

The Agile Heartbeat

How Agile Development Puts Fast, Automatic Builds Center Stage

by John Graham-Cumming | © 2009 Electric Cloud, Inc.



The Agile Heartbeat

Introduction

In this whitepaper I argue that the person most affected by the introduction of agile or extreme programming techniques is not the software or quality assurance engineer, but the build manager. And the build manager can no longer make do with home grown tools; software production automation tools are required to make agile development a build reality.

The reality is that agile techniques are a throwback to the age when developers were able to work on small projects in small teams. Each developer (or sometimes pair of developers) concentrates on small building blocks of code (with associated unit tests), and integrates regularly with other developers to ensure that the overall software project is progressing. For developers, agile techniques are a natural fit because they reflect how developers like to work best: on small, manageable pieces of code with regular feedback.

Even though developers are working on small sections of code, the overall projects they are part of are now very large. So there's no going back to a small build on a single machine for build managers. While developers may have broken their work down into small units that they can code and test, the overall size of most enterprise software projects is constantly growing. And it's the large body of code that the build manager is expected to work with, not the manageable chunks that developers deal with.

In fact, the build manager is expected to cope with ever larger software, on ever more platforms, while at the same time dealing with developers' requests for fast integration builds. Those integration builds enable agile development at the software engineer's desktop because they are able to integrate their local changes into a large build, but cause build managers to be required to produce one or two orders of magnitude more builds per day.

All these changes are brought about by one central tenet of agile development: *continuous integration*.

Continuous Integration

The seminal paper on Continuous Integration (and an excellent, and readable introduction to the topic) is Martin Fowler's article entitled, simply, Continuous Integration and available at <http://www.martinfowler.com/articles/continuousIntegration.html>. Its abstract states:

Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

To the build manager the words 'multiple integrations per day' and 'each integration is verified by an automated build' mean a radical change from once-nightly builds.

Fowler goes on to list a number of 'Practices of Continuous Integration'.

The first build-related practice is '**Automate the Build**'. He argues that the set of tasks required to go from source code to the final shipping binary is so complex that it should be automated so that it is repeatable. If the



build is not automated, he argues, the build process will be error prone.

He also argues that the build includes much more than just compiling the software and building a binary. He sets the goal of 'anyone should be able to bring a virgin machine, check the sources out of the repository, issue a single command, and have a running system on their machine'.

It's worth stopping and thinking about the implications of that statement for your build organization. It's very likely that getting to such an automated stage seems impossible, when starting from what is often a creaky build system held together by a collection of Makefiles, Perl scripts and other programs whose exact function or operation is often unclear.

Clearly, the goal of going from a virgin machine to fully running code is a stretch, but I think it's a good overall goal. Once you reach that stage you will have put together a build system that is very reliable, and should, if written and documented well, be easy to modify and adapt as software changes.

Such a fully automated build has advantages outside the direct realm of agile development. How many times has your build team been asked to rebuild an old version of your software and been unable to do so? Important customers can sometimes demand that old versions of code be patched or updated, or a security problem can mean that all versions of a company's currently supported code need to be fixed and released.

In general, only very well prepared teams are capable of building an arbitrary version of their code. Even if a good archive of the sources was made at the time of release, other components of software are likely to have decomposed: you may not have the right build scripts, or the right compiler, or the right version of some third-party component any more.

Fowler's goal, which I refer to as a 'pickled build' (the entire build is pickled in a jar ready for use whenever demanded), done right, means that old versions of software can be rebuilt at will, and helps support developers in their move to agile development.

Fowler's second practice for builds is: '**Make Your Build Self-Testing**'. This implies that the automatic build also includes an automatic test. He asks that any body of code have a set of automatic tests that can check a large part of the code base for bugs. Many developers are, in fact, already writing such test suites through the Extreme Programming focus on unit testing, or by performing Test Driven Development (where developers write the tests before the code).

With this test suite in place, Fowler asks that the automated build test itself by running the test suite and reporting the results. Results would typically be reported on screen or by email if the build and test was run remotely.

Fowler's next two practices have a profound effect on build management: '**Everyone Commits Every Day**' and '**Every Commit Should Build the Mainline on an Integration Machine**'. The implication is that every developer will be committing code once per day (at least) and that every commit will cause a build and test on some build resource (the 'integration machine') managed by the build manager. To put that in perspective for a team of 20 engineers committing once per day during an 8 hour work day that's a build and test every 24 minutes. For large teams that number increases, and the time between builds is greatly shortened.

