The Pragmatic Programmers

Real-World Kanban

Second Edition

Do Less, Accomplish More with Lean Thinking

Mattias Skarin
Foreword by Henrik Kniberg
edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 4

Using Kanban to Save a Derailing Project

The first step in solving a problem is seeing that it exists. That sounds simple enough, except in situations where everyone wants to solve a different problem. Everyone involved knows there is something wrong, but each person sees different things. It's really difficult to figure out how to fix the problem if no one agrees what the problem is in the first place.

Throw in other challenges, such as a looming deadline and lack of coordination across teams, and the problem seems unsurmountable. Let's see how a company used Kanban to identify—and agree on!—the problems, prioritize the issues, and come up with a plan to address them.

The Challenge: Restoring Trust by Solving the Right Problem

In situations like these, you'd be well advised to equip yourself with proper protection!

Company F, an open-source platform provider, was six months into an eight-month project when the client threatened to pull the plug because of a series of problems. This would've meant saying goodbye to a key client who was considered important to the company's operations. What had initially run well—a pilot delivered on time with a happy client—had turned into a growing pain of overtime, rework, and technical debt. After several scope adjustments, the final product to be delivered was the bare minimum the client needed. There was nothing left to trim, and it looked like the team was still going to miss the delivery date.

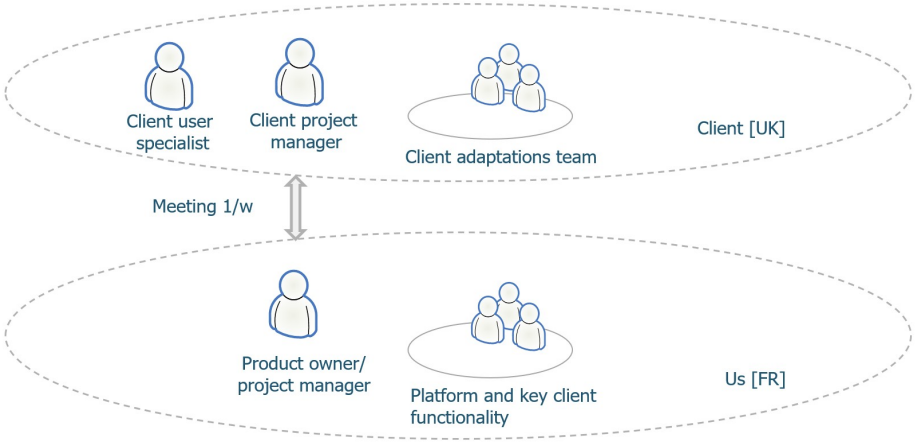
On a positive note—and there is always something positive—the development team consisted of very committed members and a positive project manager/product owner. The team had strong backing from management and weekly conversations with the client. The dialogue wasn't always pleasant, but at least both sides were talking regularly.

We discovered the first problem right away. When I asked each person to identify the biggest problem, everyone had a different answer. While everyone had his or her own view of the situation, there was no common view or shared perspective.

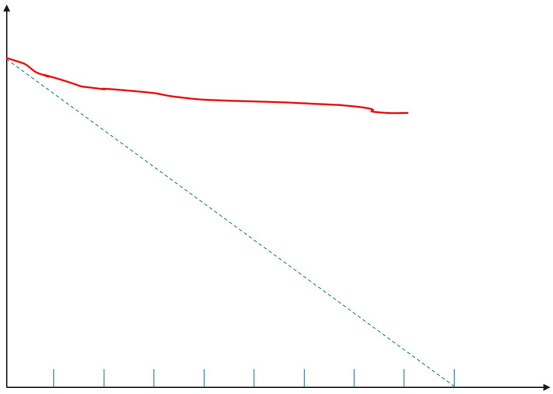
The project manager saw the team struggling to deliver quality products, getting overstressed, and consistently underestimating the level of effort and time required. For new tasks, the estimates were as much as five times off. The development team felt there was too much switching between tasks and the members were doing too many things at the same time. The client felt the team was in way over its head and was not up to the task of completing the project.

As you can see, the answers varied greatly depending on who you asked. Interviewing everyone involved didn't help us figure out what to address first, so we looked at how the team was working together.

