Agile Architecture IS Possible – You First Have to Believe!

Mark Isham  
ChannelAdvisor (www.ChannelAdvisor.Com)  
Mark.Isham@ChannelAdvisor.com

Abstract

In early 2007, ChannelAdvisor undertook an ambitious project to rearchitect one of its core product offerings. Flush with early success from rapid customer adoption, the foundation for this product had grown well beyond its initial design. As defects grew in frequency and severity, pressure mounted to “fix it yesterday”, but consensus on exactly how proved elusive. “Fix it right” ideals battled with “fix it now”, ultimately resulting in a new architecture that was inefficient, incomplete, and, after 3 long months of “working harder, not smarter”, cancelled.

In this experience report I will recap my experience as a product development manager sponsoring this project; revealing how the project went awry, what the team learned, and how the utilization of the Scrum process not only created a scalable, reliable architecture, but also greatly improved the ongoing productivity and morale of the product team.

1. The Problem

Introduced about 2 years prior to this project, ChannelAdvisor offered a new product that at the time had undergone a massively successful launch with over 300% yearly growth in both number of customers as well as data footprint. The architecture that supported this product resided on 2 core database servers:

- The first server was a “transaction” server dedicated to processing millions of near-real time data transactions originating from external eCommerce partners every day. This server was online and processing transactions 24/7.
- The second server was a “reporting” server dedicated to aggregating the raw transactional data from the first server into numerous composite tables specifically optimized for query performance and utilized for the product’s reporting and data mining needs. Additionally, the data on this server provided the inputs to a large scale automated system that controlled both pricing and cost amounts for millions of eCommerce transactions every day. This server commonly managed hundreds of millions of records and was also functioning in a 24/7 environment.

In addition to these 2 core database servers, several home-grown processes existed to synchronize the data between those 2 systems. As more and more customers were added to the platform, it was not uncommon for this synchronization process to take 12 hours or more every day. Additionally, as load increased so did the occurrence of system timeouts, data inconsistencies (due to synchronization “race conditions”), and numerous other “at scale bugs” that would result in missing or duplicated data in the reporting tables.

From the customer’s point of view, this meant:

- Inconsistent reports: report A would show one data value, while report B would show that same data with a different value.
- Misleading reports: report totals reported higher or lower then what really happened.
- Incorrect pricing and cost amounts on customer’s eCommerce transactions, resulting in overspending and/or underselling.

This legacy architecture also suffered from reliance on an inherently expensive “Scale Up” design – as more and more customers were to be added to the platform, more memory, CPUs, and storage space was required to handle the increasing load. As the primary servers would need to become bigger and bigger, the hosting costs would in turn would grow exponentially larger.

2. The Response

As the bugs grew in severity and frequency, so did the cost: customers were losing money and faith in the
product. External and internal pressures mounted – something needed to be done **yesterday**!

The response was as you may expect: round up the smartest people in the room, throw in some pizzas, and lock the door until a solution was found. Clever ideas rolled out left and right, but quick consensus on approach was not easily achieved between 2 primary schools of thought: “fix it right” versus “fix it now.”

2.1 “Fix It Right”

“This system was not built for this load, if we do not redesign the core architecture once and for all, we will hit these same problems again and again!”

This was a common argument from the senior architects and engineers on the project. A true enough statement to be sure, but came attached with a large price tag. The proposed solution:

- Combine all tables on the “transaction” and “reporting” servers into 1 master database server, eliminating the need to synchronize data, and thus reducing the cause of many of the data inconsistencies.
- Duplicate this consolidated architecture in whole on multiple servers, and update all client logic to partition all queries based on customer ID. Customers are assigned a specific server, and as you add more customers, you add more servers. (“scale out”).
- Partition large tables further by date, and limit queries to be directed to specific date tables. This in turn will optimize query performance and reduce timeout failures.

The design approach was deemed sound, but the plan of attack proved its undoing:

- 3-4 weeks needed to prototype this approach to make sure it would indeed solve the problems.
- 3-6 months implementation time by 3 “core engineers”, to be done in parallel with ongoing feature work for the product
- Perform massive code merge at end of project, as it was “too hard” to merge these large changes any earlier.
- All existing data had to be transformed into the new architecture via lengthy “reprocessing” task that would take days or weeks to complete due to the sheer volume of data.
- When complete, historical reports may produce different results since new architecture “fixed the bugs” that caused incorrect numbers in the past.

