active.collab

Fundamentals of AGILE Project Management

Contents

3 Introduction

4 Waterfall Project Management
- Phases in waterfall projects
- Advantages of waterfall
- Disadvantages of waterfall
- On what types of project you should use waterfall

10 Transition From Waterfall to Agile

13 Agile Project Management
- Agile values
- 12 agile principles
- Elements of agile project management
- When and who can use agile
- Advantages of agile
- Disadvantages of agile
- Agile project management in enterprises
- Agile methodologies and frameworks
- Agile tests

28 Is Agile Really That Different From Waterfall?

31 Case Study: How We Implement Agile Into Our Workflow
- How we used to do things
- From suggestion to feature
- Designing for the user
- Kicking things off
- Teamwork on display
- Putting it to the test
- Release day
- The result

46 Case Study: Agile Reporting
- Answers rather than raw data
- Design answer-oriented reports
- Implementing automated reporting
- Sprint report questions
- Real-world example of agile reporting

Introduction

There are two ways you can manage software development projects:

- **Waterfall**: plan everything in advance, then build according to the plan for the next whole year
- **Agile**: plan what you'll build in the next few weeks, and see how it goes from there

Agile is an approach to project management that favors responding to change over careful planning. It's important to emphasize that agile is not a methodology but a set of principles that define how we approach project management.

Before there was agile way, all software projects were managed using Waterfall techniques. To really understand agile, we first need to understand the traditional project management approach and how the software development industry changed.

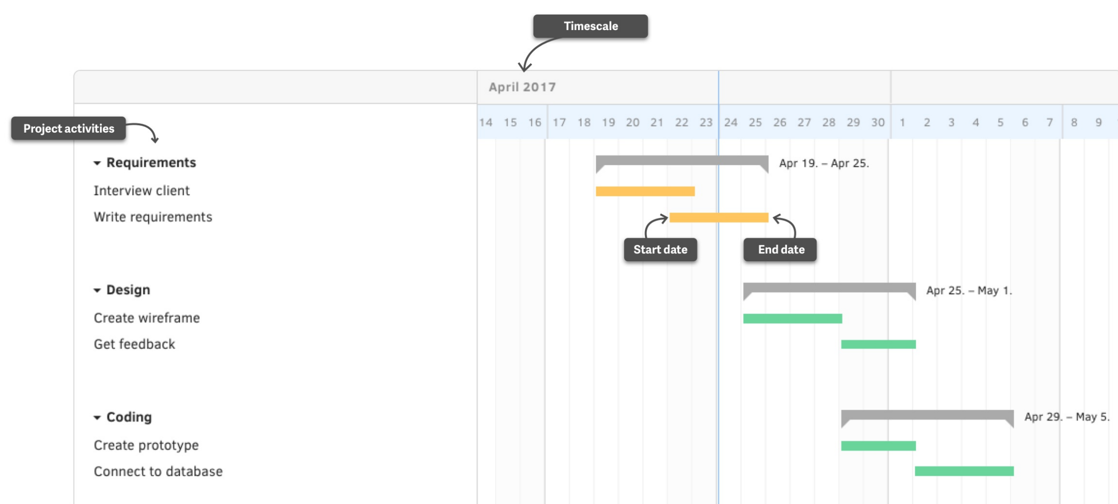
Waterfall Project Management

When you take traditional project management and apply it to software development, you get Waterfall. As such, no one invented waterfall - instead, we gave it name once we realised that there are others ways to manage projects.

In waterfall, a project is completed in distinct stages and moved step by step toward ultimate release to consumers. You make a big plan upfront and then execute in a linear fashion, hoping there won't be any changes in the plan.

Waterfall was the first software development methodology, inherited from manufacturing and construction industry where you can't afford to iterate (after you've built a tower or a bridge you can't go back to "improve" the foundation). But because software is prone to frequent change, waterfall is not the best solution.

Waterfall is often mentioned algside Agile and stands in contrast to it. The main difference between them is that waterfall doesn't react well to frequent changes, which is why it gets a bad reputation in software development community, where frequent changes are the norm.



