software that goes into production and coordinates activities between the teams.

Where binary dependencies exist between Scrum teams, an authoritative artifact repository is integrated with this build system. This insures smooth, transparent component sharing rather than ad-hoc approach like emailing artifacts. Like their practice's "scrum of scrums" the component aware ECD system for All Green performs a "build of builds" that smartly builds a dependency tree of components into a larger system. Now that changes are shared between teams only after passing local tests reducing the incidences of failed builds across the system. Further, when failures do happen they can be quickly traced back to the originating team so that they are corrected faster. With so many teams and builds All-Green has a distributed build grid. With a larger pool of machines the individual teams now enjoy faster feedback than when they maintained their own systems. Such a system puts the AGUS team solidly into an Intermediate level of maturity from a Building perspective and includes some Advanced components.

Testing maturity varies throughout AGUS by application. All teams are developing automated regression suites that they are expanding over time. The focus is on developing functional tests for new areas so older legacy applications are relatively less tested, while the newest applications have comprehensive automated functional tests. A couple legacy applications were both mission critical and frequently changing. In these cases speeding the release cycle required an additional investment in robust automated testing. For scrum of scrum teams there is an additional emphasis on testing dependencies across team boundaries with unit level tests.

To speed the release process, All-Green automated deployments starting with their various test environments. Standardizing the process between teams was the first step, followed by standardizing across environments. The applications with a SOA architecture encountering deployment time integration challenges focused on deployment coordination – driving directly for this Advanced practice. After proving the system in all the test environments push button deployments to production allowed All-Green to eliminate their release backlog. Some of the newer applications at All-Green have extensive automated testing and aren't coupled to other production systems. These applications have stepped up to the Extreme level of continuous deployment, fully automated deployments through to production.

The other key element to coordination between Scrum teams was Reporting. Prior to ECD even within scrum teams data was stuck in the tool specific silos of development, testing and production support. All-Green used their ECD system to break down those walls so they could correlate and trend their data end-to-end within their sprint and across sprints. This improved reporting fed the scrum of scrums, reducing the coordination overhead. Even single scrum teams found they had better insight into their progress as they could relate their stories to commits to their test results. The other side of Reporting maturity for All-Green United Systems was to provide a unified view through the full process including the release team.

For All Green United Systems, Enterprise Continuous Delivery helps harness the efficiency of small Scrum teams by enabling them to work together effectively. Disparate development teams can cooperate in large system builds automatically and build, test and deployment teams can use a common system and gain insight into each other's activities.

Diversity Handhelds: ECD for Embedded Systems

Diversity Handhelds writes software platforms for mobile devices. This private company delivers builds to partners who work with the hardware manufactures. Diversity must build the same software for a number of different handheld operating system and hardware configurations. A typical product may be built in thirty configurations.

Prior to implementing ECD, Diversity has been building and testing individual configurations. Because inactive configurations weren't built regularly, conflicts with tool chains could go undetected for quite some time. This made diagnosing where the problem was introduced difficult at best. Careful manual tracking has helped Diversity keep track of the contents of each build released to customers, but finding this information is a challenge for support.

When it comes to ECD, Diversity Handhelds had a few key priorities. The most pressing was orchestrating and managing builds across the build machines hosting specialized compilers and tool chains.

To address this problem Diversity Handhelds implemented an Enterprise Continuous Delivery system that could distribute their builds across a smart cluster of build machines. Their cluster is aware of which tool chains are installed on which machines, and performs load balancing. In some cases builds are performed on virtual machines and the ECD system can spin up additional images as appropriate. This is done especially for obscure or retired platforms that are built rarely. To detect conflicts Diversity now builds for all platforms on a regular basis with the ECD system ensuring that active platforms have the highest priority.

Because of requirement to track builds handed off to their customers, Diversity now manage their built artifacts in a software library. This has reduced the effort required to generate the build of materials and release notes for a release as well as reducing errors caused by having out of sync components.

Because of the testing challenge implicit in this scale of effort, Intermediate testing was the minimum bar for Diversity. Static code analysis was a good fit for Diversity as it was able to detect common problems in native code that simple smoke tests might miss. Automatic tests run in the available simulators and good smoke tests for the most important and representative platforms are now in place. To make this work efficiently, Diversity put in place automatic deployments to simulators and target devices. Unit tests run with ever build are part of the testing mix as well as some static analysis. In the future Diversity wants to make the move up to Advanced testing to make the most out of automation and ensure that their manual testing was spent optimally.

The Advanced level of reporting is probably desirable for Diversity Handhelds. As a product company, information about historical releases is highly relevant. Plus with large builds, aggregating data across silos and providing trending of results will help put key items into context. The project's scale precludes keeping a firm grip on its historical status off the cuff. For DH, Intermediate level reporting is an absolute must, and Advanced should be pursued aggressively.

Diversity Handhelds has put in place a build system that generates repeatable, consistent builds across a number of platforms and tool chains. Their builds are tracked with basic reports available for released builds. Automated testing supported by automated deployment has kept their cross-platform quality high reducing the time required to deliver their software for next generation devices.

About UrbanCode

Urbancode is the technical leader in Agile build and release management solutions since 2001.Urbancode offers the complete Enterprise Continuous Integration platform to get your builds and deployments under control: Replace manual script-driven bottlenecks with web-based self-service automatio Provide agility and control in harmony with fast feedback and audit trails. Customers in the financial, banking insurance, software and high-tech industries achieve end-to-end automation using .NET, Java, and/or native technologies. Urbancode solutions deliver complete lifecycle automation, management, visibility, and traceability, covering the span from build to production, from development to operations. Over 400 savvy, market-leading companies -- including 25 of the Fortune 100 -- rely on Urbancode for Build Management, Deployment Management, and Application Lifecycle Automation.

uBuild

Urbancode's UBuild helps development organizations scale while maintaining control. By integrating with source control, issue trackers, test automation, and other code quality tools UBuild combines build automatic and management with visibility into code quality. Automation allows for faster feedback and shorter release cycles while efficiently implementing quality gates on the promotion path to production. Teams looking to bui an application then deploy it through environments will find UBuild an especially effective end-to-end automation solution.

uDeploy

Urbancode's UDeploy allows development, release management and operations teams to collaborate on reliably consistent deployments across environments. Out-of-the-box integrations with IIS, J2EE servers, loa balancers, and more allow organizations to replace cumbersome deployment scripts with reusable automate workflows. UDeploy provides coordination and management for the deployment of complex, multi-tier or mul component applications.

uCloud

Urbancode's uCloud tool supports the easy creation of environment from private clouds. An environment template holds the plan for the various servers needed for a test or production environment. When a new environment of that type is required, uCloud integrates with the virtualization platform, instantiating a collectic of images, an configures them into a full environment. It then notifies and configures uDeploy for the new environment enabling immediate deployments to that environment.