The net result was a proposal for a 3-6 month project fraught with risk and no easy rollback. Although the end result was attractive (**IF** delivered as promised), the business could not afford to wait so long for relief, something had to be done sooner.

2.2 “Fix It Now”

“Why do we have to rebuild the whole system? Let’s just fix the bugs!”

This was the common response from the business and customer representatives, but was this approach enough? The proposed solution:

- We don’t have time for a full rearchitecture, not everything can be bad. Let’s instead just focus on the top bugs and apply fixes to the existing architecture.
- Once the major bugs are fixed, we will revisit the “fix it right” plan.

This counterproposal received solid support, especially given the cheaper price tag (results in days instead of months), and ultimately was pursued with some short term success. Within 2-3 weeks, though, it was soon discovered that this approach would not be good enough. Efforts had degenerated into a “Whack-A-Mole” bug hunt (fix one bug, create another) and customer problems did not seriously abate after the initial flurry of success.

“Fix It Right” again got its day in the sun, but with some bounds…

3. The Failed Project

Three months was simply too long to wait for a fix, but “just in time” bug fixes weren’t helping either. As a result, the architect team was given the green light to pursue their proposed design with some controls in place, namely:

- The product could not survive long enough to wait for the full solution, the team must deliver incremental improvements every 6 weeks or less.
- The team could have access to whatever people were deemed critical to the project, but this effort could not come at the expense of no competitive feature movement for 3 months. (“all will be for nothing if we can’t sell this product.”).
• Priority focus would be on data integrity issues first, scalability concerns second, and performance impediments third.

The project started with enthusiasm and optimism (“finally we are getting to build this the way it should have been built!”), but soon turned down a dark path. A lack of project management discipline coupled with “crisis fatigue” within the organization led to a mentality of “don’t concern me with the details, just fix it.” The architects were left alone to make things right, with little accountability or direction for the path they were taking.

For the first 6 week iteration scope initially was simple: reduce the possibility of data duplication by reducing the number of aggregated data tables. As time marched on scope creep set in (“while we’re fixing this, let’s also fix that”), purely done at the engineer level without business or project oversight. The project was construed as “an architecture project” and the architects were “handling it.”

As days turned into weeks, 6 weeks turned into 8, and then again to 10. Consistently we were “just 1 week away” from being done, but the train never left the tunnel. Morale plummeted and 80 hour work weeks settled in. Team members were added, but this only slowed things down further (a phenomenon well explained in the classic computer science book, The Mythical Man Month, by Fred Brooks).

After 3 months with no end in sight, the plug was pulled. Time to pause, reset, and rethink the approach. Three months down the drain, and the original problems were only getting worse.

4. What Went Wrong

Numerous contributing factors led to the project demise, but arguably they all traced back to poor project management. The urgency of problems coupled with crisis fatigue resulted in a relatively unchallenged blank check for the architects to “fix the problem.” When delays set in, the technical details were too obscure to be challenged, a leap of faith was made that the architecture team was doing all it could.

To be fair, the team was admirably working hard to “do the right thing” for the product, but classic “waterfall” training led to cultural biases that favored monolithic redesign over incremental value wins – without appropriate guidance and mentoring in place, there was simply an inability to understand how to deliver such a large project in smaller incremental wins.

Among the mistakes made:

• The original project mandate of 6 week iterations, although well intentioned and agile inspired, was not accompanied by the appropriate project management needed to make it happen. A 6 week iteration milestone soon became a 6 week deadline to complete a 3 month project.

• The highly technical nature of the solution led to a lack of accountability to the architects. The details of the project were difficult to understand to the layman, thus scope increased unchallenged and time delays were accepted simply because “we have to do this.”

• Time pressure caused the team to skip crucial research and prototype steps, resulting in solutions that were not identified as failures until weeks later.

• Scope creep pervaded – while fixing one thing, why not fix another? Estimates were not tracked and customer prioritization was not considered. As deadlines neared and the original goals remained incomplete, working hours increased while morale decreased. Too much time had been spent “working on the wrong things” and the team could never quite catch up.