Most common way of planning waterfall projects using Gantt chart

Phases in waterfall projects

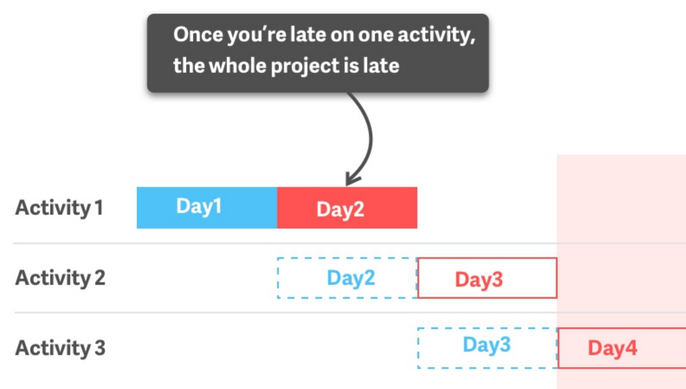
All projects are grouped by type of activity and each projects follows the same phases:

- **Requirements** - where we analyse business needs and document what software needs to do
- **Design** - where we choose the technology, create diagrams, and plan software architecture
- **Coding** - where we figure out how to solve problems and write code
- **Testing** - where we make sure the code does what it supposed to do without breaking anything
- **Operations** - where we deploy the code to production environment and provide support

Once you put all the activities on a Gantt chart, you get something that looks like slopes of a waterfall, hence the name.

Usually 20–40% of the time is spent on requirements and design, 30–40% on coding, and the rest on testing and operations.

Activities on waterfall projects have to happen in the exact order and one set of activities can't start before the previous one ends. This is why planning is the most important thing on waterfall projects: if you don't plan right, a phase will be late and will push every other subsequent phase, thus putting the whole project over deadline.



The problem with using waterfall method on software project is that planning is very tricky in software development. You can never be 100% sure how much time you'll need on something or how much time you'll spend debugging. As a result, waterfall is risky.

Advantages of waterfall

Extensive documentation

Because you can't go back to a previous activity, you're forced to create a comprehensive documentation from the start, listing all the requirements you can think of.

Knowledge stays inthe organization

When you have extensive documentation, knowledge won't get lost if someone leaves. Also, you don’t have to spend time on training new members as they can familiarise with the project by reading the documentation.

Team members can beter plam their time

Because everyone knows in advance on what they'll work, they can be assigned on multiple projects at the same time.

Easy to understand

aterfall projects are divided in discrete and easily understandable phases. As a result, project management is straightforward and the process is easily understandable even to non-developers.

Client knows what t expect

Clients can know in advance the cost and timeline of the project so they can plan their business activities and manage cash flow according to the plan.

Client input not reuired

After requirements phase, client input is minimal (save for occasional reviews, approvals, and status meetings). This means you don't have

coordinate with them and wait for when they're available.

Easier to measure

Because waterfall projects are simple, it's much easier to measure your progress by quickly looking at a Gantt chart.

Better design

Produts have a higher cohesion because during the design phase you know everything that must be taken into account. There is no one-feature-at-a-time problem that leads to usability problems down the road.

Disadvantages of waterfall

No going back

Once you're finishedOnce you're finished with one activity, it's difficult and expensive to go back and make changes. This puts a huge pressure on the planning.

No room for error during requirements phase

Everything relies heavily on the requirements phase and if you make an error, the project is doomed.

Deadline creep

Once one activity is late, all the other activities are late too, including the project deadline.

QA too late to be useful

Testing is done at the end of the project which means that developers can't improve how they write code based on QA feedback.

Bug ridden software

Because the testing is done at the end, most teams tend to rush the testing in order to deliver the project on time and hit their incentives. This short-

term wins lead to sub-par quality and long-term problems.

Not what the client actually needs

Most of the time, clients can't articulate what they need until they see what they don't need. If the client realizes they need more than they initially thought, the project plan will need a major overhaul (as well as the budget).

Unexpected problems

Designers can't foresee all the problems that will arise from their design, and once those problems surface, it's very difficult to fix them.

