

Large Build Teams: Help or Hindrance?

Julian Simpson
ThoughtWorks
Berkshire House
168-173 High Holborn London, WC1V 7AA, London
jsimpson@thoughtworks.com

Shane Duan
Guidewire Software, Inc.
2121 South El Camino Real, Third Floor
San Mateo, California 94403 USA
shane.duan@gmail.com

Abstract

Should we use build and deployment teams on large projects? Build and deployment work often emerges as a specialization on project teams. This specialization becomes important on medium to large projects as the complexity of deploying code and configuring enterprise environments increases. However, how do we coordinate the work of this team with the work of the development teams and how do we ensure this team helps the development team that it serves rather than hinders it?

In this report we share the experience of an eight person distributed Agile build team on a large bespoke software project at an investment bank. The report spans the formation of the team from the USA, the UK and India to the adjournment of the team after finishing the deployment to production on time without fuss. Topics include: how we worked in a distributed Agile 8-10 person team in San Francisco and London, how we used agile methods to track build and deployment tasks; how we worked in iterations, estimated the work for each iteration, and tracked velocity over time.

1. Introduction

The authors were members of two project teams: a British team that wrote a broadband registration system, and a distributed UK/USA project team that was writing investment software for a global bank. Both projects were large and divided into different teams; both centralized the build and deployment efforts of the teams into a build team. One author worked in the build team on both projects. One author was a customer of the build team on one project, and the Iteration Manager of the build team on another. Both projects delivered, the build team was ramped down.

During the development process, developers wrote code and QA verified it. The build teams acquired many different technical responsibilities, including the deployment

of the application into test or production environments, release administration, and maintaining the ANT scripts and continuous integration system for building and testing the application.

Due to the vagueness of the role, the teams had initial difficulty delivering the tasks that were assigned to them. Following agile principles, the teams were able to identify the issues, tackle them in order and at the proper time, identify the success patterns and lessons learned, and continuously improve their process.

As the projects reached an end, the team conducted retrospectives to capture the experience and the conclusion they reached

2. Build Team

In both projects, a central build team was responsible for managing:

- Continuous integration system for several development teams
- Deployments to formal test environments
- Ad-hoc technical support for the developers and testers

This specialization allowed developers to focus on writing code. It made sense to merge the effort of build and deployment; and leverage the skill of the full-time build engineers. In both projects, the build team was considered large at the time, but reducing the size of the build team was not an option until there was a track record of deploying the new code to a production environment. In both cases the scope of the team's work was very wide. The phrase "Build team" did not really capture all of the activities that the teams did. Really the teams were there to ensure that the code could be built, tested and deployed to production.

There were three major areas of activities:

- Build activities: maintaining a build system, Continuous Integration, environments for the continuous integration environments to deploy to
- Testing Support: deploying to formal environments if there is no self-service process or if privileged access is required
- Production deployment: maintaining an accurate release system, having contingency plans, etc

There is a fuzzy trade-off between these activities. In addition to this work there are often other services that the team will provide, like assisting with credentials and database schemas.

Perhaps a change in name might help to reinforce the importance of deploying to production. One of the key themes of this report is the priority of deploying to formal environments as opposed to the activities around building code. We should not even need to call a deployment a “build”: we do not often build code to deploy it, as the artifacts from the Continuous Integration process should be available to deploy to every environment.

One of the teams was called “Shared Services” and that name was (independently) proposed for the other; it is a suitably vague name that allows the team to acquire a mixed bag of services to provide.

3. Agile Development in the USA

The development of build and release is different from an Extreme Programming style development due to the nature of the project. This turned out to be a good example of shaping the process to fit the team and environment, because the team came from Extreme Programming style agile projects, we all felt a strong need to keep the practices as much as possible for the benefits that it could bring.

The nature of the project still fits into the agile software development: A group of people with different roles and skill sets working together towards a common goal: to bring value without driving up the costs. The team focused on iterative development, retrospectives, and backlog. The team identified the items, laid out the tasks and reflected constantly about how to make things better.

3.1. TDD and Pair-programming

UNIX shell scripts (the deployment *lingua franca*) are generally very hard to test. Some tasks are just not cost-effective or feasible to do in pairs. Access rights were not granted equally to the team members, so some people had to specialize on tasks like building production systems.

After we identified the tasks, we talked about the necessity to work on them in pairs. For each story, we also

talked about how we could test to make sure that it was implemented correctly. For some functions that were used throughout the deployment system, we did write tests that we could run manually. We also changed some part of the build system to be implemented in Ruby so that we can write less code and have better control of the process flow.

3.2. Stories and Backlog

Due to the nature our projects, stories were often open-ended and had their fair share of ‘Yak Shaving’. It is very common to have a story, or a task on a story described as “have a first meeting with client, find out the process of performing a manual task, and document it so that we can create more stories”. This made the stories hard to estimate and sometimes hard to identify the acceptance criteria.

