urban{code}

Enterprise Continuous Delivery Maturity Model

BY ERIC **MINICK** & JEFFREY **FREDRICK**

office 216 858 9000
fax 216 393 0006
2044 euclid ave., ste.600
cleveland, ohio 44115
www.urbancode.com

How mature are our Continuous Integration 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 have tied their automation efforts together into an end-to-end solution. These Enterprise Continuous Integration 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?

urban{code}

Enterprise Diversity

One challenge we faced when 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 when creating an internal SOA application. A single maturity model can't work for everyone.

Instead, we've chosen four attributes of Enterprise Continuous Integration 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.

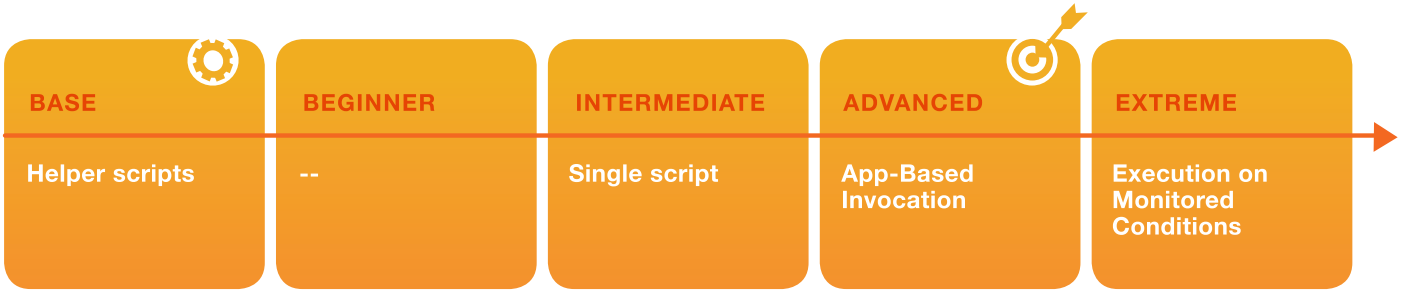
Levels in the Maturity Model

Throughout this paper the levels of maturity of various components of Enterprise Continuous Integration will be described in the same way. The weak starting points we commonly see are presented first, followed by an examination of the levels of maturity. The levels we are using are Introductory, Novice, Intermediate, Advanced, and Insane. To explain these levels, let's look at what automating a process typically looks like. 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. We can map out a maturity model for process automation that looks something like:

As the team starts down at the Introductory level they create helper scripts to automate some particularly slow or problematic portions of their process. The benefits of this automation would include both saving time and reducing errors. To take their maturity a step further, the team can reach an Intermediate level by fully automating the process into a single script. The payoff of the script is a streamlined process that is easier to handoff to other people. As the team looks to mature further to Advanced level, they can then replace manually running the script with invocation from an application. This application might be running the same scripts behind the scenes but it executes the process with the correct flags, in the correction location, etc., every time. The application might be a simple push-button interface, or perhaps the script may be run with set parameters in a scheduled manner.

This Advanced level of maturity is excellent for most processes, and is therefore marked as the recommended target. In most teams, however, many engineering processes are only automated at the helper-script level, so the Introductory level is the industry norm. Thus a team that lacks helper scripts is not just short of ideal, they are also behind the curve.

Elements within the Insane category are ones that are expensive to achieve, but for some teams should be their target. Put another way, most organizations would be insane to implement them, while this minority would be insane not to implement them. For example, an Insane practice would be an active monitor that would invoke the process when



needed, based on complex criteria. This level of investment isn't needed for most processes, but is required for some operation support scenarios.

Of course, what's considered Advanced or even Insane changes with time. At one point Continuous Integration (CI) was limited to a certain lunatic fringe that found it worthwhile to write systems to trigger automatic builds in response to the event of a source-code commit. CI tooling is now commonplace enough that rolling it out no longer requires the effort to qualify it as Insane. Basic CI is now 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 Integration: Building, Deploying, Testing and Reporting. These topics cover the essential elements of the end-to-end Build Life-cycle, the journey from source code to software in production.