On what types of project you should use waterfall

Waterfall is suited for projects where:

- Budget, requirements, and scope are fixed (eg. you're building a one-off project which doesn't need further development)
- You can accurately estimate the work (you're familiar with technology and you've done the same work before)
- You can't afford to iterate (eg. you're building a heart rate monitoring software)
- Project is innately low-risk (you're building a clone of something that already works)
- Project has a hard ship date (eg. you have to ship a video game by Christmas)
- Your users can't or won't update software (doesn't apply to web applications where updates are seamless)

You shouldn’t use waterfall:

- Where a working prototype is more important than quality (eg. you first need to test if there’s a market demand)

- When you don't know what the final product should look like
- Where client doesn't know exactly what they need
- When the product is made for an industry with rapidly changing standards
- When you know you won't get the product first the right time and have to incorporate user feedback
- When your users are happy with v1.0 and you can ship additional features as time goes on

Whether you'll use agile or waterfall doesn't matter on your preference but type of project and your customer/client. While strictly speaking agile is better for software development (as according to the Standish Group 2015 Chaos Report), if you can't iterate, you have to use waterfall.

Size	Method	Successful	Challenged	Failed
Small Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%
Medim Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Large Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Total	Agile	39%	52%	9%
	Waterfall	11%	60%	29%

Standish Group 2015 Chaos Report: Waterfall vs Agile

Transition From Waterfall to Agile

To understand how and why agile came into being, we first need to understand what was it like to develop software in the past and what changed in the meantime.

First, we used to manage software development projects using traditional project management techniques because we inherited them from other industries (like construction). The traditional approach consists of dividing a project into phases, planning everything out on a timeline using WBS and Gantt charts, and then following the plan.

But there's a problem - the traditional approach works nicely when you don't expect any changes in the middle of the project. But in software development, things are always changing. If you plan what you'll develop for the next 5 years, as soon as there's some new technology (eg. new protocol, new service, new hardware) your 5-year plan will become obsolete.

Second, the software economics worked differently in the past. In the 1980s, the cost of owning and maintaining software was twice as expensive as developing it. Today, it's reverse: everyone can run software on their personal computer and developers can deploy their own apps on AWS or Heroku within minutes.

Third, internet changed how we develop and consume software. With the rise of internet, the number of people who use software on a daily basis exploded. As the market demand for software expanded (driven by consumers and small businesses), any developer could quit their corporate job and start their own company. Software development was no longer a market reserved only for big players.

This democratization of software had a profound effect on how we develop software. But that's only half of the story.

We also have to take into the consideration that 3/4 of all software products used to fail because they didn't meet requirements or weren't used. That's a

lot of money down the drain.

So, one hand we have a rise of small software development companies, and on the other, we have market uncertainty and a lot of risk.

Small software development companies needed a simpler, faster, and less riskier way to develop software. They couldn't afford to spend several months planning only to end up having to rewrite the whole thing because there was some fundamental error in their logic. Plus, they needed to move fast so they could capture the market and start making money as soon as possible.

Several lightweight methodologies were developed that would let companies have prototypes faster:

- 1991 Rapid Application Development
- 1994 Unified Process and Dynamic Systems Development Method (DSDM)
- 1995 Scrum
- 1996 Crystal Clear and Extreme Programming (XP)
- 1997 Feature-Driven Development

Today, we refer to all those methodologies as agile methodologies (because they follow agile principles, as defined in the Agile Manifesto). But at the time, they all were just a bunch of disparate methodologies, living in their own universe.

The initial adopters of these early agile methods were small-to-medium-sized teams who worked on unprecedented systems where it was difficult to scope requirements. To design and achieve product/market fit of those systems, you needed exploration and constantly change thing until you get the system right.

Thanks to methodologies that favored rapid-prototyping, small product companies could create and release the main software faster and add new features as time goes on.

Rapid-prototyping gave smaller companies an advantage because they could react to market needs faster than an enterprise. While enterprises relied heavily on documentation and long lead time, startups could build a prototype, test it, and ship in the fraction of the usual time.

Heavy, document-driven processes (like TickIT, CMM, and ISO 9000) supported the huge organizational structure of enterprises, but it hindered their innovation and market responsiveness.

Product companies benefited the most from these early agile methodologies, but development shops discovered that, in some cases, they too could benefit from a more lightweight approach (although that depended on the nature of their clients).

All the disparate lightweight project management trends came together in 2001 when 17 software developers met in Utah to discuss their processes. Together, they defined the concept of agile software development in Agile Manifesto.

