Agile Architecture – Changing Application Servers

Veljko Krunic
Rally Software Development
krunic@ieee.org

Abstract

Some projects are simply too big to finish on the release schedule that you want to maintain. Rally’s agile engineering team delivered a very disruptive architectural change lasting nine months in parallel with three releases. This paper will show how this was done without prolonging the usual release cycles and without technically crippling our product or doing a lot of throwaway work. The main elements of our approach were: the "Sacrifice One" pattern, incremental delivery of the architectural change when possible, technical infrastructure modifications to support incremental changes to the architecture, and addressing the biggest risk first. When we were close to finished, we added the whole team for the final push to release with the new architecture. The end result was three successful releases of the existing system prior to the final release rolling out the new architecture.

1. Introduction

Rally Software has built a Software-as-a-Service tool that unites Agile project management with tracking of requirements, tests and defects. In September of 2005, Rally Software decided to move from using a proprietary J2EE application server and portal framework to the open-source JBoss [18] based infrastructure due to the high licensing costs and difficult unit testing when using the proprietary server, and the fact that we felt the proprietary server was not superior to JBoss in the features we used. That involved not only changing our application server, but also writing new infrastructural components and significant rework to break dependencies on the proprietary stack. We had work with fixed scope ("remove all dependencies on old system") to be done with a fixed team.

There was no way to accomplish the entire JBoss transition as a single release. We had features that had to be delivered to customers roughly every three months, and the JBoss transition didn’t fit into that time-box. We would not be agile if we accepted that fact and said, “We will have a nine-month release cycle and no demos in the meantime.” We needed to move forward on the JBoss release while in parallel delivering multiple releases with the old application server.

We had to find a way to introduce a very intrusive technological and architectural change that could not be released piecemeal in an agile way, while following a moving target – new customer-facing features introduced in our software during the transition. We were in a marathon, but we still wanted to be able to do series of (Scrum [11]) Sprints.

2. The Marathon: As Seen by Reporters

There was a limit to how many people could work on the initial transition to JBoss, but at the same time, we couldn’t work on GUI-related tasks until the back-end of the app was working on JBoss.

We decided to use a single, full-time developer to start this work for our “Sacrifice One” [1] pattern. We planned to ramp up the size of the team working on JBoss in the later stages of the project [2]. That meant one team member was permanently on a task that was not the highest priority of the organization, but because it was the highest priority of that one developer, it got a lot of attention.

“Sacrifice One” applies to one programmer, but could we use “Team per Task” [1] and apply it to to a pair or a team of programmers? We wanted to finish the initial feasibility investigation with minimum investment. One recent hire wasn’t fully up to speed on the product, but was otherwise well-qualified for all transition tasks in a first few months. Due to that and the fact that not all of developers in Rally practice pair programming all the time, we decided to put one developer on the task (as opposed to the pair), and do informal knowledge sharing sessions with other members of the team, so that knowledge was not concentrated within a single person.
With our selection of runners completed, it was time for our marathon. In further race descriptions, the whole team running for the Rally will be referred to as “the runner.”

Start of the Race: The runners were eager. We determined that the first tasks were to get all unit and in-container tests to pass and have the application deployable in JBoss.

Mile 0 to 2 (first two weeks): Our runner was in the leading group. The build system was modified to build either on the old platform or on JBoss, depending on a property setting.

Mile 2 to 6 (weeks 3 to 8): Our runner was alert and deciding on the best strategy for the race. The majority of architectural decisions impacting the transition were made in this time period. Infrastructure was written to allow use of the same source files both in the old and new platforms. We had our first demo with a minimal feature set on JBoss and one release on the old application server.

Mile 6 to 12 (months 2 to 4): Sustainable pace. Our biggest technical risk was always scheduled first. Once we deployed a minimal application on the JBoss server, we identified our biggest risk and addressed it as soon as technically possible. If new infrastructure features from the JBoss version could be used easily on our old application server, we immediately transitioned both versions to use it so that we could try new features fast on the market in order to prove infrastructure in the field. This not only reduced risk, but we also used the opportunity to refactor existing code and significantly reduce our technical debt. We shipped one more release on the old application server.

Mile 12 to 17 (months 4 to 6): Our runner tripped and fell. Due to the “next release” pressure, we were forced to remove our “sacrificed one” from JBoss to work on new features. We were able to recover fast after that, but some time was lost. We shipped one more release on the old platform.

Mile 17 to 26 (months 6 to 9): Our runner was in the lead, and victory is close. When we had done enough work to be reasonably confident we would succeed, the whole Rally team moved to JBoss. All new features were implemented on JBoss only.