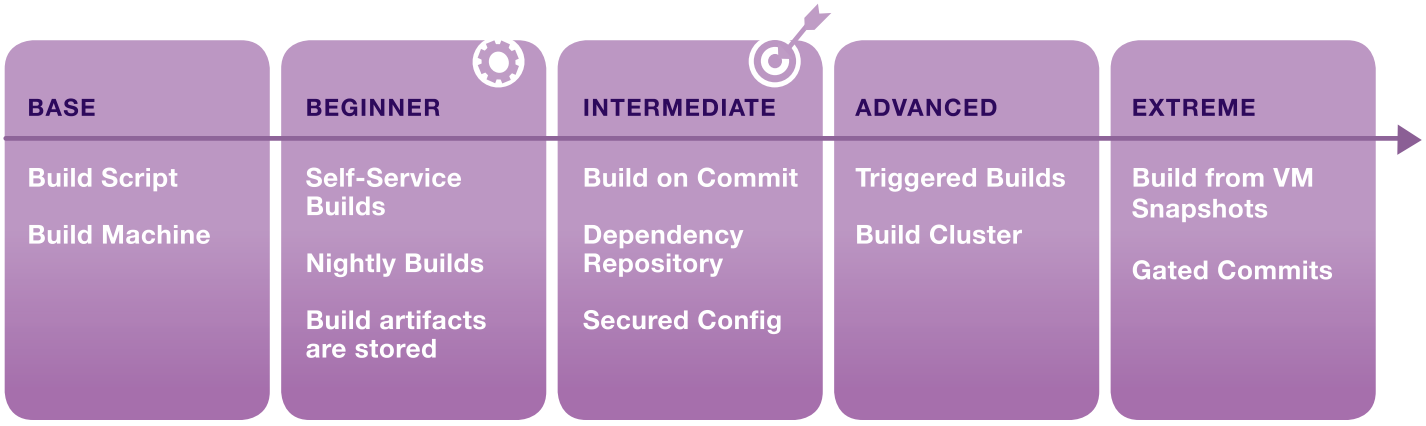
Building

Garden-variety, developer-centric CI is about fast feedback from building the software. When CI meets enterprise concerns, build management, dependencies between projects, and a control over the build process become critical elements.

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, and thus start a 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 that the chosen convention be applied consistently. With these practices in place, the team has reached an Introductory level of build competence.

Any team using standard Continuous Integration will take this a step further and automate execution of the build steps. The build server will dictate machines, source-code rules, and run those build steps, providing a Novice 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 third-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