Two development teams were working on this project. The foundation and the main bulk of the work was completed by Company F's team in France, while the client made its adaptations through a team in the United Kingdom. The two project managers met once a week to revise priorities and to review progress.



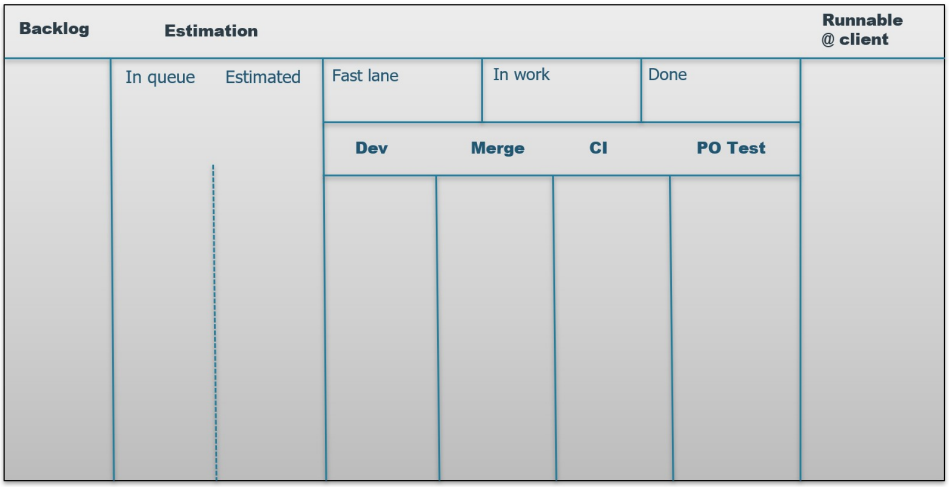
Company F's team had been using Scrum for a couple of months, and the project manager was a certified Scrum master. The team didn't think Scrum was working. There were several signs of trouble, such as the fact that the team was working massive amounts of overtime, often until one in the morning. You can see the worrisome pattern in the sprint burndown:



My first thought when I saw the burndown was, "Why didn't this trigger a reaction?" Let's look at how we regained control of the team's time.

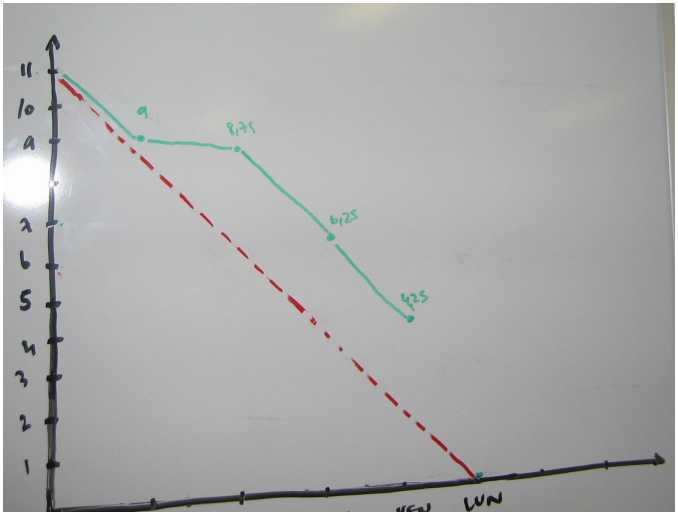
How We Got Started

The decision to try Kanban came from Company F's CEO. The team didn't know about Kanban until the first board went up on the wall. The board listed all work-in-progress items under the Dev, Merge, CI, and PO Test columns. Urgent items went into the Fast Lane at the top. We allowed only one item in the Fast Lane at a time.



We didn't change anything else in the team's work procedure or process. The team continued to run two-week sprints, but with this board. The main difference between the new Kanban board and the previous Scrum board the team used was the fact that there was now a limit on work in progress. We talked about WIP limits, which each team defines for itself, in *Leading with Kanban*. The new board also made the value stream more prominent.

As you can see in the sprint burndown chart, after two weeks little things began to change. There was an actual burndown, as opposed to the flatline we saw on previous charts.



