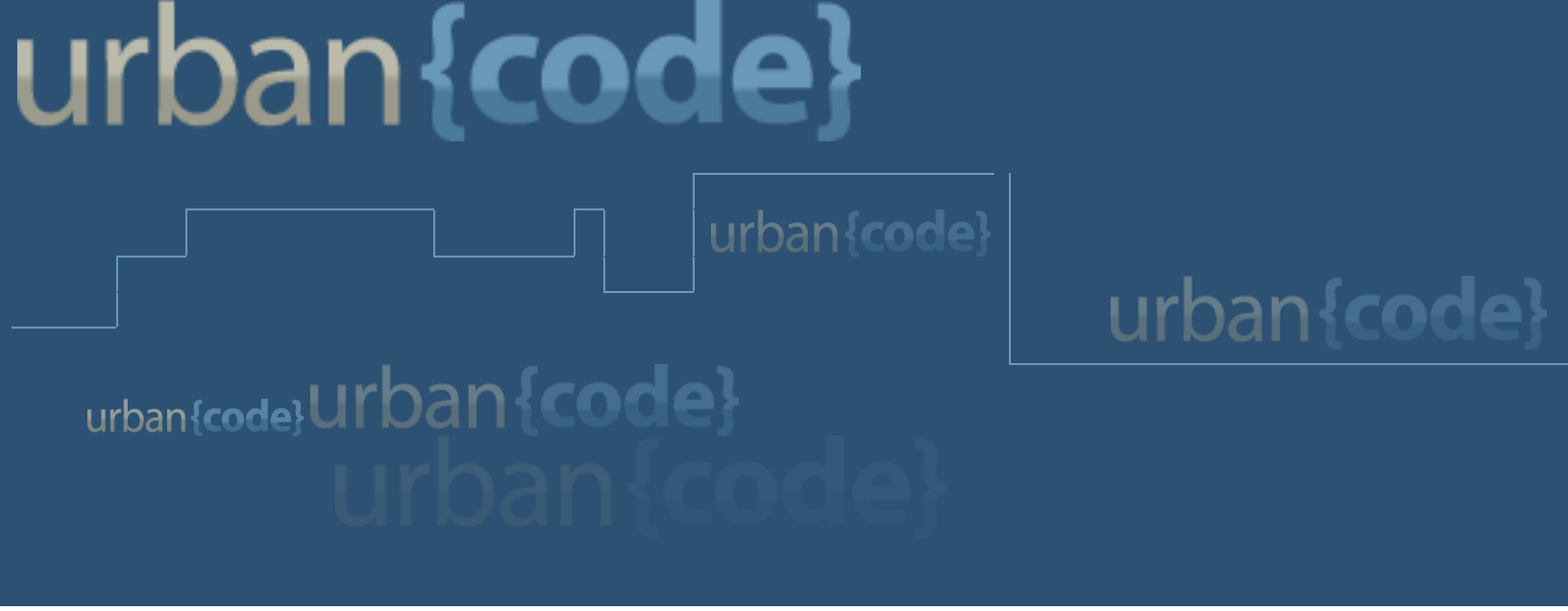


Build and Deployment Automation for the Lean Economy

Improving Efficiency by Removing Waste

by: Jeffrey Fredrick & Eric Minick

Urbancode, Inc. | 2044 Euclid Avenue Suite 600 | Cleveland, Ohio 44115 | P: 216.858.9000 | www.urbancode.com

Build & Deployment Automation for the Lean Economy

How can we get more done with the same resources? How can we get Lean and Mean without getting stretched too thin? This guide can help you answer these questions.

In good economic times teams attempt bold experiments that promise to take them to new heights of productivity. In lean times like these, few have the appetite for such speculative ventures. Any new investment needs to address immediate pain and show immediate payback. But the pace of demands hasn't slowed and the ever increasing need to do more continues. For build and release teams the demands come from all directions. The shorter cycle times of Agile means more builds, more deployments and more releases. The adoption of SOA means more complexity and more elements to juggle. The move to global, round-the-clock 24-hour development means more teams to service and less downtime. And in this economy adding headcount is likely not an option. Instead you're told to get Lean and Mean. But how do you get to Lean and Mean without being stretched too thin?