The Finish Line: At the final sprint, we won! The JBoss release is shipped to the market. Quality was high, performance was good, users had positive feedback, and we were able to add significant UI improvements in the product. We had three releases on the old platform and one JBoss-based release in the period of nine months.

3. The Marathon: As Seen By Runners

Now we know how reporters saw the race. Let’s look behind the scenes. What did engineers running the race have to face?

3.1. Architectural Decisions

There are multiple views on what software architecture is [12]. For the purpose of this discussion, we define “architectural” as: any design decision of high enough consequences that it could cause cancellation of the project if done incorrectly [8].

We took the approach that due to the high impact of the decisions made early on, anticipation is better than reaction. We used a lightweight approach to design, similar to the one described by Fowler [10]. We had informal sessions around a whiteboard or at our desks, made a decision and enumerated a list of risks we were concerned about. Then we made a quick prototype to check architecture, in effect testing architecture by making simple code implementation.

In addition, when practical, we preferred postponing architectural decisions until the “last responsible moment” [13], as opposed to making them in advance “because now we are doing architecture.” If we were not sure that the impact of the architectural decision would be immediately felt, we would generally postpone it for a later time. As a result, architectural decisions interleaved with development to a large extent and were made “just in time.”

We spent about 5% of the total time on the project (in man-hours) on architecture and its prototyping. We faced only one situation in which we had what we perceived as a bad architectural decision, and it was fixed within three person days of work. We are not claiming that 5% is optimal on any project – just that it was how much time we happened to spend on our project.

3.2. Build System

While we were transitioning to JBoss, we shipped three releases using the old proprietary application server to the market with a host of new features. All of these new features had to be supported in the JBoss release. Most of code was shared on both platforms, but there was a significant amount of platform-specific code as well.

Our choice was to either keep a single code base that would be used to build both versions, or branch the code base and develop support for JBoss in a separate branch. Added consideration in our decision was that we were using continuous integration and a lot
of automated in-container and unit testing. Some of the testing was done manually. Obviously, the choice we made had to support automated testing and continuous integration, but different decisions had different impact on manual testing.

If we chose to have separate branches, we would have to do occasional “upmerges” [4] from the old platform branch to the JBoss version. This would isolate the releases we would be sending to market from the JBoss release we were developing, leaving us more time to test the JBoss version before we delivered it to customers. Having to do these periodic “upmerges” would have meant the two versions were not in sync all the time. It was likely that a significant amount of time would be spent on integration. Furthermore, as most of the team would have initially worked on the old platform, it was likely that only the one “sacrificed” person would work on manually testing the JBoss version. We would have more time to test the JBoss version before sending it to market, but it was unlikely that we would actually use that extra time to manually test JBoss.

If we chose to have a unified code base, we needed to work on infrastructure that would support parallel development, and it was possible to have “JBoss induced” bugs in the version shipped to the market. The advantages of this approach were: simpler code base management, developers seeing code related to both JBoss and old platform when making changes (regardless of whether they were working on the JBoss or old version) and the whole team getting an early look at the code that one day would run JBoss.

With these considerations in mind, we decided to use a single code version as opposed to the separate branch in the source code control system for JBoss code. We felt that quality was the result of good teams and good process, and those wouldn’t crumble if we added a few more moving parts, as long as these parts were well made. The added benefit was that we minimized the “Big Bang” event. At some point, we needed to move the whole team on the final JBoss release. If we chose separate branches, the “Big Bang” would be the first time our JBoss code was used by the whole team. As a result, we built a build system that was capable of creating both versions from the single code base.

We preferred the use of conditional logic (if/else in code) for addressing application server specific considerations over refactoring code to use of Proxy pattern [6] or dependency injection [17]. We felt that the amount of code specific to the app server in an average file was small enough that refactoring would not be worthwhile compared to conditional logic and applied “Do the Simplest Thing that Could Possibly Work” principle [7]. In situations where conditional logic was not an option (e.g. imports specific to app server), we used a preprocessor. For Java, that was a non-obvious choice as the core language doesn’t support it and there are disadvantages of preprocessor if overused [5]. We were very careful to limit its use, and we removed the preprocessor when the transition was done.

3.3. Initial Demo

Initially, we had to deal with a “chicken and egg” problem, since it was not possible to compile the system on JBoss due to dependencies on the old application server. We could remove all such dependencies, which required a lot of throwaway work to make an initial demo of the JBoss application. Files we modified for the “quick and dirty JBoss demo” would be changed later for the complete JBoss transition.