The reason agile developers want these per-commit builds is to ensure that integration between developers is working and that the software being built works, and is testable, at all times. The idea that the software should run at all times is central to agile development, and the health of the per-commit build and test becomes an important sign of health for the entire project. If per-commit builds are done quickly and frequently enough the



build will reflect changes made by a single check-in by a single developer offering unparalleled opportunity to narrow down the cause of a breakage.

In fact, this build is so important that I refer to it as the Agile Heartbeat. Agile teams will install monitoring devices (such as red and green lava lamps) that make the status of the heartbeat build visible to all. Fowler states that no developer should 'go home until the mainline build has passed with any commits' they've added. This means that all of engineering looks every day to the heartbeat to measure their own progress (Fowler refers to this in the practice '**Everyone can see what's happening**').

Another important practice mentioned in Fowler's paper is '**Keep the Build Fast**'. This seems like an obvious corollary to the previous practices, since a team that requires a build every few minutes and every time the code changes will necessarily need very fast builds. In fact, Extreme Programming outlines the goal of 'ten minute' builds, and I've written previously about what I call espresso builds (builds that only take as long as a coffee break).

Fast builds are also important because developers are looking to the status of the build as a measure of their progress. Every minute that is shaved off the build is a minute saved for all developers who have committed code to that build. With many developers, and many builds, that time saved adds up quickly and fast builds improve the productivity of the entire team.

Fowler suggests that this entire process be either managed by hand (if the team is very small), or through the use of a 'continuous integration server'. This whitepaper will later describe tools that can be used to implement continuous integration, but first the very idea of getting to continuous integration may seem daunting. Luckily, it can be broken down into stepping stones that are of a manageable size.

Stepping Stones to Continuous Integration

Although pickling your entire build system (getting the build under ten minutes, building every time a developer checks in and giving automatic feedback) is an enormous task, a stepping stone approach can get you to continuous integration without a single, painful push to change everything in the organization.

And, typically, build managers simply do not have the time or resources to spend on a major change to everything they do. This is especially true when build managers are, naturally, expected to keep churning out the builds that they currently create while improving their processes to meet the needs of agile development.

However, a five-stage approach can help ease the way to continuous integration. Those five stages are: fully automated builds, fast builds, lava lamps, build and test and pre-flight builds. At each stage the build manager should carefully consider the tool available for automation of the software production process as continuous integration requires a very large effort.

1. Fully Automate

Although some builds are already fully automated, most require some manual intervention on the part of the build manager (for example, during the installation of the software when a dialog box needs clicking). But the core of continuous integration and agile builds is the ability to build your software automatically.

So the first stepping stone is to make the build runnable from a single command invocation (the build manager should simply be able to type build followed by some arguments specifying a particular branch or version of the software to build).

Once that build script is in place the build system should be set to run automatically.



This is mostly a matter of detecting when the source code in the source code management system has changed. One way to do that is to have a simple periodic job that checks for changes. Once changes are detected, the job then waits for the repository to stabilize (to give a developer time to commit all their code). The job could then wait for a period of quiet for fifteen minutes following a commit.

Once the quiet period has ended, the build system starts a full build and test using the build script to build from the main line of code.

This is probably the only stepping stone that can be performed without resorting to new software production management software. The build script can be written using existing tools (such as GNU Make or Perl), but it's worth looking at available automation tools (open source or commercial) since the detection of source code changes and kicking off of an automated build is a common feature. Getting started with a new tool in a limited way (just for the change detection and build kick off) is a good way to learn the tool without having committed to each of the five steps to continuous integration.

2. *Fast Builds*

Having automated the build, the next step is to speed up the builds themselves. Although Extreme Programming preaches a ten minute automated build and test, I think that a realistic goal for an existing project is to get the automated build and test under one hour.

Getting to fast builds can be hard.

For some projects it's simply a matter of upgrading from outmoded build hardware to newer machines (and often newer, faster disks since builds require a lot of disk access getting sources and writing objects), but for others the build will either require restructuring (so that it can be broken down into parts runnable on separate machines), or a purpose-built parallel build system will be needed. Parallel build systems are available as both open source and commercial products (including from Electric Cloud). With multi-core processors and multi-processor machines becoming commonplace, exploiting parallelism available in builds is a relatively simple way to achieve significant build speedups. However, missing or incomplete dependency information makes implementing a homegrown parallel build error-prone, and requires specialized tools to overcome the inherently serial nature of almost all large build systems.