This need to keep costs fixed while adding capacity makes improving efficiency key. Instead of attempting a wholescale change to how you develop software we will take an idea from Lean Software Development and focus on reducing waste, allowing you to deliver more in the same time and for the same cost. Build and deployment automation offer fertile ground for dramatic productivity gains that will improve the efficiency of the entire team.

Removing Waste with Lean Software Development

Lean Software Development was derived through a combination of analogy, inspiration and sometimes direct mapping from Lean Manufacturing. Compared to named Agile methodologies such as eXtreme Programming (XP) or Scrum, Lean Software Development is more a set of principles than a set of practices. Mary and Tom Poppendieck laid out seven Lean principles in their book Lean Software Development: An Agile Toolkit:

Seven Lean Principles
Eliminate Waste
Amplify Learning
Decide as Late as Possible
Deliver as Fast as Possible
Empower the Team
Build Integrity In
See the Whole

Though relatively few people know it, the four values of the Agile Manifesto are backed by a set of twelve principles. A comparison between the Lean principles and the Agile principles will find little cause for disagreement. This means the ideas from Lean Software Development can easily be introduced to any development organization that has already adopted Agile, but at the same time Lean benefits from being distinct from Agile. Because Lean Software Development hasn't received the same level of advocacy as Agile and is firmly grounded in practical manufacturing practices, there is often less resistance to the ideas in organizations that are Agile skeptical.

For our purpose of increasing efficiency, this list is perfectly sorted: we're primarily concerned with the first principle, *Eliminate Waste*. Focusing on eliminating waste is a low risk approach to improving efficiency. This is because rather than facing the disruption inherent in adopting a new set of practices – such as Test Driven Development or a prioritized backlog – you can focus on improving the process you already have. This pragmatic approach is an easier sell than large scale change in uncertain times.

Seven Wastes of Software Development

Eliminate Waste is a simple enough concept, but for this to be actionable there needs to be a shared understanding of what constitutes waste. When Lean Manufacturing hit American shores, part of the impact was from a new and unfamiliar set of metrics. Similarly, Lean Software Development moves to the fore ideas that haven't been part of the traditional measurements in software development organizations. The list of wastes offered by the Poppendiecks was a translation from the seven wastes of manufacturing developed by Shigeo Shingo to describe the Toyota Production System:

The Seven Wastes of Manufacturing	The Seven Wastes of Software Development
Inventory	Partially Done Work
Extra Processing	Extra Processes
Overproduction	Extra Features
Transportation	Task Switching
Waiting	Waiting
Motion	Motion
Defects	Defects

A complete discussion of a software methodology would examine the sources of all seven wastes. But our aim here is simpler: To find the low hanging fruit we can reach through automation. The two obvious candidates from this list are Waiting and Motion. To help identify opportunities to remove these types of waste, we'll turn to two tools offered by Lean Software Development.