• Adequate communication did not occur between the architecture team and the product team working on features in parallel. Features were being developed against an architecture that was radically changing, causing developers to shoot at a moving target. Additionally, the architecture team did not always fully understand the repercussions their design changes would have on existing and new product features.

5. Enter Scrum

Flush with humility from a failed project, the team was receptive to a new approach. Scrum had already been utilized by other ChannelAdvisor projects for several months prior. It was now time to ignore calls that “scrum can’t work for architecture” – it was time to give Scrum a chance!

As part of the introduction of Scrum to this project, numerous changes were made:

• A business oriented Product Owner was assigned to the project, providing clear guidance and accountability to customer needs.

• The Product and Architecture teams were re-integrated, both holding a stake in the success
of the project. Coding was done in a common branch in the spirit of “continuous integration.”

- Sprint iterations were shortened to 1 month or less. Backlog scope was clearly identified prior to the sprint and team was held accountable to working with the product owner to stay focused on the backlog priorities.
- Research “spikes” and prototypes were allowed and accounted for with time-boxed restrictions. Time was made to “figure out next steps”, but not for open ended research walkabouts. An initial vision and roadmap was put in place prior to diving into the first sprint.

The addition of the loose structure of Scrum proved to be just the right amount of project management. Within the span of 3 months (the same timeframe as the first “failed” project), the following accomplishments were achieved:

1. Sprint 1 (first month): Prototyped a “consolidated single server” architecture where functional pieces of the application that originally were distributed across the 2 core database servers were now to be consolidated in one place on a single server, which in turn could be cloned to copies used for different sets of customer data. In other words, where all customers would access certain features on certain servers before, now some customers would access all features on certain servers instead. This approach would solve 2 core problems:
   a. Eliminates the data synchronization problems across servers since a single customer would not need to access data on any server other then their own.
   b. Provides a linear “scale out” solution where more servers could be added at incremental cost to accommodate additional customers by simply cloning the configuration of that first server again and again.

2. Sprint 2 (second month): Implemented consolidated server technology researched in phase one, and further reduced number of aggregated data tables (tables that include copies of data from other tables) , in turn reducing the possibility of data integrity problems since there are less places for the data to get out of synch.

3. Sprint 3 (third month): Prototyped approach to improve performance by partitioning tables at date level. Further reduced number of composite tables to reduce data integrity problems further.

4. And onward…

Within the same period of time of time, the data integrity bugs reduced greatly, the product was much better prepared to scale out, and morale on the team increased significantly. The Product Owner kept priorities grounded in business realities, but the architects were also given their chance to set things right. Scope as well managed to facilitate “work smarter not harder” – the death marches were over!

6. One Year Later

Scrum was the secret sauce that allowed this product to break through its glass ceiling. The architecture is now reliable and scalable, and the team has never been more productive. Among the accomplishments in the last year:

- Data integrity problems have been all but eliminated through a simpler, more elegant design. The number of composite tables has been reduced from 16 to 4, and there is no more need for the failure-prone synchronization process that had caused so many of the failures in the past.
- The product is now truly a “scale out” solution, well partitioned across 4 servers and ready to handle ever increasing customer load simply by purchasing and plugging in more low cost servers.
- The product has more then doubled the number of active customers, with a corresponding doubling of data footprint. Over 180 million records are processed daily representing well over a terabyte of active data.
- The scrum team has held over 12 successful sprints since this project, releasing hundreds of enhancements and fixes over consistent monthly releases.
- Morale is much higher, and team is working sustainable 40(ish) hour weeks by “working smarter.”

7. Lessons Learned

Although a failed project is never desirable, if the lessons learned are heartfelt and a true “buy in” to improvement occurs, great opportunity can be built from the experience. Such was the case with this
Among the lessons learned:

7.1 Make the Time for Research

A major rearchitecture should not be rushed, but it should also not run unbounded. Spend the time to put a vision and roadmap in place at the start of your project, but expect change and plan for course corrections along the way. Do not allow for weeks of unbounded research, as even the best laid designs will need adjustment as time passes and the business environment changes.

In general we like to use a series of chained research “spikes” (the results of each drive the requirements for the next), each 2 days or less with clearly defined acceptance criteria. Acceptance criteria may include a flushed out high level design, a list of further research topics, or other tangible and reviewable artifacts. An output of a research tasks should never be as vague as “Yep it’ll work for us” – get specifics and hold the team accountable to next steps based on those specifics.