Kanban brought a shared overview of flow and a work-in-progress (WIP) limit to the team, and the team could see positive results after just two weeks. The

team worked on fewer stories at a time and was able to both develop and test the features. We saw the team could produce working software if the conditions were right.

Develop a Shared View of Progress

We realized the team and the project manager were tracking two different things. The project manager was tracking what activities were completed and checking the project plan. The team tracked the progress of the current sprint. Because the client, the project manager, and the team were looking at different things, they could not view the project's overall progress in the same way. Kanban provided a way to see all the steps needed to move from the customer request to executable software. Visualizing the process also helped flag problem areas.

Before Kanban, producing this overview would have required extensive discussions with multiple people and compiling data from different tools.

Getting a Shared View on Progress

There are many ways to develop a shared view of the team's progress.

It doesn't matter which indicator is being used, as long as it is meaningful to the people involved.

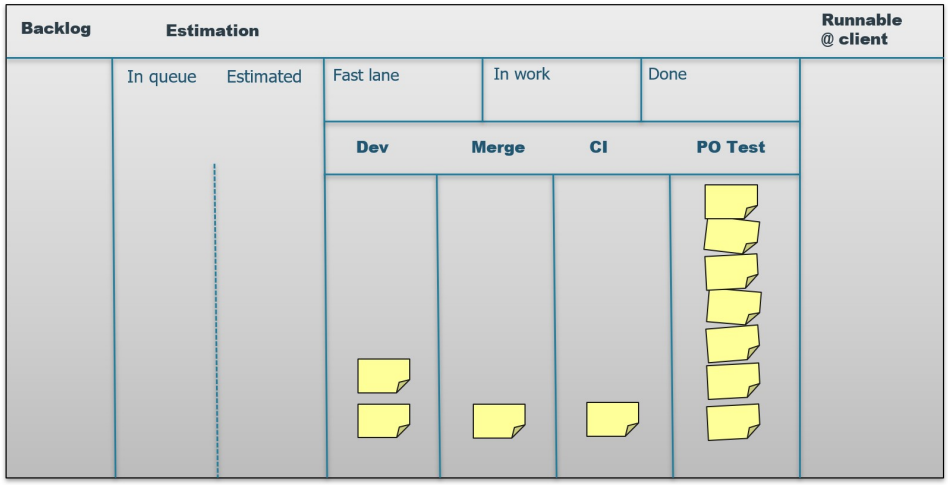
Here's a tip: If the indicator being used to report progress is not being used to make decisions, pick something else that's more useful.

If you don't have a progress indicator you trust, take a quick poll of the project participants' confidence level about making a deadline (on a scale of 1 to 5, with 5 being very confident). Yes, it's a crude measure, but it's better than nothing and better than using a bad indicator. If you include a cross-section of the people involved in this poll, the results actually reflect the team's current best knowledge.

Or, you can use a simple release burndown that's visible to everyone. Make sure everyone agrees that *done* in this context means *tested*. Have testers handle progress reporting to keep this honest.

Figure Out What to Fix

Two weeks in, we uncovered more interesting patterns on the Kanban board. Stories frequently got stuck in the testing phase. Other stories on the board would be removed after the sprint even though they were incomplete. Even more worrying, those stories resurfaced on the board a few sprints later.



The PO Test column showed our first bottleneck. The team was quick with development but struggled with testing and quality assurance. It turned out the project manager handled acceptance testing, but she could spend only half of her time testing. At the end of the sprint, many of these features shipped even though testing wasn't complete.

A part-time person testing code from four developers—no wonder we had a bottleneck!

We found that some of the stories were far too complex to complete coding and testing within a single two-week sprint. Because they were too complicated, these half-completed stories were removed from the board. However, because they were crucial to the overall project's success, they would come back on the board. The rework was affecting team morale.

Releases were also taking too long. The team would learn about a detail that needed to be tweaked while preparing the Monday release. The release date would shift a day or two to accommodate the change, requiring the team to redo some of the prep work. We found the team was spending two to three days on release work instead of just one day.

Make Changes in Workflow

When you want to change a project stuck in a downward spiral, you need to take a stand. If it works, trust will build, and you will gain a little bit of slack, which you can use toward long-term improvements. You will slowly work your way back up instead of going down.