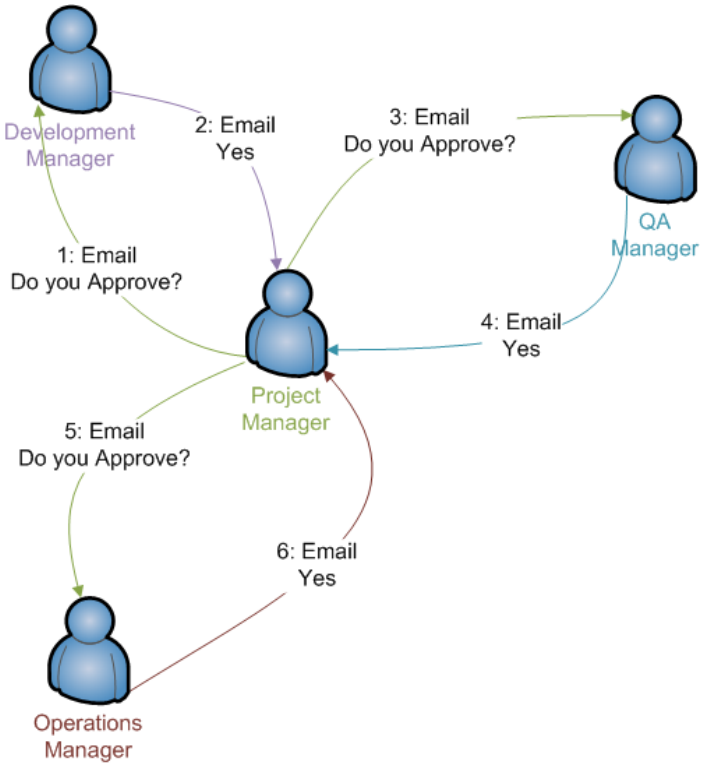
Tools for Visualizing Waste

The first tool we'll consider is something called a Spaghetti Diagram. Spaghetti Diagrams come from manufacturing to visualize the motion of an assembly across the shop floor. To create the diagram in a manufacturing context you start with a map of the factory and place your pen at the start of the process, then simply trace the journey of one particular assembly through the process being examined. If what you end up with looks like a plate of spaghetti, you've probably uncovered some wasted motion.

In the software application of Spaghetti Diagrams we won't usually be tracking physical motion around the office; but rather the flow and handoff of information between people and systems. Consider this simple process to get the sign-offs needed for a release to production:

1. Project Manager emails the Development Manager for approval
2. Development Manager emails the Project Manager
3. Project Manager emails the QA Manager for approval
4. QA Manager emails the Project Manager
5. Project Manager emails the Operations Manager for approval
6. Operations Manager emails the Project Manager

The corresponding Spaghetti Diagram shows the motion involved in the sign-off process:



The second tool we'll use is a Value Stream Map. A Value Stream Map shows the time spent to go through a process as falling into one of two categories: value being added or waiting for value to be added. A classically formed value stream map would also be a diagram, but the information is also nicely captured by a table. In our example of the sign-off process the value adding part is when the various people are writing their emails, while the waiting is the time spent until the next person reads and responds to the email in their inbox.

If the average response time is 30 minutes and the time to read and respond is a minute, then the Value Stream Map looks like:

	Waiting	Working
Project Manager emails Development Manager	--	1
Development Manager emails Project Manager	30	1
Project Manager emails QA Manager	30	1
QA Manager emails Project Manager	30	1
Project Manager emails Operations Manager	30	1
Operations Manager emails Project Manager	30	1
	150	**6**

Our Value Stream Map shows that we have a terrible ratio of waiting to working. Our Spaghetti Diagram makes it clear that the Project Manager is acting as a router between the other managers; the Project Manager still needs to collect all the sign-offs, but they aren't adding value in the intermediary role. With this information we might propose several different process changes to reduce waste, but the easiest change is to take that Project Manager out of the middle and have the various managers pass the sign-off email to the next in line. Our revised Value Stream Map shows the improvement:

	Waiting	Working
Project Manager emails Development Manager	--	1
--	--	--
Development Manager emails QA Manager	30	1
--	--	--
QA Manager emails Operations Manager	30	1
Operations Manager emails Project Manager	30	1
	90	**4**

This is a trivial example but it is enough to show Spaghetti Diagramming to illustrate motion and Value Stream Mapping to illustrate the time spent waiting.

A Lean View of Build and Deployment Automation

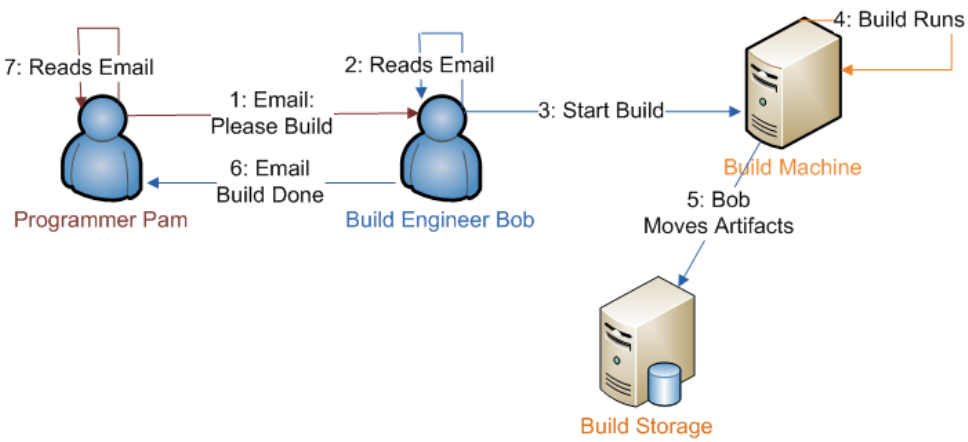
With that tiny introduction to Lean Software Development we're ready to consider the software build and software deployment, with a particular eye on opportunities to eliminate waste. We'll take build and deploy in turn, create a Spaghetti Diagram and Value Stream Map for each, and then look at the impact of automation.