3. *Lava Lamps*

Lava lamps (those bubbling colored liquid lamps popular in the 1970's) may seem like a silly way to monitor build progress, and in a physically large team (or with remote teams or a complex project) they may not actually be suitable, but introducing some sort of build monitoring is the third stepping stone. Lava lamps are just a fun, easy-to-implement example of a monitoring system.

A build monitor could be red and green lava lamps, or an internal web page showing live build status, or a flat screen monitor mounted high on the wall giving build status. Introduce a mechanism for build feedback that is clear (red and green for bad and good, for example), easily visible (perhaps those lava lamps are right by the water cooler) and continuously updated.

The build monitor will get its input (good or bad) from the fully automated builds set up in the first stepping stone. Everyone will look to the build monitor first thing in the morning and throughout the day as new builds are run.

Although the build may only run and give feedback a small number of times per day (perhaps as little as four times), the build monitor will start to represent the pulse of the engineering team, illustrating the health of the



build and hence the software for all to see. This is another important step towards the goal of every commit initiating a build with results for all to see.

Even with just four builds a day with feedback, developers will be able to identify integration problems much quicker than with a nightly build. And with the automation effort expended in the first stepping stone, these automatic builds will be reliable and repeatable and a good indication of the software's health.

4. *Build and Test*

If developers are following agile methods they'll be developing automated test suites using tools like jUnit or cppunit. The third stepping stone is to integrate these tests (and other 'smoke tests' that may already exist) into the build system now in place.

This is probably a relatively simple task given that the tests themselves are already automatic. This step is also important because it will raise the visibility and importance of the builds, and becomes the start of the agile heartbeat.

5. *Preflight Builds*

The primary need of developers practicing agile development is the ability to see that their code integrates with the changes made by other developers. Although they could do this on their local machine (by getting all the relevant sources and running a tool like Make), the integration build may take too long, or need special build resources to be able to construct every part of the software. Typically, that means that developers are forced to limit their builds to small parts of an overall project. Even if developers could run full builds on their machines, the production build environment is often different from the environments on developers' machines, so a build that works on a developer's machine may not work in the production environment. Because of these issues, developers frequently introduce errors when they ultimately integrate with the complete system by checking in their code.

A good first step is to put full builds in the hands of developers. The build manager needs to assign a machine (or perhaps more than one if multiple platforms are involved) and set up an internal web page where a developer can request a build.

The developer's request should include their email address, the location of the software they want built (this could be the name of a branch to build, or a directory on a shared server where the developer has placed a personal copy of the source), and the opportunity to determine the extent of the build (for example, the developer could request a build on all platforms, or limit the build to just a single operating system).

The system should maintain a queue (visible through another internal web page) of pending build requests and should handle the builds in first-come, first-served order. When a developer's build has completed running (perhaps a couple of hours after their request), the developer receives an email giving the status of the build and any error output generated by the build system.

In addition, the build system takes the entire output of the build and stores it on a shared disk for the developer to examine. After some fixed period (perhaps a week), the output of the build is automatically deleted by the build system to free up space.

The advantages to the agile developer are immediate: they no longer have to wait for a nightly build (with the inherent 24-hour delay) to determine whether their changes broke anything, and by isolating the build to their sources, or their branch, they are able to much more quickly fix a problem even if their preflight build took many hours to complete, because they can narrow down the changed parts of the code more quickly. Furthermore, they can be confident that their check-in will not introduce environment-related build breakages, since the pre-



flight build was run in the production environment.

At this point the entire team can decide to introduce a new rule: no one commits to the mainline of source code control until they've run a successful preflight build. Here you've introduced another part of agile development which will greatly reduce the number of errors in the nightly builds. With developers checking their own code against a full build run through the pre-flight system before committing, they ensure that most integration errors are removed before affecting the entire team.

For the team the great advantage of preflight builds is that the nightly or automated builds suddenly become more reliable. If developers are checking their code for breakages against a preflight build before checking in, the nightly build's reliability will increase greatly.

For the build manager there are two significant challenges to implementing preflight builds: automation and build resource availability.

The first prerequisite of an preflight build system is an automatic build, so tackling the first stepping stone is critical. It is, in fact, Fowler's first build-related practice and the cornerstone of continuous integration. The price of getting to a fully automated build (without automated tests at this stage) has to be paid up front and will be the most expensive (in terms of time) part of the move to continuous integration.

Secondly, the internal web page needs to be written. This is relatively easy with free web servers (such as Apache), and free tools (such as PHP and Perl) being widely available and well documented.

Lastly, as developers start to use the system (and they will if they are trying to be agile), the number of available build machines will become a problem. As many developers begin to ask for preflight builds the build queue may become long (especially if the build time is also long).