Additionally, if an engineer says something will take a week to research, they should be challenged to break those tasks into smaller components. Avoid chasing rabbit holes, leverage the product owner’s perspective to keep the team on priority focus as it matters to the customer.

7.2 Avoid The Monolithic Project

Confronted with the large scope of changes needed to be accomplished, the architecture team had originally presented the following arguments:

1. This will take us at least 6 months to deliver.
2. Architecture just can’t be done incrementally.
3. It will take longer to do in “parts” than all at once.

Experience on this project had proven these arguments to be naive, and in some cases flat out incorrect. Let’s take a look at each in turn:

7.2.1 “This will take us at least 6 months to deliver.”

A statement like this is sure to create sticker shock in any business sponsor. If you are facing problems today, you often can’t wait 6 months to fix them — you need something quicker. Where this project went wrong, though, was a misunderstanding that “quicker” meant “implement the same scope in less time.” Such an approach is unrealistic, fraught with risk and is not sustainable in the long run. Instead, the team learned to break a large task into incremental deliverables, and provide steady “production ready” value to the business along the way.

In time, the team learned the value in avoiding the “all of your problems will be fixed in 6 months” trap and instead focusing on “we will improve the highest priority issue in 4 weeks, revisit the next priority and improve that 4 weeks later, and repeat…”

7.2.2 “Architecture just can’t be done incrementally.”

Assuming the level of granularity mentioned is not considered to be trivial, I assert this statement can almost always be refuted. The misconception of the team on this project was that all changes had to be completely finalized and ready for mass consumption at release. A focus on how your customers truly work can help provide guidance on what really is needed when. Here are some strategies we commonly employed to provide incremental architectural wins:

- Roll out the new architecture “offline” and in parallel to the legacy architecture. Beta test select customers on the new architecture or side by side with the existing architecture. If things go wrong, pull the plug on the new architecture and revert back to the old.
- Given an idea on where you want the ultimate design to be, it is not always necessary that all components are refactored at the same time. Divide and conquer constituent components of the design in iterations, using temporary scaffolding when appropriate (proxy classes, interface adapters, etc.) to link the old components to the new architecture. When all old systems have been converted, destroy the old scaffolding and move on with the new.
- Leave it alone. Often the temptation to redesign an old system is due less to inherent design flaws then it is a lack of familiarity with how the old code works. You wouldn’t rebuild your house because you didn’t like the original color of the paint. Always keep the business perspective in mind when you determine what should and should not be refactored. “Joel On Software” Spolsky wrote what I consider the classic text on this point in his blog “Things You Should Never Do, Part I [Spolsky2000].”

7.2.3 “It will take longer to do in “parts” than all at once.”
This is only true if you do it 100% right on the first time, but how many of us can claim to do that? The value of incremental deliveries is when you can learn and adjust along the way. The sum of the individual parts may be greater if you are priced for perfection, but as weeks go into months it is inevitable that some factor will change, internally or externally, that will cause you to have to rethink your design to some degree. Incremental thinking gives you the ability to checkpoint and adjust along the way, rather then committing to a path that will not truly solve the problem when you finally get there.

7.3 Don’t Work in a Vacuum!

The architecture team worked independently from the product team to minimize disruptions to the feature work already underway. In the end this proved to be disastrous as the changing designs were too radically different to easily be merged back into the functional code branch without a major collaboration effort. In the end it was learned that the entire team needed to be involved in the communication and decision process, and constant merging through Continuous Integration reduced the complexity and risk considerably.

8. In Closing

This rearchitecture project certainly encountered its challenges and was a valuable lesson in learning that solid project management was just as important as a clever architectural design. More importantly, it proved to a skeptical audience that the old ways of “rebuild from the ground up” have become outdated and inadequate in the modern Internet business climate. Refactoring a large scale architecture is not only possible using agile methods, in this project it proved instrumental to its success! Don’t let the architects sell you on a full redesign, Agile Architecture IS possible – you first have to believe!

9. References

http://en.wikipedia.org/wiki/Whack_a_mole

http://en.wikipedia.org/wiki/The_Mythical_Man-Month


http://www.joelonsoftware.com/articles/fog000000069.html