Making a stand for quality is always a good bet. Our line in the sand was that we would include only stories of acceptable quality in each release, regardless of what was previously promised or what people were expecting. We also said release dates would always be on Mondays and would never shift. It wasn't easy convincing a nervous client, so we moved away from a two-week release cycle to a weekly one. That way, if we missed the expected release date, the client just had to wait one more week to get the desired feature.

With this decision, the team could focus. There was less emergency patching, less time spent on re-estimating tasks, and less time spent on repeated acceptance testing. Stories stayed on the board until they were complete, regardless of how many sprints they took. Instead of chasing things that didn't quite turn out as expected, the client focused on validating the quality of the releases. There was a faster learning cycle because the team was getting feedback once a week.

We saw a boost in the team's morale the first time it successfully shipped a story that was too big for one sprint.

Making the shift wasn't easy because it meant saying no to the client a few times when the feature wasn't good enough to be released. But we bet that shipping features that worked as expected would grow trust and put us on the positive upswing.

We had no shortage of ideas for things we wanted to improve, but an immediate one demanded our attention: we needed time to make improvements. This project was already late, and the team's schedule was jam-packed just to complete the essentials. We needed to find slack.

We found it in different places. One was in how the team approached estimates. Instead of spending a full day coming up with estimates for each story, we tried using the T-shirt sizing scheme (small, medium, large). The sizes corresponded to time: small meant one day or less, medium meant one week or less, and large was anything that would take longer than a week.

The first time we did this, we discussed the stories over lunch. By the time we got to coffee, each story had an updated estimate. By changing how we did estimates, we freed up time to work on other things.

Finding Spare Time During Periods of Stress

A trick to free up time is to look for planning efforts aimed at future work. The more distant the future, the higher the chance the work will be rendered useless by later changes and your preparations will have to be redone. So, if you have no slack time for improvements, scaling down time budgeted for future work is a good bet. It's what you deliver that counts, not what you're starting.

For any estimates, start by clarifying the question you need to answer first. Then proceed with the simplest possible estimate that gives you that answer.

A very simple and effective technique to estimate a bunch of stories in one shot is to gather your team and silently sort the stories on a table, from the highest complexity to the lowest. Then estimate the story with the highest complexity, the one in the middle, and the simplest story. Any other stories in between will get their estimates relative to these three anchor points.

In this particular case, the new estimates didn't have a purpose other than to keep the plan updated. We knew we were late—updating the plan wouldn't generate any new information or insights. I was tempted to use a guerrilla technique and just randomly pick estimates. We used the time we reclaimed with our T-shirt scheme to figure out where we needed the most improvements.

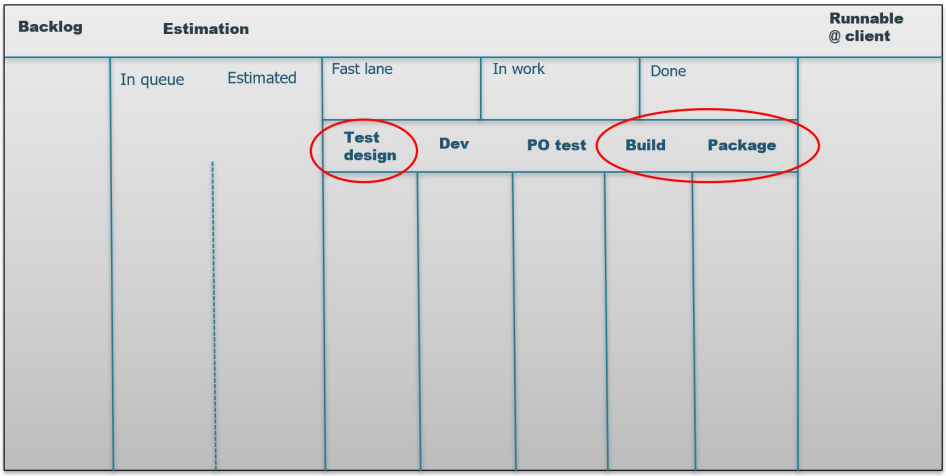
How Our Process Worked

We kept making new discoveries about how things really worked, what processes we actually wanted, and how we should communicate the changes to everyone involved. We were constantly in a state of flux for the better. We even found ourselves changing our processes at least once a week.