Agile Project Management

Today, when we say that some methodology is agile, it means it follow the value and principles from Agile Manifesto.

Agile Manifesto recognizes that there's no one-size-fits-all, so it doesn't prescribe how to run projects. Instead, it lays out guidelines on how to best manage software projects.

Agile values

Most important part of Agile Manifesto are the 4 values. They are the heart of what it means to be agile.

Agile values help you focus on what's important. For example, one of the values is "working software over comprehensive documentation". It doesn't mean that documentation is bad - it means that if you have to choose whether to spend your time on writing a detailed user story or fixing a bug, you should choose the latter.

Individuals and interactions over processes and tools

Knowledge workers prefer autonomy. So in software development it's more important to let people solve problems by collaborating than forcing them to follow a procedure for the sake of satisfying some dusty policy.

Every company needs processes (especially after they've grown to a certain size), but you must know why a rule is in place and when you should break it. For example, when daily standups stop being useful, don't force them just because some agile methodology says you must have them.

The way you know when process doesn't work is when people can't collaborate efficiently anymore. People are the engine behind every project. If they can't interact because of hierarchy or a lengthy/complex protocol, they have to spend more time on managing tools and processes than doing their job.

> *"Good process serves you so you can serve customers. But if you're not watchful, the process can become the thing. The process becomes the proxy for the result you want. You stop looking at outcomes and just make sure you're doing the process right. It's not that rare to hear a junior leader defend a bad outcome with something like, "Well, we followed the process." A more experienced leader will use it as an opportunity to investigate and improve the process. The process is not the thing. It's always worth asking, do we own the process or does the process own us?" - Jeff Bezos*

> *"There's something really wrong with our definition of what a 'completed project' is. If it means 'Did Chris get all his project tasks done?' then it was a success. But if we wanted the project in production that fulfilled the business goals, without setting the entire business on fire, we should call it a total failure." - The Phoenix Project: A Novel About IT, DevOps, And Helping Your Business Win*

Working software over comprehensive documentation

In traditional project management, phases happen in sequence and if you mess up the first phase (requirement gathering and documentation), every other phase will suffer. That's why waterfall needs comprehensive documentation. But on agile project we expect things to change.

Do you really want to spend your whole time updating the documentation? What matters the most is having a working product that real users can test. If you had to choose between fixing a bug and writing a report on it, fixing it is the best use of your time.

This doesn't mean that you should forsake documentation. Developers fall in this trap often and write terse one-line user stories, which creates trouble for QA and maintainers because they can't figure out the proper user acceptance criteria.

The perfect documentation should be "Just Barely Good Enough". Too much and it goes to waste or can't be trusted because it's out of sync with code; Too little and it's difficult to maintain and get new team members up to speed.

When writing documentation, you should ask yourself what would you want to know if you joined the team tomorrow and document based on that. If you have trouble with documentation, grab a copy of Living Documentation by Cyrille Martraire.

Customer collaboration over contract negotiation

Contracts create the culture where change isn't an option. Agile creates the culture where change is expected. But how do you manage change? By collaborating with customers.

Agile presupposes that you have unlimited access to your customers and that you can always sit down with them and talk. Developers are natural problem solvers but they need access to the customer so they can better understand what the real problem is.

Contracts are useful, but they have a nasty side effect: people tend to care more about delivering the project within time and budget than fulfilling the real business goal. Further, when the team falls behind schedule, they are pressured to get things done which results in frustration, panic, and lower quality.

Also, when you sign a contract early in the lifecycle, you're guesstimating and more often than not, you're wrong. But you still try to hit the milestone even though they have nothing to do with real needs.

That's why agile favors customer collaboration and delivering work in small increments. This lets scope work as you gather more information and discover what you don't know.