However, making constant progress is the key activity for a project. With these stories, the team was able to make progress by understanding the issue and creating more stories. In the world of build, release and deployment, we have identified the stories as a group of action items that made sense to be executed together and were also small enough to finish in a few days. They were estimated in ideal days. Sometimes the estimation process was very slow as we had to discuss the story in some detail before we could estimate.

For each story, we tried to identify the acceptance criteria as well as the customers. Sometimes the customers did not even realize the benefit of story or that they could be the customer. After we finish the story we would go to the customer asking for sign-off. By getting the customer involved in the process, we were able to get the feedback and made sure that we were providing the value to the program. For stories requested by the customers, we would invite them to our iteration kick-off meeting.

Getting involved with the customers also helped us identify the missing stories. At one point, we hosted a brainstorming meeting. We first wrote up anything on our mind on a post-it and stuck it on the wall. Then we put the ones related to each other closer so that we can identify the stories. After that, we prioritized the stories based on our understanding. The external customers were not involved in the process like a typical agile project, but it still provides a baseline for our work.

3.3. Identifying the Customer

The customer of the story would not always appreciate the benefits they might gain, or be willing to invest their time. For example, it is very cost-effective to have the developers and QA work together deciding when to deploy the last successful build to the testing environment because it will tremendously cut down the turn-around time for the bugs and help quickly identify the ones that

are environment-dependent. However, the complexity of the deployment process had always seemed intimidating to them and they wanted the help of the build team. We played a series of stories that greatly simplified the process and made this possible.

It was often difficult to identify the customer. Sometimes the customer was the manager of the build team, sometimes a project manager of the development team, sometimes a developer or even the build team itself. The team identified the customers of each story, and never considered a story as done until the customer had signed off the story. The feedback from the other teams were also converted into stories. By identifying the customers and sticking to the acceptance process, the team could get quick feedback on the impact of the stories. Interacting in such a way greatly enhanced our relationship with our customers, which allowed us to influence them in a positive way.

3.4. Communication

The team came from the USA, India and the UK. Two development teams had members in the USA and the UK, and the build team maintained a presence in both offices. This caused complications with morning stand-ups and Iteration Kickoff meetings: stand-ups had to be early in the USA and late in the UK, and held over a telephone call. The UK based team members were not able to participate in the estimation for iteration kick-off due to the time difference and the interactive and visual nature of the process - a large part of it was conducted around a board estimating tasks using flipcharts and sticky notes.

We took different measures: We sent an experienced team member to the UK office, and imported another from the UK so that majority of the team is collocated, which helped greatly. We also tried to identify the stories that were isolated for other team to work on. Each team would have a hand-off meeting at the end of the day so that the other team could take over the appropriate tasks.

3.5. Iterative Development

This was the part that we did not need to change at all. At the beginning of the iteration, we would have a retrospective about what we have done well and what we still needed to figure out how to improve. After we closed the last iteration, we kicked off the next iteration with the stories that we think we could finish. We tracked our velocity to our best ability so that we could detect any issue that might come up.

4. Negotiation

In a way, the management of the team's work was like a different kind of negotiation, i.e., negotiating with the other

team for what you think the build team should do instead of what other teams wanted. The team needed to resolve the immediate problem for the other teams in order to provide the immediate value and gain the trust. At the same time, the team needed to negotiate some time so that they can work on the long-term goal of a successful deployment.

Using the trust that the team earned as leverage, the project manager slowly pushed back work that should stay with the development team thus guiding the role of the team in the right direction.

5. Specialization

“Cross-functional team” is a popular phrase when it comes to agile development projects. Yet at the same time, we still have the role of developers for software development, Quality Assurance (sometimes called testers) for acceptance testing, and Business Analysts for gathering of business requirements. For database related projects, we have Database Administrators that help with database design and performance tuning. All these indicate that specialization to a certain degree brings enough benefit to outweigh the risk of having a bottleneck.

Because our build project is part of a program, supporting several other projects at the same time, we have found out that our specialization brought in enough value to be worth the overhead.

One of the key reasons for the split between developers writing code and another team deploying it seems to be the UNIX platform. You often require privileged access to the system to deploy code. The default corporate desktop of choice being Windows makes it harder for developers to become proficient as Unix users. This stimulates the requirement for a specialist team, which creates a vicious cycle.

Here is a list of some common tasks that we attended to during our project: build machine set up and performance tuning, source control administration, scripting for deployment, ANT, design for security and performance, setting up continuous integration systems, and build and deploy patterns. All these tasks require the person who does the initial work to know enough about the system to figure out the right approach for the project. At this point, it is more efficient for a build engineer to do the job, even with the overhead for developer feedback.