We wanted to improve our testing and address the imbalance between our development and testing throughput. We looked at test-driven development (TDD) to help us be smarter with manual testing, decrease regression testing overhead, and give us readable code. Our team members came in an hour early for four mornings to attend TDD training workshops. Because the goal of these sessions was to make the team feel confident that they could change any part of the codebase, two of the training sessions focused on using TDD techniques with legacy code.

The team refactored the code after the workshops. They had known for a while that the codebase needed it, but now they were fixing the issues as a team and not as individuals.

We updated the Kanban board with two new columns: Test Design and Build/Package. Test Design made the choice between manual or automatic testing explicit. The Build/Package columns were used to coordinate release changes with the client team in the UK.



We knew that TDD couldn't be used to develop every single part. For example, the code for our graphical user interface (GUI) didn't have a stable testing framework. Although we had some ideas on how to fix this, implementing the necessary infrastructure would be time consuming.

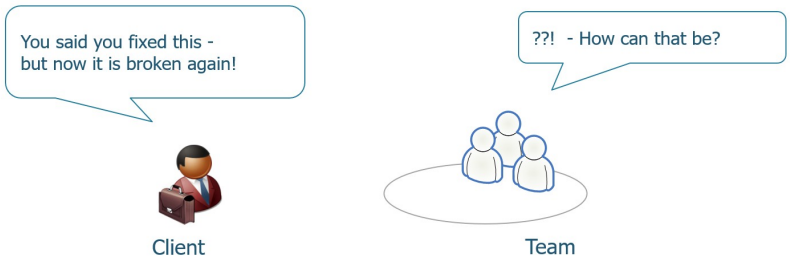
We compromised by focusing on baby steps. We designed for how we would test, and we used TDD wherever we could. We conducted manual testing where we couldn't write automated test cases. And finally, we committed to take one step forward each week toward implementing a working GUI test framework.

Even though the pressure was still intense, we saw small signs of improvements. We heard comments such as, "Why do we have things like this in our code?" and team members took the initiative to refactor code and fix quality issues. After a client meeting, the project manager said, "You know what? Even if they're still stressed out, they now trust me when I say we're going to deliver something."

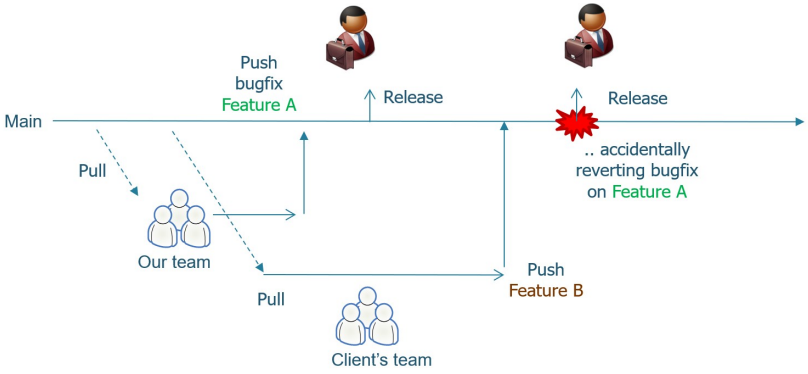
These small indicators showed that we were on the right track.

How We Addressed Problems

After a couple of working releases, we were feeling confident. Then, out of the blue, we got a call from our client that a feature we'd recently fixed had suddenly stopped working.



After some investigation, we realized our bug fix wasn't the problem. It was a regression error because a UK developer had inadvertently reintroduced the bug by committing old code.



It was clearly time to change how we worked with different teams. To prevent this situation from happening again, we created team branches and made it clear that committing code to the team branch meant it was ready for integration testing. We moved the test suites to each branch to validate the code before committing to the main branch. Code in the main branch meant it was ready for release. Finally, we made sure that each team branch was automatically updated with the latest release on a daily basis.