At this point the build manager has achieved agile builds. Developers can build whenever they need to, when code is checked in it gets built automatically (and quickly), and everyone sees the state of that build. If the build breaks developers can stop work to get the code integrating quickly.

The build manager may also choose to stop doing nightly builds all together. With preflight and per check-in builds, the nightly build may be a thing of the past.

Customer Story

Intuit Attains Truly Continuous Integration

Silicon Valley-based Intuit, Inc., is known for its series of highly intuitive financial software products aimed at consumers and small and medium-sized businesses (SMBs). The company's products include TurboTax, QuickBooks, Quicken, and numerous other titles. Each product had a large code base that previously could not be revved frequently because builds took more than three hours—which meant a build could rarely be run more than once a day. After searching for an effective solution, Intuit chose Electric Cloud products to automate its builds. It set up a build cluster consisting of 25 inexpensive x86-based servers on which the Electric Cloud software could leverage its parallel capabilities effectively.

The Intuit logo is displayed in a blue, lowercase, sans-serif font within a blue-bordered box.

For more information on this deployment, see: http://electric-cloud.com/downloads/EC-CS_intuit.pdf.

Today, according to John Burt, senior manager of SCM at Intuit, the company runs its build integration truly continuously: "With Electric Cloud, we compile and link every 30 minutes." And because of the quick turnaround on fixes, the costs of broken builds "are a thing of the past." The company's current practice is to use the last good build of the day as the clean, fresh build from which developers begin work each morning. Due to the fast turnaround Electric Cloud technology delivers, this build contains all nonbreaking check-ins and includes advisories to developers whose pending code generates errors.

Tools That Can Help

A good starting point when searching for continuous integration tools is the Wikipedia page entitled 'Continuous Integration' (http://en.wikipedia.org/wiki/Continuous_integration). There you'll find a long list of commercial and open source tools for building continuous integration servers.

To preserve the team's sanity and reduce downtime for developers and the build team when looking at tools or considering a build-versus-buy decision, you should aim to change as little as possible about your existing build environment. The goal should be to automate the build completely (so that it can be fired off from a single command) without changing the entire system. All that's needed to get continuous integration off the ground is a single command capable of running an entire build: the command needs to extract the sources from source code management, and perform the build.

Once that script is in place, the choice of appropriate tool really comes down to a few key questions:

1. Does the tool support scheduled builds? If the tool can't support 'cron'-style of builds then it's not worth considering. These are the most basic sort of builds and even when you've introduced agile development methods they'll still be important.
2. Is the tool scalable? One of the characteristics of agile development is that enormous loads will quickly be placed on build resources as engineers start running their own builds. And those resources will be even more stressed when every commit starts a build. Scalability needs to be considered from the start to avoid having to change servers in the middle of the stepping stones to continuous integration. Any tool must scale as resources (such as servers) are added, and be able to make use of pools of resources simultaneously as demand fluctuates throughout the work day. Business demands should be considered as well, as requirements for additional target platforms or integration of new teams will place additional demands upon the system over time.



3. Does the tool provide reporting or analytical tools? With hundreds of builds per day, managing the output or even just the status of every build is a headache. What was simple with a single nightly build becomes very hard with builds every ten minutes, and it's vital to be able to understand the performance of build systems and how the load and use of the build servers are changing over time.

4. Is the tool easy to adopt? Avoid any tool that requires rewriting existing Makefiles or build scripts: it's hard enough to adopt a new methodology without being required to start everything from scratch. The tool should work with existing tools and integrate with the existing source code management system.

5. Will the tool enable pre-flight and per-commit builds? Since these two styles of build are fundamental to continuous integration, the tool must make it easy to integrate with source code management to detect changes to the source base and automatically start builds, and to allow any user to start a build at will.

6. Does the tool support access control? Since the build servers will now be shared with all of development (and not with the small build team alone), access control is vital to ensure that developers have access to the appropriate resources from any location whether local or remote, and that the build team is able to override running builds to perform high-priority builds at will.

7. Will the tool support multiple teams or will each group or division require its own investment in a tool (plus configuration and maintenance)? Once automated, many build procedures (such as the job that monitors the source code management system for changes) will likely be generic in nature and easily shared across the organization. To maximize investment in a tool and reduce duplicate work, can common procedures or company standards be rolled out across teams?

In many, if not all cases, a homegrown approach will fall short in one or more of these areas. Or, if the homegrown system is adequate at the outset of the stepping stones to continuous integration, it may fall short over time as the load increases, the number of target platforms increases or as the number of users increases. Commercial tools will also pay off in terms of the reduced administration and maintenance required of a manual set-up.