With a specialized team focusing on build and deployment, it was a lot easier for us to spot the patterns that can be shared by the projects.

Once a project is set up and the process is running, a person simply needs to understand the parts involved for the project to maintain them. At this point, it makes more sense for the development team to take over in order to fine-tune the process. For a large project or a program with multiple projects, it is common that the process will need to be

changed for new deployments or testing requirements. Fortunately for these environments, there will always be a build team on site to handle release related tasks. At the same time that we worked hard to answer the request and work out the solution, we also tried as hard to hand-off the final tasks to the project teams themselves.

6. Developer Incentive

At bank in the USA, the development teams had established agile practices like story-driven development, iterations, and continuous integrations. The developers signed up for stories and bugs, measured and maintained velocity. This goal, however, turned out to be misaligned with the long-term goal. Developers ended up taking short cuts and have the build team pay for it.

For example, developers only needed to care about making the build and test pass on their machine in their IDE. When the Ant build or test fails, they had no incentive to fix it and expected that the build team would make the fix. This took a lot of time from the build team because they did not know anything about the code. This also led to a poor project structure that generated difficulty in maintaining the build scripts. The harder it was to maintain the build scripts, the less likely that the developers want to touch them even when the build was broken. Another vicious cycle was formed.

From the build team's perspective the application code was written very quickly, and with little consideration to the eventual deployment, or the workload of the build team. In a way, the development could have claimed to have "achieved agile", but by sacrificing the build and deploy process. What is worse, the project was scheduled based on this false sense of speed, which brought tremendous risk.

In retrospect, we concluded that this happens when developers did not have any incentive to make deployment easy. We think there are two other incentives that the developers have related to this integration and deployment business.

One incentive for developers is to make sure that the application works in a production-like environment. The only way to be sure is to deploy the application as soon as possible. With an efficient deployment process, developers can have the power to see the application in action whenever they want, closing the feedback loop on the development.

Developers generally have an incentive to triage defects soon as possible before analyzing the root cause. Since some defects can only be completely verified in a deployment environment developers need to be able to debug the application, change the code or configuration, and redeploy the application quickly.

7. Automation

"You might be tempted to delay automating the build until your project has amassed enough code worthy of the automation. Unfortunately, that's usually too late." [1]

Both projects initially suffered from manual deployment. The broadband project had no deployment process at one stage, and the build team set about deploying the code to a clustered J2EE server by hand. This generated some documentation that slowly turned into a set of ANT and bash scripts to automate the deployment. The banking project had some build automation and the team had generated documentation. The documentation so closely resembled a bash script that the decision was taken to make it a bash script. It was copied verbatim into an editor and distilled into executable script.

We have found out that even though the build and deployment automation does not come without a price, the benefit definitely makes it worth the cost. When you can push a button (or invoke a command line) to have the whole application redeployed to a full featured environment with nothing stubbed out, you can make a lot of other things happen.

At the beginning, we had a manual process. This process was not understood fully even by the team, let alone the developers who just run the application through an IDE and check in the code. We reached automation slowly through several phases.

- We documented the process in a wiki page, and rotated the deploy tasks so that everyone understood it.
- We started with the steps that could be easily converted to a deployment script, e.g. back up the installed application, copying the file to and downloading the file from release area,
- After fixing the small things, we proceeded to add some intelligence to the code through close teamwork. For example, we could only deploy applications that were compatible, i.e., they had to depend on the same build of the shared libraries. One engineer wrote a Perl script to check the versions, and then used it to generate an HTML page that could be viewed on-line. At the same time, two other engineers wrote a simple web application to initiate the release. They put these two together and got a release process. With this process, the user just needed to go to the web page to look at the builds that passed the test and released the matching products together with minimum effort.
- After fixing most of the problems in the deployment script, we started getting the developers involved.

They learned how to deploy the application themselves, which gave them the power to control their testing environment.

- Once the developers could do the job better done by themselves, we freed ourselves to take the feedback and improve the process. For example, we learned that with SourceForge Enterprise, we can associate the change sets with the bugs, and associated the bugs and new feature requirements with the releases. Therefore, we modified the script so that all the tag names and directories were based on the release name. With the new naming system, it was pretty easy to generate the link so that the user could look at all the releases available to deploy and the features/bugs related to each release.

8. Staff Up, Staff Down

In each case, the build team was initially unable to control its own workload: QA would request deployments or intervention to assist them in testing non-functional requirements. With each major release the build and continuous integration system would need to be reconfigured. In some ways the team became a proxy for the IT department and assisted people with credentials for SCM systems, etc. There was also a constant firefighting effort as deployments failed in new and interesting ways. In the second case study there was enough forward thinking done such that an agile planning process was introduced.