Build Automation, Before and After

For the majority of the teams we visit we find the state of the art to be a build engineer armed with scripts who does builds on demand. This pattern is so common that we've given it a name: "Bob the Builder." You need a build? You email "Bob." Laying out this cycle starting with the build request from a programmer (Pam) the process is:

1. Programmer Pam emails Bob to request a build
2. Bob the Builder reads the email
3. Bob logs into the build system and runs the build scripts
4. Build scripts run the build while Bob monitors progress via the console
5. Bob moves build artifacts to network
6. Bob emails Pam to let her know the build is complete
7. Pam reads the email

Tracing this process with Spaghetti Diagramming we get:



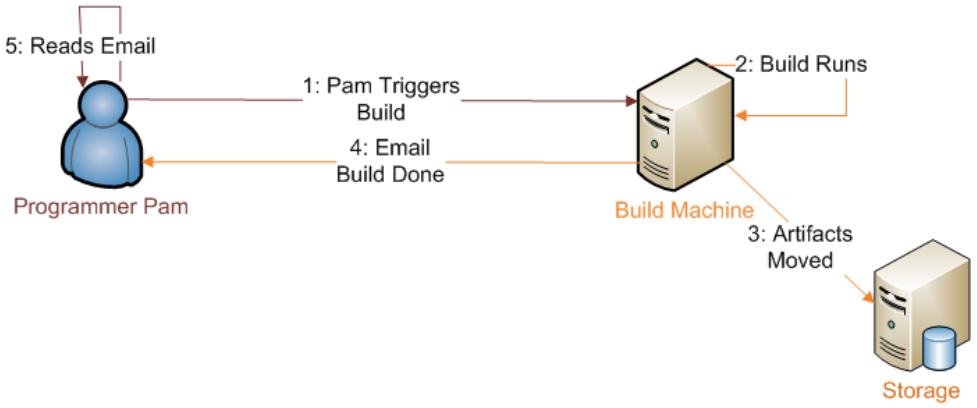
Just like the approval example, Bob is acting is a router or gatekeeper between Pam and the build system. There's also a good deal of back and forth motion between Pam and Bob that is potentially troublesome. In the real world, that communication line can become more cluttered as Bob asks for additional details around what should be built.

Where things get tricky is the Value Stream Mapping. It is easy to time how long the build scripts run, and how long it takes to copy artifacts across the network, but how long do we need to wait for Bob for each of the four actions he needs to take?

	Waiting	Working
Programmer Pam emails Bob to request a build	--	1
Bob the Builder reads the email	**?**	1
Bob runs the build scripts	**?**	1
Build script runs the build	0	10
Bob moves build artifacts to network	**?**	1
Bob emails Pam	**?**	1
Pam reads the email	1	1
	?	**16**

If we assume Bob is bored and has nothing else to attend to, then perhaps there is no waiting at all, and we end up at a fantastic ratio of only 1 minute of waiting for 16 minutes where value is being added. This might be the case when a project is just starting out and the builder role is played by someone on the team who is always available. But we've visited lots of teams, and have yet to meet many build engineers who have nothing else to do but wait for build requests. More typically they are running around putting out fires while mundane build requests queue up.