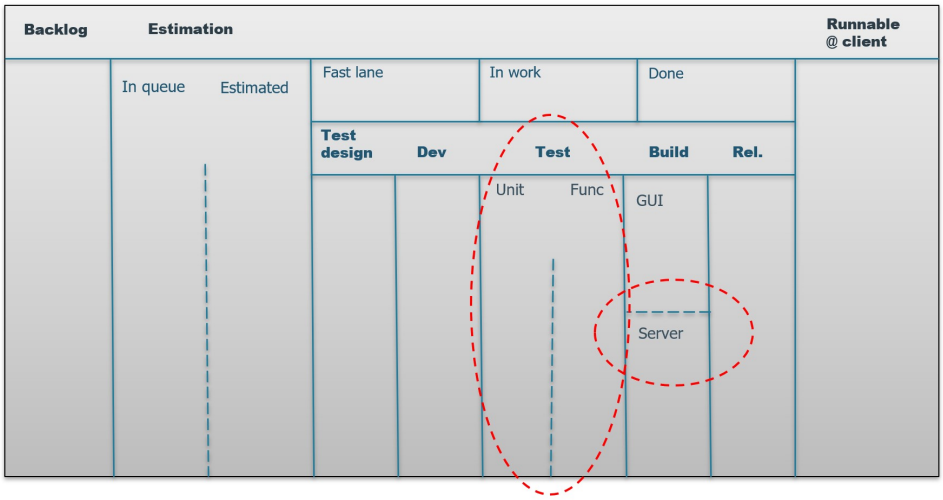
This is a case in point where you learn about how work is actually done under high pressure, as opposed to how it *should be* done as documented by some process or procedures manual.

To improve, we quickly realized that we had to get the other team on board with our changes. After all, we committed to the same codebase. The first step was to have the two development leads check in with each other on a weekly call. We also kicked off a developer exchange program where developers from other teams came and learned how we worked. We covered topics such as branching, TDD, design patterns, integrated development environment (IDE) setup, continuous integration, and testing environments. After all the developers went through the exchange program, we were confident we would be able to work together as one team to maintain a higher standard of quality.

The team, working together to maintain higher standards.

How We Continuously Improved

Everything so far was on the defense as we fought problems as they cropped up. After a few weeks, we saw team members kicking off their own refactoring initiatives and taking control of the board. Every change they made in their processes was reflected on the board. For example, the Test column was split into two to reflect manual testing. A triggering mechanism in the Build column indicated whether a partial (client) or a full build was needed. The Kanban board became a living process and galvanized the team's commitment for each change.



We don't always have all the information we need at once in software development. To get to the bottom of a complex problem, we need to involve people with different perspectives. Getting everyone into the same room and in front of a whiteboard is a very effective way to get to the crux of the problem quickly.

After a streak of successful releases, we ran into trouble with one. After some discussion, the CEO concluded, "I think this happened because the vision for the product wasn't clear." This could've turned into an argument. Instead, the team and the CEO gathered around a whiteboard and wrote down all the factors that could have caused the problems. It didn't take long for us to identify a set of factors that had been flying under the radar so far, the most pressing one being the lack of knowledge within the team to update an unstable third-party component vital to the product's use.

We decided to take ownership of the code, learn it, and refactor the unstable parts.

During times of stress, it's easy to squander your energy on the wrong things. If we had gone ahead with the CEO's initial thought to "fix the vision," we wouldn't have been able to improve in the few weeks we had left in the project. By doing a root cause analysis, we identified a tangible problem we could fix. We pulled different people together and identified more than one potential cause of the problem. The rest of the team and the CEO supported us once we decided what we had to do.

The Five Why's and When to Stop

A challenge to using the 5-Why technique (similar to our root cause diagram) is knowing when to stop. If the cause is outside your sphere of influence, then stop. Strive to do something small that improves things, even if it isn't the perfect approach. Over time, the small things add up.

How We Kept Focus During the Last Weeks

During the last couple of weeks of the project, the developers and managers worked closely together to fix any problems that could derail the final release. For example, the CEO made sure a senior developer was available to speed up fixes in core modules.

The team made the deadline. Although this had seemed like an unreachable goal just two months before, passing the finish line felt like any other day

instead of an extraordinary achievement. The fact that the client signed on for another project a week after release was the official acknowledgement of a job well done.