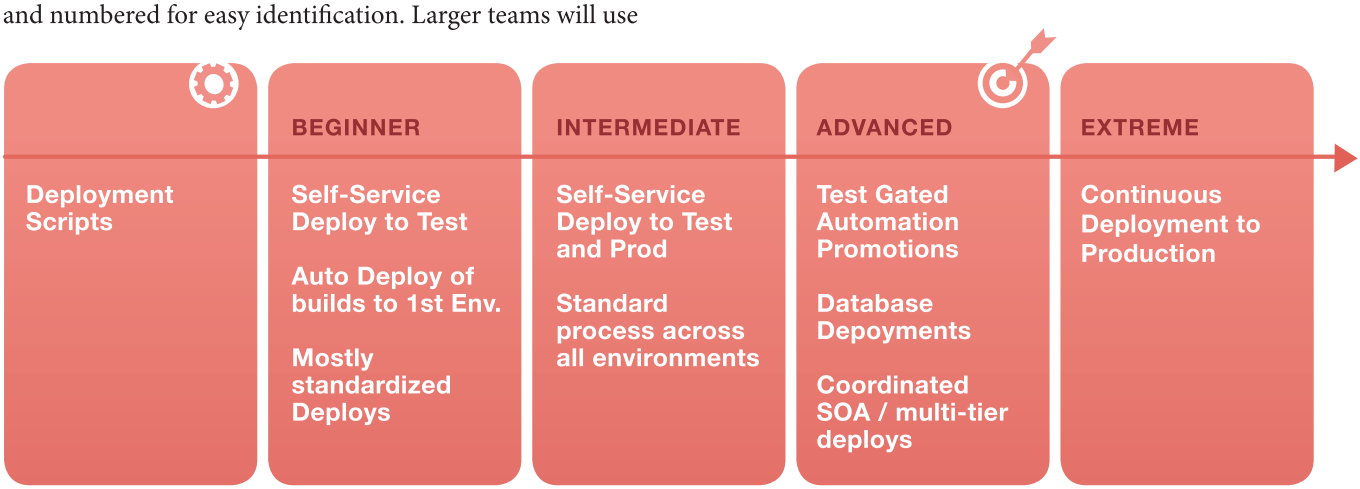


by other builds are put into the repository for access by the dependency-management tool.

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. All builds are stored (be it on a network drive, or just the CI server) cleaned up regularly and numbered for easy identification. Larger teams will use distributed build infrastructure to handle a large number of builds executing concurrently. At this point, many organizations have their needs satisfied.

More regulated organizations will step up to Advanced controlled 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 Integration becomes a production system, Advanced controlled-build processes should be the target; for other teams Intermediate level is adequate.

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 Insane level. At some levels of regulation, the team would be insane not to take these steps. However, these steps would be a painful burden on most teams without providing much value.



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.

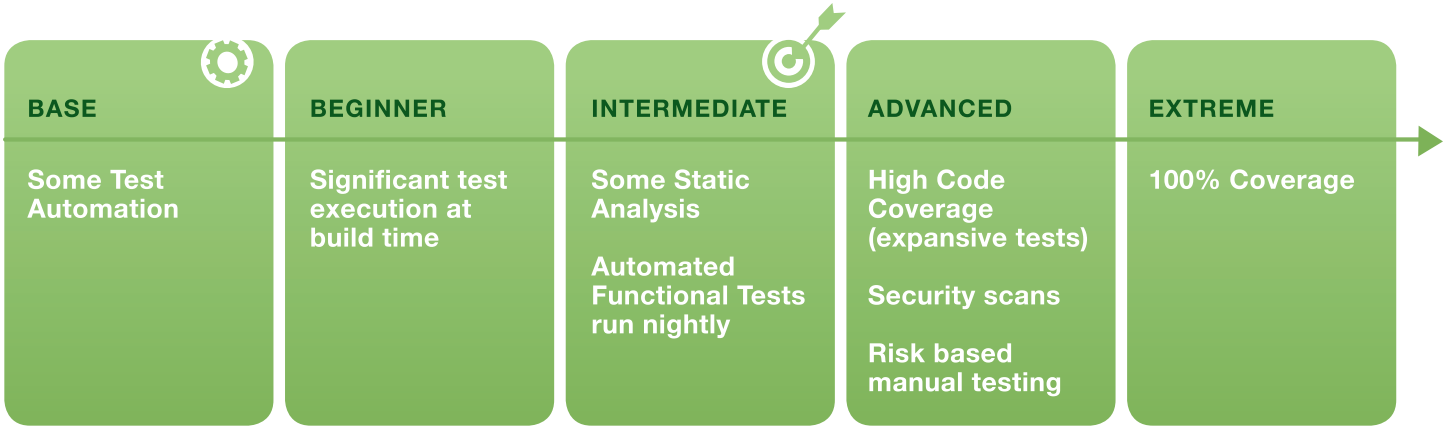
Deploying starts as mostly manual processes. The location of the build is sent to a deployment engineer who moves it to the target machines and runs through the installation procedures. This leads to slow deployments as well as high rates of deployment failures. Engineers are often forced to work evenings and weekends to perform risky deployments of production 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.