When Bob gets busy he becomes the bottleneck. The impact of automation is to remove that bottleneck. The Spaghetti Diagram shows that Bob is acting as the intermediary between Pam's request and the scripts doing the work. What if we change the process by implementing self-service automation that would allow Pam to request a build on demand? With that change the new process looks like:



Fewer steps, fewer handoffs of information means less motion and much of the motion that remains has been handed off to automated systems. Importantly the steps that have been changed are exactly the ones with the uncertainty in the Value Stream Map. With our changes the new Value Stream Map becomes much more predictable:

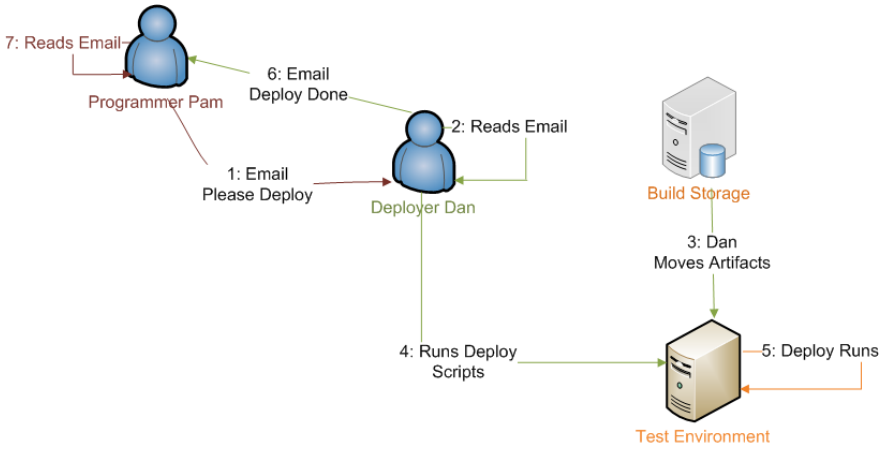
	Waiting	Working
Programmer Pam triggers a build	--	1
--	--	--
--	--	--
Build system runs the build	0	10
Build system moves build artifacts to network	0	1
Build system emails Pam	0	1
Pam reads the email	1	1
	1	**14**

The important win here with automation isn't just that the numbers are smaller: it is that they are also predictable. When using the manual process the person is a point of great uncertainty. Is Bob at lunch? In a meeting? Has he gone home for the day? Then you're going to be waiting. Even worse, the time in the cycle where timely builds become most important – approaching a critical milestone – is the time when Bob will be busiest, and the bottleneck will be the most backed up.

Removing Bob from our workflow doesn't benefit Pam alone. Because Bob isn't required to babysit each build request he can attend to all the other parts of his job, including introducing more automation.

Déjà Vu: "Dan the Deployer"

As common as it is to have build engineers providing builds by request we have found the same pattern with deployments to be even more common. The impact of waste in deployments can be even worse, as builds need to be deployed to several different environments before release. Typically builds are deployed in at least Development Test, QA Test, Staging and Production, but this list can include many others such as Performance Test, User Acceptance Test (UAT). And each one will need to go through a similar process where someone asks "Dan the Deployer" to move the build into the environment. Here's the Spaghetti Diagram for our programmer Pam requesting a deployment to the Development Test environment:



The parallel with the email driven build requests is almost perfect, and that's reflected with the same questions about the wait time for Dan's steps:

	Waiting	Working
Pam emails Dan to request a deployment	--	1
Dan the Deployer reads the email	**?**	1
Dan moves the build artifacts	**?**	1
Dan runs the deployment scripts	**?**	1
Deployment script run while Dan monitors progress	?	10
Dan emails Pam to let her know the deployment is complete	**?**	1
Pam reads the email	1	1
	?	**16**

As mentioned above this uncertainty in deployment wait times is usually repeated across deployment environments. Additionally, the different environments have different owners and even different steps for deployment. As a result a successful deployment to one environment doesn't act as a test for the downstream environments, and it isn't unusual for deployment problems to crop up very late in the cycle. The result is that if we bring in self-service automation for deployment requests, and we reuse our

deployment automation across environments, we not only reduce the waste of waiting but we remove a source of deployment defects as well.