In situations like this, the best approach for an initial project phase is usually the simplest [7]. In our case, the simplest approach was to compile the system using the old libraries but run with JBoss libraries only. As a result, if a feature wasn’t using code specific to the old platform, it would still work. That approach allowed initial JBoss demos with a minimal amount of work.

As the project progressed, we removed dependencies on the old library and removed the old library from the compilation classpath. If we forgot to remove a dependency, removing the old library from the compilation path would show us. – that was our “fail fast” [15] safeguard for forgetting to remove “demo hacks” from final code.

3.4. Sacrifice One

During the entire race, our transition was a moving target, since new features were always being added to the application and released on the old platform. Even though we discussed all changes in advance to make sure that there weren’t any obvious problems that a feature would induce on JBoss, constant effort was still needed to reconcile these changes with the JBoss infrastructure. As small details are often not detected until tests failed. The “sacrificed” person would look at the daily build reports and automated test results and immediately address failed builds and broken tests in the JBoss version. Sometimes it would be a simple problem to fix, but often it would be beneficial to consult team members responsible for changes that broke the tests, as they were more familiar with the context of the code that they worked on yesterday.
Not surprisingly, “fail fast” [15] is the cheapest way to fix defects because when people remembered the context of their problems, they were better in fixing them. As long as we followed the “Sacrifice One” pattern, the effort needed to prevent code degradation was fairly small. For example, one developer could keep up with five people on the branch with only 20% of his time devoted to integration.

We had one situation in which the whole team was assigned to the new feature release for over a month due to business priorities, and no one worked on JBoss. We found that the effort needed for reintegration after that break was on average four times more per test case that needed to be fixed. The primary cause was that if the test was broken yesterday, most of the time the developer who is responsible would be able to immediately point to what might be a problem. If the test was broken a month ago, we needed to pass through the full debug cycle.

3.5. Technical Risks

The best approach in any agile project that is technically driven (like refactoring or changing app server) is to tackle the “biggest addressable risk first.” The reason for this approach is that if you are unlucky and a technical risk is “terminal” for the project, the least money is spent for the company if the project is killed early. If the technical risk is not “terminal,” you get the most time possible to address the problem while the team working on it is small (e.g. one person).

With that in mind, we had a specific goal to schedule any big risks as early as possible. To identify them, we used a combination of the brainstorming analysis (similar to vulnerability analysis done in STRIDE security model [16]), and early prototyping of the complex technical issues. To score the “biggest risk,” we used the Delphi agreement [3] of developers to decide how serious the risks were.

As an example, JBoss integration with TopLink was required but risky. We mitigated this risk by early prototyping of the integration.

3.6. Automated Testing

We used both in-container and unit tests [9] for the testing. Unit tests were beneficial, but we found that in-container tests were more useful for diagnostics of JBoss specific transition problems. In-container tests exercised JBoss server integration logic, which is where many bugs were.

3.7. Whole Team on JBoss

Although work was done to minimize “Big Bang” events, when we switched the whole team onto the JBoss release, it was still a big event. It was necessary to ensure that the code was stable before we moved to JBoss. Otherwise, significant time and effort would be spent on tracking “why doesn’t this work on JBoss” instead of “why doesn’t this work at all.”

The team did a great job on transitioning to JBoss, and we had a stable transition. Still, it was important that the team member participating in the JBoss transition was available during the first days after the transition for consultation and help in diagnosing problems.

We learned that when you use the “Sacrifice One” pattern, the “sacrificed” team member should use all the tools that the team is using. In our case, the “sacrificed” member was using an Eclipse IDE, while most of the team used IntelliJ. As one would expect, there were IntelliJ-specific issues during transition.

Another lesson learned was that presentations are not a substitute for practical experience. Although there were multiple presentations explaining how the new build system worked, the team transitioning to JBoss uncovered awkward parts of the workflow with respect to JSP development that were less obvious during presentations. It is possible that this issue could have been avoided if we rotated “sacrificed ones”. Fortunately, workflow issues were limited on needing to provide “in-place” JSP edit capability and were small enough that we were able to address them with minimal work.

3.8. Estimation

We constantly struggled with ballooning estimation of the remaining work. Initially, the automated solution to identifying all dependencies was to compile on JBoss and see what broke. This didn’t work. The list was huge and required a human to sort it, and we still had issues with JSP pages, configuration file modifications, and run time dependencies. We decided to instead depend on meeting with team members to discover and estimate high-level dependencies.

Unfortunately, that didn’t solve the problem either. In retrospect, our main problem was forgetting whole areas that needed to be done - we missed some dependencies on the old application server in every single estimation meeting and as a result we constantly underestimated remaining work.