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.

be interrupted. The last marker of an Intermediate 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.

installed in the stress-test environment. Teams at the Insane level practice continuous deployment, automatically deploying to production without any manual intervention: A build is created then automatically deployed through the testing environments. As it moves through the pipeline, the build is subjected to automated testing at every stage, and once it has passed all the appropriate quality gates, it is immediately deployed to production.



Intermediate 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 of the deployment engineers while reducing the downtime of test teams waiting for a deployment. Just as continuous builds are a characteristic of Intermediate build teams, 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

Advanced 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 deploys to production on internal infrastructure should consider targeting the Advanced level: having a consistent deployment process across all environments dramatically reduces the likelihood of last-minute failures when moving to production. Advanced teams are also 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

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 has long been 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. Within the scope of Enterprise Continuous Inte-

gration, multiple types of automated tests and manual testing are considered.

Despite this, many teams 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.

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 Introductory stages of Enterprise Continuous Integration are generally just becoming acclimated to automated testing.

To reach the Novice level of 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 of the time.

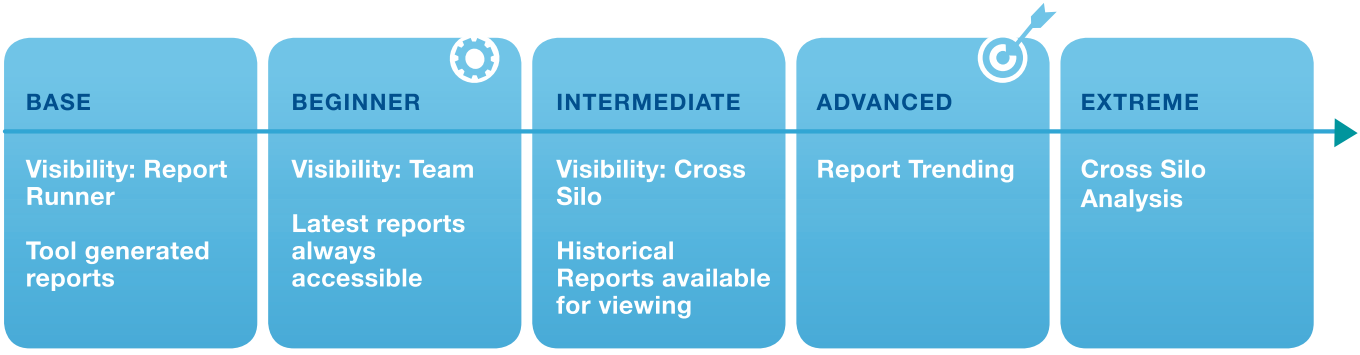
When the tests fail, the development team is notified so that they can fix the problem before their memories of the changes that broke the tests become too foggy. It's responding to the failure notifications that are important for this level of maturity: a team that has failing tests that they don't respond to are below the Novice level of testing maturity.

Teams at the Intermediate maturity level expand on the fast, build-time tests. Mature testing in Enterprise Continuous Integration is characterized by a diverse set of tests. An Intermediate team not only has the fast build-time tests and manual testing, they also have automated functional tests. Intermediate teams often employ some static source-code analysis run by the CI 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 tests may be present as well. Static code analysis is run frequently, and complemented with instrumented runtime 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 the automated test. 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 teams it would be insane.

Reporting

Historically, Continuous Integration tools have focused on reporting the state of the most recent build. Reporting is a critical element of Enterprise Continuous Integration but with wider concerns. In Enterprise CI, reporting covers information about the quality and content of an organization's software, as well as metrics relating to the enterprise CI 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 Introductory 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 Novice level, each group – developers, testers, release engineers – 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 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: such as which files changed 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 build lifecycle has access to at least summary information about any reports that are 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 of 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 also 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 happy about it 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 that break most frequently. Further, they are aware of which source files, when changed, are most likely to break unit tests as well as functional tests run by the testing group. Identifying areas of fragility helps the team decide which areas of the code base could use additional tests or redesign. Historical data from several silos is aggregated, cross-referenced and digested in specialized reports. The Advanced team is also one which actually reads those specialized reports and acts on them. Generating these actionable cross-silo reports should be the target for an Enterprise Continuous Integration system.