	Waiting	Working
Pam triggers a deployment	--	1
--	--	--
Deployment system moves artifacts to environment	0	1
--	--	--
Deployment system runs the deployment	0	10
Deployment system emails Pam	0	1
Pam reads the email	1	1
	?	16

The benefits of this kind of self-service automation are amplified by distance, when there are teams distributed across time zones. In one example a customer of ours had an off-shore testing team that was working against on on-shore test environment. The time zone difference meant that when the test team requested a new build, usually "Dan" had gone home for the day. Likewise, when a new build was deployed, the test team was at home. The worst case was if there was a problem in the deployment and resolving the problem took more emails with 24-hour turn around. By introducing self-service deployment the remote test team was able to cut the turnaround time for deploying a new build from days to minutes.

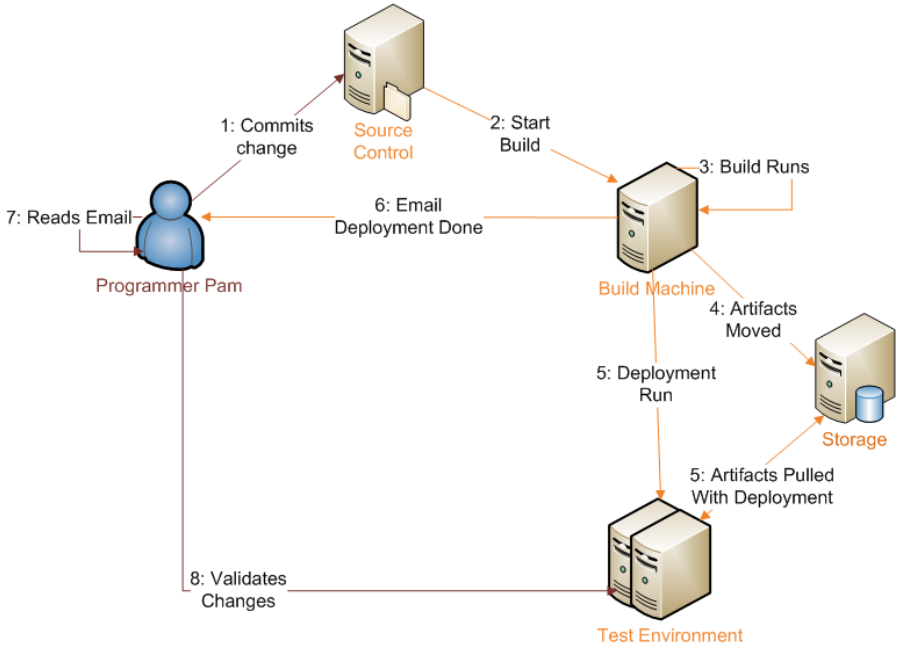
Lean, Automation, and Enterprise Continuous Integration

So far we've considered build and deploy as standalone processes. In reality they are both part of the same build lifecycle that runs from the time a developer commits until the code is running in production. Just as we've automated the build and deployment steps, we can seek to automate this end-to-end lifecycle. Extending automation through the lifecycle is something we term Enterprise Continuous Integration.

Enterprise Continuous Integration is something that has evolved over time out of the original XP practice of Continuous Integration (CI), and the original CI servers that focused on fast feedback to developers. Enterprise CI serves the entire development team by tying together the elements of Building, Deploying, Testing and Reporting. (For a full description of Enterprise CI see our Enterprise Continuous Integration Maturity Model whitepaper.)

An example of Enterprise CI is combing build automation and deployment automation, an emerging practice called Continuous Deployment. (While most teams practicing Continuous Deployment automatically deploy to just the first test environment, extreme teams automate the entire build, test, deploy process all the way to production. In these cases developer changes that pass automated tests are deployed to production many times each day without any manual inspection.) At first glance the Spaghetti Diagram for Continuous Deployment doesn't look very different from just combining the build and deployment automation diagrams given earlier. But a closer look shows that we've removed the

need for Pam to explicitly trigger either the build or the deployment. Less motion required, less waiting, and therefore less waste. This might seem like a small win but a developer we know who has moved to this kind of Continuous Deployment setup describes it as the difference between walking uphill and walking downhill; everything feels easier when the right thing happens by default.