We tried many approaches to improve the accuracy of our estimation. Basic planning meetings in which
team members enumerated remaining tasks were supplemented with presenting architectural diagrams on a whiteboard and analyzing dependencies as a team. Questionnaires were handed to team members asking for remaining work to be enumerated in private and results then cross-correlated (this approach was inspired by Delphi agreement [3]). Finally, we extrapolated underestimation trends from the previous estimation sessions to judge how much remaining work was missed.

The end result was that as much as we tried, we were still significantly underestimating total work remaining. However, a week after the whole team was on the project, we had estimation that was within 10% of the final work achieved, mostly through informal communication of discovered issues.

This result was certainly biased by the fact that each successive estimate was done on smaller amounts of remaining work. Even with that in mind, based on the dramatic improvement we saw, we still considered it to be more effective than any other estimation technique tried. This is especially true in situations where the real problem is not as much quality of estimate as it is remembering all the work that needs to be done.

3.9. Fallback Positions Must Be Safe

When the whole team was transferred to JBoss, we were reasonably confident that we would deliver the next release on JBoss, but we still wanted to have a contingency plan. We had options to develop all features in parallel on JBoss and the old system, or to “freeze” the old version and use the “frozen” version as fallback.

The team decision was that we would move the whole team on JBoss and have the “frozen” version as a fallback, so there would be no throwaway work in the case of success. We planned, and were able to deliver, success and had a failure mitigation strategy that required minimum work.

6. What We Would Do Again

The “biggest risk first” policy was a plus – the project had practically no technical issues that were considered showstoppers once the whole team was on it. We knew we had a lot of work remaining, but we didn’t have issues that could kill us unless we found some non-obvious solution soon.

In-container tests were very helpful in capturing misconfigurations of the JBoss environment.

We delivered three releases in a space of nine months, had biweekly demos and improved automated testing coverage during this project.

Working from a single code base for both new and old architectures was a very good decision. Engineers working on the old architecture were working on the same code base as the ones working on JBoss, and all were aware of the dependencies to the other app server by just looking at the code that had JBoss conditional logic. This could be looked at as a form of osmotic communication [14] through code. We were lucky enough that not a single bug made it to the customers as a direct result of that decision.

Most importantly, the project was a success. Quality of the release was good and in most of the metrics identical with the quality of any of our previous releases. This validated the approach we took with respect to architectural decisions, in particular interleaving architectural decisions with development, quick prototyping of architecture and proactive anticipation of issues.

7. What We Would Do Better

Not all went smoothly. We were constantly underestimating the amount of work remaining. It is difficult to estimate work remaining on something you haven’t started. Working for a short period of time and then estimating work remaining is much more effective. If we wanted better estimation, we should have put the whole team to work on JBoss for a day or two earlier in the project.

Is “Sacrifice One” a better approach than “Team per Task”? In our case, we felt that initially we were willing to risk concentrating most of the knowledge with a single person and didn’t mandate pair programming. Anyone who feels differently should use the “Team per Task” approach. Also, the “sacrificed” team member should use all the tools the rest of the team is using (e.g. if multiple IDEs are used in the team, they need to be covered by the sacrificed people).

Presentations are not a substitute for practical experience – people discover problems more easily when they actually work on something. Rotating people so that different people are the “sacrificed one” at different times should be considered if is feasible.

Due to the “next release needs all the help it can get” pressure, we were not fully committed to “Sacrifice One” all the time. The result was significant time spent in tracking issues affecting JBoss for which context was totally forgotten weeks earlier. It could be argued that it is very difficult for any organization to look past the next release, but due to the higher frequency of the releases, agile organizations are in a different position than organizations that ship their software e.g. once per year. As agile organizations feel
the results of the “next release” over-focus earlier, we believe in concentrating on the question, “what is impact of the decision for the next six months?” (as opposed to the “next release is king” approach).

8. Summary

We presented how Rally succeeded in delivering a long-running architectural project without sacrificing the agility of its environment. The key elements of the solution were the “Sacrifice One” approach at the beginning, addressing the biggest technical risks first, and frequent demonstrations of progress on the JBoss line. We anticipated problems but we chose the simplest solution to address them and were careful to always have a fallback position. Architectural decisions were made “just in time” and were backed by prototyping.

We finished with a very successful architectural change that provided all the benefits we expected from it, while continuously releasing new features in an agile way. This demonstrates that it is possible to use agile approaches even with architectural changes that can’t be released incrementally.

9. Acknowledgments

I would first like to thank the Rally team, without which all of this wouldn’t be possible. Thanks to Alex Pukinskis, Sam Newman and Anne Greenhaw for many great suggestions. Finally, thanks to Rally executive management for being supportive.

10. References