It proved to be impossible to just use “roles and responsibilities” as the reason to refuse doing the work or put out the fire that were not necessarily the build engineers’ job. It would only make things worse because it would increase the tension between the teams, encourage negative politics in the projects, and worst of all, leaving more and more broken windows.

A good lesson that we have learned in resolving the workload issue was “Staff Up, followed by Staff Down”. We were overloaded for some time before we were able to increase the team size. For a while, the way out was not clear. Through a chain of events, we expanded the size of the team and were able to take care of the daily business. With room to breathe, we started to look into ways to better use our time. We focused on two areas: to make process more efficient so that we can deliver more in less time, and to identify the work that should not be done by the build team in the first place.

We improved the processes so that they were no longer intimidating, and we trained the other teams to do it themselves. For example, with the experience that we have gained by putting out the fires on the continuous integration server, we identified the issues and came up with solutions. We implemented some of them so that the devel-

opment team can start taking the responsibilities of keeping the continuous builds passing. We also helped by monitoring the builds and provided our recommendations to the developer teams. Our Iteration Manager brought up the phrase “We’re just dumb robots” from time to time to remind the developer team members that they also had a responsibility to take on.

9. Frequent and Early Deployment

At the ISP, we did not deploy very early at all. We used continuous integration process, and had a build team of several people looking after the care and feeding of the continuous integration system, juggling databases for developers and continuous integration, and the deployment to standalone systems. When we were allocated test hardware we then had to implement a deployment process. The first deployment took two people two weeks to finish. The code had been developed against another project and had a runtime dependency on that project. Introducing clustering also broke the deployment as all the testing had been done on single hosts.

The deployment was seen as a constant risk to the project because it appeared fragile as it evolved with the project codebase. Each day two members of the build team would add features to the deployment tool as they were requested by stakeholders (for example, generating UNIX init scripts). Other members of the team were running the continuous integration system and version control, applying a new configuration system to the project, providing technical support to the developers and testers, deploying builds to the test environments, and a having lot of communication with the various stakeholders.

With such a large team we were able to make production deployments a predictable process, and after the first major deployment to production the organization was able to reduce the team size considerably. Had the deployment effort started earlier, some issues would have had less effect on the delivery.

Implementing a deployment process early in the project does not mean that you need to design for everything from the beginning. Just like any other agile software development, the build team just needs to write enough of deployment to handle the immediate needs of the project. This means that the build and deployment script is being changed all the time. Unlike software product where a team of QA and business users are allocated to make sure that the product works correctly and meets the user’s needs, the only way to know for sure that a deployment script works is by putting it in action. The more deployments, the better the feedback.

At the bank, as described above we manually rolled out our applications and automated small steps at a time. We became good at writing documentation and scripts that we

were able to create a self-service approach for the developer and QA teams. They acquired the correct access and followed instructions to deploy the application when they wanted to. We could have attempted to use the same production deployment process from the continuous integration system. This would certainly have provided value at the ISP project because we had a disconnect between the way we deployed to a J2EE server (ant tasks), and the way we deployed to production and similar environments - in the latter case we installed a complete J2EE server alongside the existing applications so we could reduce the cut-over time. Rehearsing that process would have reduced the risk eventual deployments to production because we would have tested the deployment process before deploying to a formal test environment.

10. Conclusion

The build team is a necessary part of a large project, but raising an organizational barrier between development and a build/deploy team comes at a price and a risk. The build team can end up as the owner of the build and release processes, changing the build scripts and deploying every release in the building. This can reinforce the importance of the build team to the other teams, thus perpetuating the role of the build team long after the original issues that led to the formation of the team have been resolved.

Any specialists on an agile project team should be collaborating with the rest of the project team instead of negotiating verbal contracts with other groups. We would always recommend that a deployment expert be enabling a team to manage their own continuous integration and deployment rather than trying to serve them. The build team should be trying to make it safe for people to learn about the specialist activities that they undertake, regardless of that person's experience and background. In our experience, trying to supplement the activities of a developer team can be like drinking from a fire hose; a small team cannot deal with the backlog of technical debt from a large team, it is much better to try and work alongside a team.

On the projects we worked on, we suffered from a myopic view of the software development lifecycle. On one, we decided that work was complete before we could deploy it. Learning our lessons from the previous challenging project, we made sure that the stakeholders would sign off on work done at a production-like system. Ignore that one at your peril - being able to deploy to a production system is something the entire team, from the program manager down to the janitor should care about deeply.

We do not believe that there is a case to delay deployment and automation of the deployment process. Likewise, at the start of a project there is time often time to experiment with the build and deploy process that we can guarantee will not

be available at the end. So use that time wisely and have a good go-live!

References

- [1] M. Clark. *Pragmatic Project Automation*. The Pragmatic Programmers, Raleigh, NC, 2004.