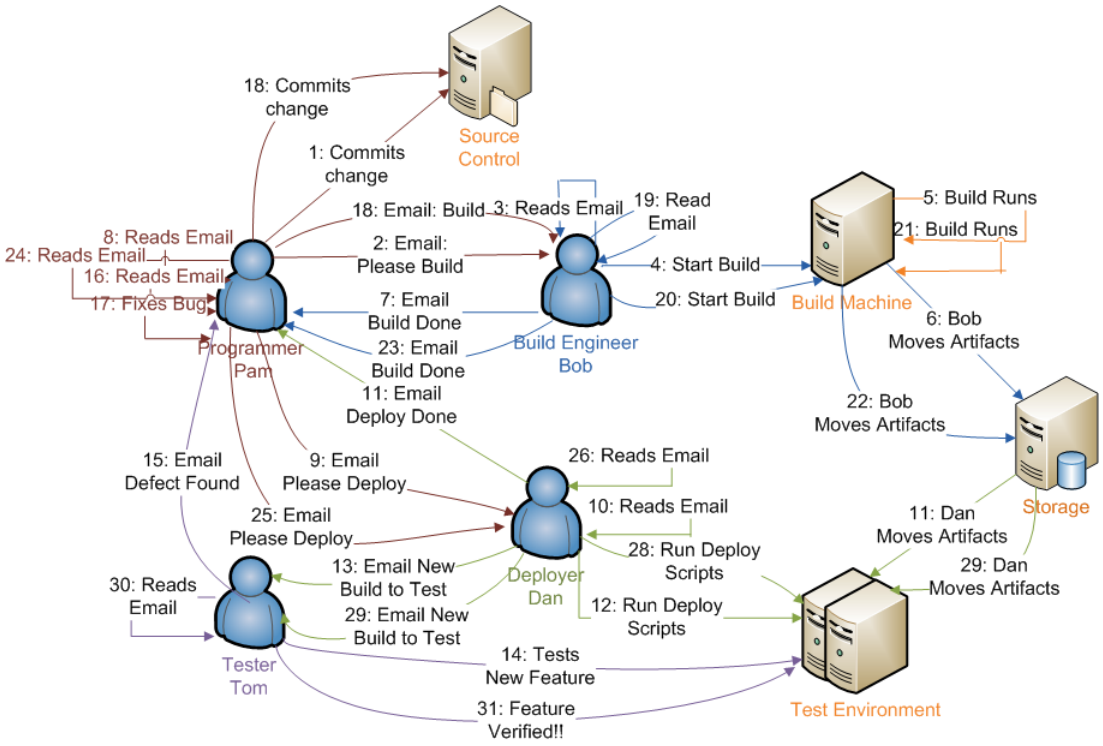


Removing Waste and the Extended Lifecycle

Enterprise Continuous Integration, with its broader view of the full lifecycle, offers a better vantage point to gage the true impact of the automation improvements such as Continuous Deployment. Consider a common scenario: a developer finishes work on a feature, a bug is found in the feature and a fix is made and verified. A summarized view of this scenario looks like:

1. Programmer Pam commits a feature
2. The new feature is built
3. Build with new feature is deployed
4. Tom the Tester tests the new feature and reports a bug
5. Pam fixes the bug
6. Bug fix is built
7. Build with the bug fix is deployed
8. Tom verifies the bug fix

This seems straightforward, but this seemingly simple process has the Spaghetti Diagram live up to its name:

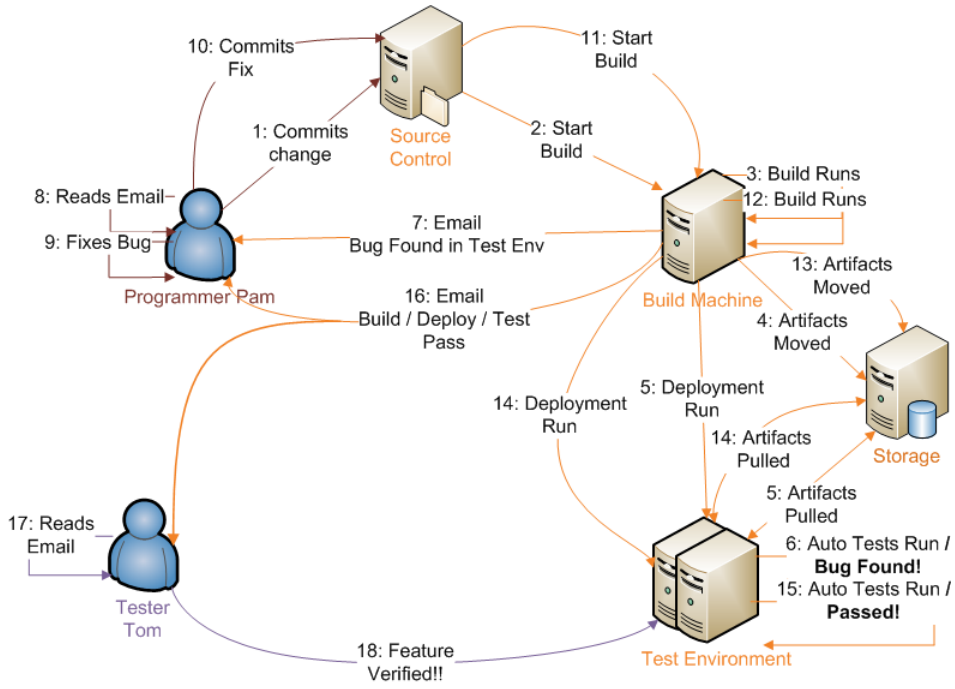


As ugly as this looks the numbers out of the Value Stream Mapping are even uglier. The numbers we're using here reflect the practices we commonly see in teams across the industry: builds happen once a day, while deployments to test occur about twice a week.

	Waiting	Working
Pam commits the new feature	--	1
Feature is built on the nightly cycle	720	15
Build is deployed to test environment	3600	15
Tom finds bug and reports it	240	120
Bug report reaches Pam and she fixes it	2880	60
Fix is build as part of the nightly build	720	15
Build with fix is deployed to test environment	3600	15
Tom verifies the bug fix	240	60
	12000	**301**

The full cycle described here takes just over 8.5 days, but of that only 5 hours are spent working. From a Lean perspective there is obviously a lot of waste to be eliminated here.

The obvious place to start is with the kind of build and deployment automation provided by Enterprise CI. If we were to replace the nightly builds and the twice weekly deployments with the kind of integrated continuous deployment outlined above we'll remove most of the waiting with one fell swoop. But Enterprise CI is about more than just build and deployment, there's the testing aspect as well. Not all testing can be automated but a mature Enterprise CI system should include automated unit tests, automated functional tests and static analysis as well. If the defect in question is the kind that can be caught by these sort of automated tests then Pam can be notified and fix the defect before Tom ever takes a look it. Not only does this save Tom time – he can skip a test cycle – but it makes Pam's job easier as well. If she can be notified about a defect soon after she commits her feature it will be a lot easier to fix the problem than if she were try to fix it a week later.



With continuous deployment, automated tests and rapid feedback we end up with a very different Value Stream Map:

	Waiting	Working
Pam commits the new feature	--	1
Commit automatically triggers a build	0	15
Successful build is automatically deployed	0	15
Automated test detects bug and notifies Pam	0	30
Pam fixes the bug and commits her change	60	15
Fix is built	0	15
Fix is deployed	0	15
Tom tests the feature	1440	120
	1500	**226**

If we look back at the list of the seven wastes of software development we can see that our adoption of Enterprise CI has eliminated waste in five of the areas. In terms of *Extra Processes*, we've saved Tom a test cycle, but we've also saved the entire bug reporting, triage, assignment process. Our programmer feels the reduction in task switching; she is notified almost immediately of the defect and doesn't pay nearly the cost of trying to remember what it was she'd been thinking when she was working on the feature. The reduction in *Waiting* from moving to continuous deployment is enormous, while a glance at the before and after Spaghetti Diagrams will show a huge reduction in *Motion*, the handoffs between the different actors. Finally, because Tom has more time to spend actually testing we can anticipate a reduction in the number of *Defects* that will escape to production.

We started with the observation that build and release teams are under increasing pressure to get more done with the same resources. Every team, Agile or not, is looking to improve their time to market and to improve their efficiency. We think the principles and tools offered by Lean Software Development, particularly the emphasis on reducing waste, offer a valuable insight into how build and deployment automation can pay huge dividends for the entire development team, particularly as it is extended across the lifecycle into end-to-end Enterprise Continuous Integration.