We all know that meeting a tight deadline can be done if you work a massive amount of overtime. But the team learned something else. Right after the final release, the project manager noted, "You know what? The team doesn't do overtime anymore." That was music to my ears. We had learned how to achieve more of the right things by doing less!

What Lessons We Learned

There was no one single thing that made this work, but a combination of small things reinforced each other. Kanban helped show the team where the problems existed (or didn't exist). How we fixed the problems was up to us.

The information was cheap. All we had to do was put a Kanban board on the wall and start using it. Nothing else had to change. We kept the ticket tracking system, the project structure, and the teams. As soon as we identified a problem, we fixed it. We didn't worry about trying to put in the best process as long as what we did was good enough.

Our implementation of Scrum was certainly less than perfect, but let's look at what we didn't do. We didn't go back and order more of the same medicine that we were already taking. We didn't mark ourselves against an ideal process method like "real Scrum" and start improvements from there. We couldn't have—we didn't have the trust capital to do such a thing, and we certainly didn't have the time to make random bets if we wanted to save the project.

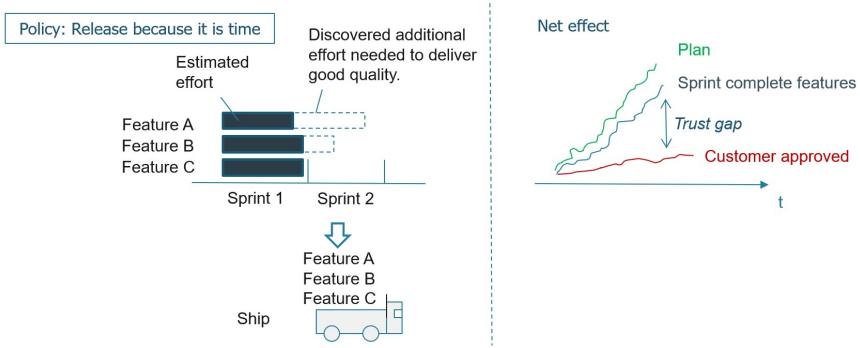
What did we do? We visualized where we had real flow problems and solved one problem at a time by focusing on quality. We wanted to get on that upward trajectory.

Why Our Scrum Implementation Didn't Work

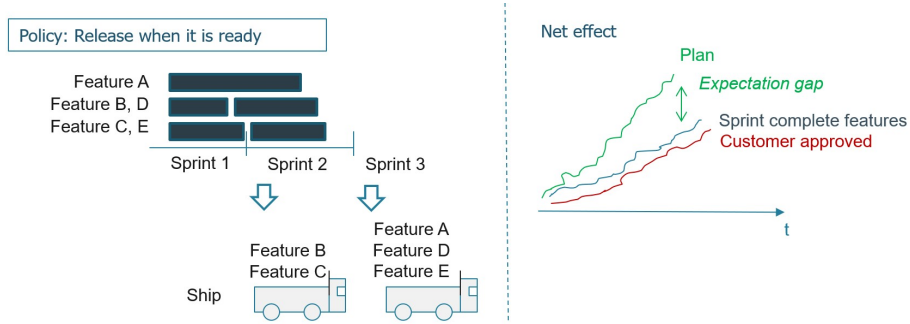
There are a few reasons why our Scrum implementation didn't work out, including rework and wasted effort, not refactoring the code, and the lack of cooperation between teams. Let's look at what we were doing wrong.

Features were designed to fit into sprints, but we didn't take into account that we might not be able to finish development during that timeframe. If we saw that we might have bitten off more than we could chew, we might have reduced scope so that we could at least finish parts of it, with plans to revisit the other parts at a later time. The next sprint focused on a different feature with a higher business value, and the cycle repeated. Because the partially

finished feature was released, the plan showed progress, but it didn't match what was actually delivered. There was an invisible but growing trust gap between all parties involved.



When we moved to Kanban, we committed to working on the feature until it was finished and released. When the features actually got released often deviated from the project plan, but because this was communicated up front, trust wasn't lost.



Another problem was the fact that features were prioritized by business needs. Mounting pressure to get the project back on track meant that important refactoring decisions kept getting held back. Even if the developers could convince the project manager to let them refactor parts of the code, they wouldn't be able to fix everything within a single point. There was the sense that refactoring was a futile effort.