Conclusion

Extreme Programming and Agile Development are being adopted by many software engineering organizations. Achieving agility is not easy, but teams can realize some of the benefits of agile development with a step-by-step approach. Careful planning is required by all parts of the engineering team with a special focus on the build resources. The build team will come under enormous pressure once agile methods are implemented and a step-by-step approach to implementing agile builds is recommended.

Careful selection of appropriate build tools is essential from the start of the roll out of any agile method. Those tools must be able to cope with the varied demands on the build resources and with the likely growth in the number and size of builds.

Enabling Agile Builds With ElectricCommander

ElectricCommander® is a tool to consider for implementing an agile production process within an enterprise environment. It is a Web-based solution for automating software builds and other production tasks to enable agile, iterative development. Only ElectricCommander is simple enough to use on a small build, yet scalable enough to support the most complex software production environments. It requires minimal process changes to get started or to deploy across teams. It provides the reporting and visibility organizations need for compliance efforts and release planning, plus an unprecedented level of flexibility and customizability to suit the agile organization.



For agile teams ElectricCommander enables:

- *Fast builds*: Run multiple procedures in parallel on the same resource, or distribute procedures or individual steps across any number of machines for faster turnaround and more efficient production. For additional speed improvements to the build step, ElectricAccelerator provides a fine-grained parallelism to speed builds by as much as 20x (see below).
- *Automated and scheduled builds*: Schedule production procedures to run at a specified day or time, on an hourly or daily basis, or whenever someone checks in code to a specified repository or branch.
- *Preflight and per-commit builds*: The underlying information architecture allows build and release teams to organize production assets into virtual, access-controlled projects, while access to production machines can also be limited or controlled. In this way, a release team can provide developers with dedicated or specified resources for preflight builds, as needed. Further, ElectricCommander supports integrations with leading SCM tools to facilitate per-commit builds.
- *Visibility and reporting*: ElectricCommander's unique analytics provide valuable insight into the details of the build, not just success or failure. The analytics engine extracts information and stores it as persistent properties of the job step, providing easy access to pinpoint statistics and trend reporting.

About Electric Cloud

Electric Cloud is the leading provider of Software Production Management solutions that accelerate, automate and analyze the software development tasks that follow the check-in of new code. These include the software build, test and deploy processes.

The reality is that homegrown systems are expensive to maintain and difficult to standardize across an enterprise, and are typically not able to support the frequent iterations at the core of Agile Development. Electric Cloud makes it simple to achieve a scalable, agile software production environment, across the organization. Electric Cloud's product suite – ElectricAccelerator, ElectricCommander and ElectricInsight – improves development productivity and product quality by accelerating, automating and analyzing the entire software production management lifecycle. In addition to the ElectricCommander production automation tool, Electric Cloud offers:

ElectricAccelerator

ElectricAccelerator® accurately executes parallel builds across a cluster of inexpensive servers to reduce build times by as much as 20x. ElectricAccelerator plugs seamlessly into existing build infrastructures, without modifying existing scripts. Faster, more accurate builds dramatically reduce the time developers spend waiting for builds to complete, and enables them to do complete builds before checking in their changes.

ElectricInsight

ElectricInsight® is a software build visualization tool that, when used in conjunction with ElectricAccelerator, provides job-level detail into parallel builds and provides unprecedented visibility into build results for simplified troubleshooting and performance tuning.

Leading companies such as Qualcomm, Intuit, Motorola, and Expedia have trusted Electric Cloud's Software Production Management solutions to change software builds and the entire production process from a liability to a competitive advantage. Learn more about Electric Cloud at www.electric-cloud.com.



About the Author

John Graham-Cumming is Co-Founder of Electric Cloud. Prior to Electric Cloud, John was a Venture Consultant with Accel Partners, VP of Internet Technology at Interwoven, VP of Engineering at Scriptics, and Chief Architect at Optimal Networks. John holds BA and MA degrees in Mathematics and Computation and a Doctorate in Computer Security from Oxford University. John is the creator of GNU Make Standard Library and has six pending patents in the build space.

Corporate Headquarters

Electric Cloud, Inc.
676 W. Maude Avenue
Sunnyvale, CA 94085
Tel: 408.419.4300
Fax: 408.419.4399

European Office

Electric Cloud, Inc.
7200 The Quorum
Oxford Business Park North
Garsington Road
Oxford OX4 2JZ United Kingdom
Tel: +44 1865 487177