As the team matured from Novice level to Advanced, it supplemented feedback about its most recent work with historical information that puts the current feedback in context. The next logical step, the Insane level, is reporting that looks towards the future. An Insane level team could have information about numerous metrics that relate to releases of software to customers. It will be able to retrieve information about defect reports and support traffic generated by those production releases. With this information, it will create reports that predict the expected support-burden

of a release. The team should be able to estimate the number of defects customers would report in the first week of release by comparing data from the current release to those of past releases. With such modeling they can ask more interesting questions than simply "are we feature complete"?

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 Integrations. To illustrate this point we offer three "personas," fictional companies that value a different mix in their approach to Enterprise Continuous Integration.

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 Enterprise Continuous Integration, 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 in adopting Enterprise CI was automated, traceable deployments. Secure, push-button deployments would speed the deployment process while providing the security and auditability. El Emeno worked to reach Intermediate deployment capability as quickly as possible to relieve pressure on the release engineers. They took the most common deployments to early test environments off the hands of release engineers.

The next step was to expand this automation to production releases. Due to the sensitive nature of these releases, adoption of the Enterprise CI system for production releases was initially slow. Given the speed and auditability 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 Advanced deployment maturity proved well worth the trouble, as there were both fewer mistakes and fewer delays when putting releases into production.

To support Intermediate and Advanced level 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. Novice-level build maturity plus a solid artifact repository 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 Enterprise Continuous Integration system in place the team recognized the opportunity to start addressing software security earlier in their cycle. Prior to Enterprise CI 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 Enterprise CI this information was available but spread across several tool-specific data silos. With their Enterprise CI system

providing cross silo data aggregation, El Emeno finds that their compliance efforts are more effective with less effort. In their first post Enterprise CI 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 Integration 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 is implementing Scrum across the enterprise. All-Green isn't a software company, but the organization has a large global IT group that writes 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 Enterprise Continuous Integration, All-Green's 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 undeployed changes from previous Sprints.

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. With each team keeping their own metrics, it was difficult to get feedback on the progress made in the sprint, and to make better decisions about what stories will be completed in the Sprint.

In evaluating Enterprise CI, All-Green had two main priorities: The first was 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, getting new functionality into use more quickly.

When it comes to building, every Scrum team in All-Green their own build system performing team-level Continuous Integration with their own changes. The first change All-Green 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 ensures smooth, transparent component sharing rather than the ad-hoc approach (e.g., emailing artifacts). Like their practice's "scrum of scrums," the component aware Enterprise CI 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 there are fewer 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 All-Green team solidly into an Intermediate level of maturity from a building perspective.

Testing maturity varies throughout All-Green 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.

The other key element to coordination between Scrum teams was reporting. Prior to Enterprise CI, even within Scrum teams the data was stuck in the tool-specific silos of development, testing and production support. All-Green used their Enterprise CI 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.

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. 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 Insane level of continuous deployment, fully automated deployments through to production.

For All Green United Systems, Enterprise Continuous Integration 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: ECI 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 30 configurations.

Prior to implementing Enterprise Continuous Integration, Diversity had 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 was a challenge for support. When it comes to Enterprise CI, 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 CI 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 Enterprise CI system can spin up additional images as appropriate. To detect conflicts Diversity now builds for all platforms on a regular basis with the Enterprise CI 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. 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 Diversity, Intermediate level reporting is an absolute must, and Advanced should be pursued aggressively.