Trying to refactor a key component built by an external vendor required cooperation from the team, platform specialists, managers, and the client. Even though the team was able to identify the problem, it couldn't be solved unless multiple stakeholders agreed it was a problem that needed fixing. We learned to involve multiple stakeholders to solve these problems instead of isolating everyone into distinct sprints.

Could we have tried other solutions? Yes, there is always more than one answer to a given problem. So what else could we have tried?

One option would've been to make our sprints longer. We didn't think the client would have agreed. I'm pretty sure that the message "We are going to work in longer sprints and keep away while we do" would not have inspired a positive reaction.

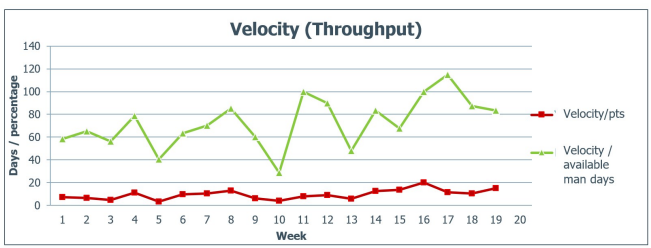
We could've changed our definition of *done*, except we weren't the only team involved. We had no control over the client's deployment process. If we changed the definition of what it meant to be done, the client was the one who had to deal with the consequences. We accomplished something similar by giving the client more visibility.

That was my view. Let's take a look at some metrics and hear what the people involved had to say.

Comparing Now and Before

The team has since then implemented most of the testing frameworks. The reason why I mention this is to show that you can overcome difficult technical hurdles and project management barriers. In the beginning, we couldn't envision ever having the time to put these fixes in place, but it happened. Many of the skills the team learned were transferred to the client's development team. The next step is to get the client's IT and project management teams on board.

We tracked our velocity throughout our project and can clearly see that we improved.



When we look at velocity in story points (the red line), we see that we averaged 7.25 story points per week in May, compared to 13.5 points per week in September. The velocity, normalized by the total number of man days (green line with the vertical axis denoting percentage), shows the May average was just 0.64, compared to September's 1.04.

I can't think of a better way of summing up than sharing the views of a team member.

```
The hardest part is to train yourself to embrace a new mentality.
To realize that if a task is coded, this does not mean the task is completed.
Kanban "forced" us to think about quality. When you have a column called
"Function test" on the board, it's a little bit hard to ignore it.
It forced us to stop the coding, coding, coding chain.

I think the best lesson we have learned was to always think about quality.
        ---Mariana, developer
```

Make Your Own Improvements

It's easy to fall into the trap of making short-term decisions when you're under heavy pressure. And when things don't get done, it's easy to blame others and their shortcomings rather than figuring out what's stopping them from doing a good job.

Visualization is essential in knowing what needs to be addressed. Good leadership is important, too.

The first step in good leadership is to clarify the common goal.

Try to resist the temptation to postpone improvement actions and defer decisions. Procrastination will set you back right off the bat. A good leadership strategy is to always do something, regardless of how small, to improve the current state. The effect of improvements is cumulative, so don't underestimate the effect of small improvements. By making many small improvements, you are also setting a good leadership example; people will do what you do, not what you *say* you would do.

The second thing to do in good leadership is to follow up with vigilance on improvement actions taken. Nothing derails trust and breeds resentment more than people not pulling their weight. Visualizing the improvement action and asking participants to report their progress in front of everyone else makes them accountable. This is a good way to follow up on improvement actions taken.

Stick with one improvement at a time. Investing in quality is always a wise choice.

Next steps

The turnaround by the development teams showed me the importance of staying cool under pressure and investing in quality, even when the customer is screaming for something else. The urgency of the project meant that we

really could not afford to put one foot wrong when we selected our improvements. Kanban helped with that precision. Kanban was also instrumental in helping the team and the customer to stay in focus, honoring commitments.

Let's leave the software realm for a bit and have a look at something completely different. How about Kanban outside IT? Can Kanban improve business operations? Check out the story of "Kanban in Back office" at a fast-growing bank.