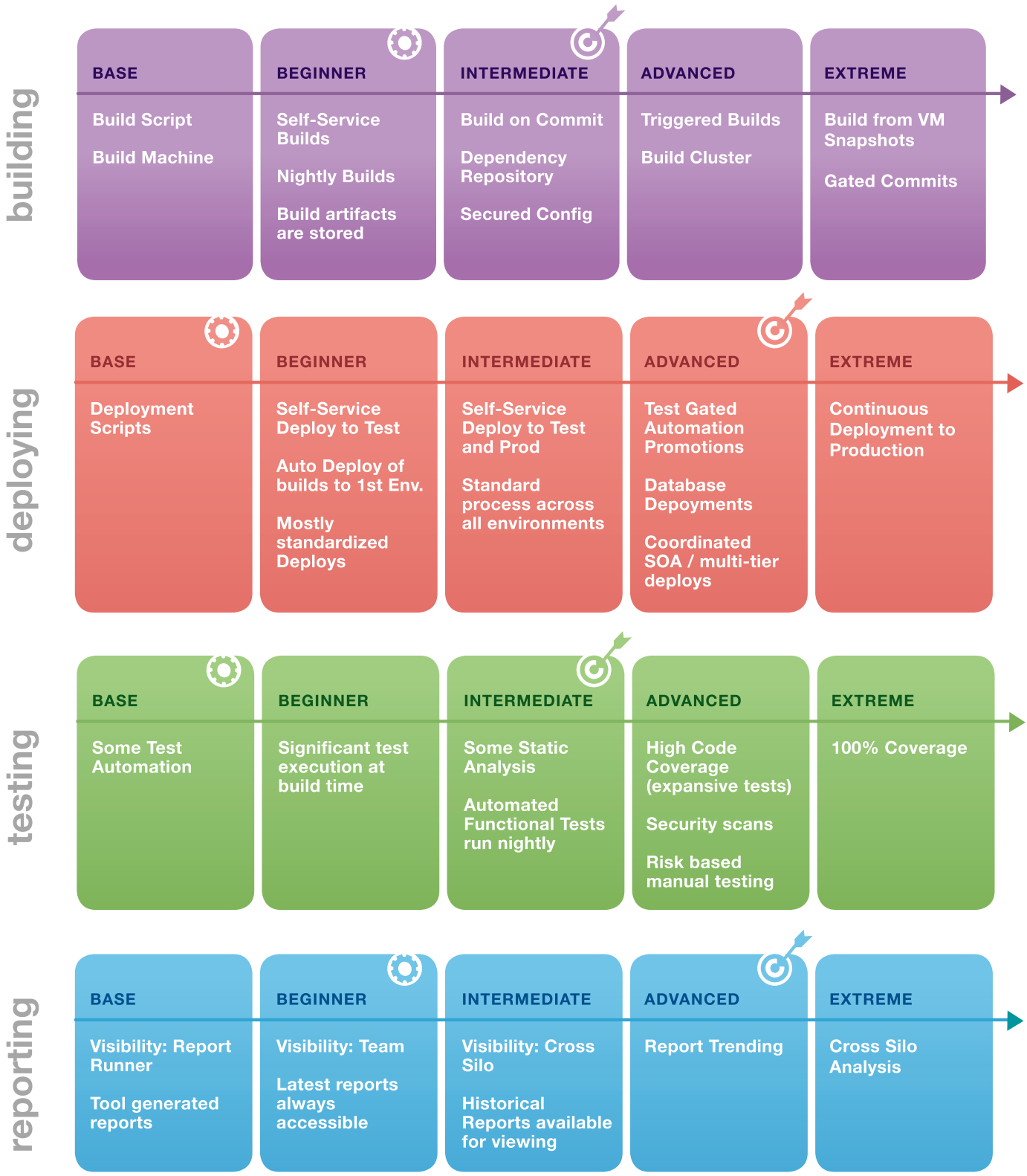
Diversity Handhelds has put in place a build system that generates repeatable, consistent builds across a number of platforms and tool chains. Their builds are tracked with basic reports available for released builds. Automated testing supported by automated deployment has kept their cross-platform quality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

urban{code}

office 216 858 9000
fax 216 393 0006

2044 euclid ave., ste. 600
cleveland, ohio 44115
www.urbancode.com

Enterprise Continuous Delivery



= industry norm | target =

Taken from our Enterprise CD Maturity Model Whitepaper
http://www.urbancode.com/html/resources/white-papers