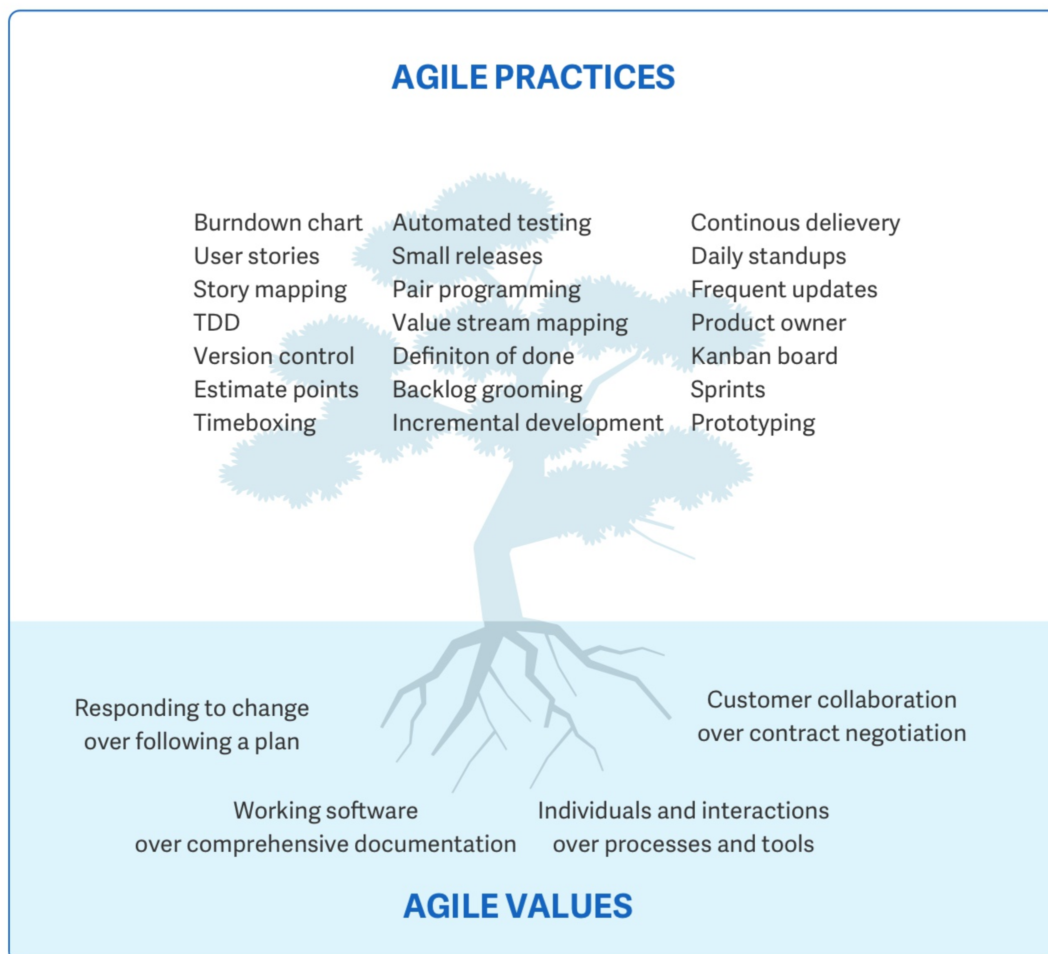
Responding to change over following a plan

The more time you spend on planning, the more you resist changes lest your efforts go to waste. But the goal is not to deliver project according to the plan (within time and budget) - the real goal is to satisfy some business goal, and if it means completely changing your plan, then so must be it.

It's more important to build what you really need than to blindly follow an obsolete plan. Developers may hate it when they their code becomes

invalidated, but clients hate it even more when they don't get the product they need.

That's why agile favors shorter lead time and encourages teams to chop things up in smaller deliverables so they won't have to redo large chunks of work. This means that you are never done with requirements gathering and design phases but you continually revisit them throughout the lifecycle.



12 Agile Principles

1. The main goal is satisfying the customer. All your metrics should be tied to the goal.
2. We don't what the customer really needs. It's best to give them a working product as soon as possible, listen to feedback, and change based on the feedback. We should welcome change even late in development because useful software is more important than deadlines.
3. Release updates as soon as they're finished, preferably every two weeks.
4. To build what the customer needs, business people and developers must work together every day.
5. Motivated individuals are the heart of every project. Give them the support they need and don't micromanage them.
6. Nothing can replace face-to-face conversation.
7. Working software is the primary measure of progress
8. Death marches and crunches are counterproductive. If you can't develop sustainably and maintain a constant pace indefinitely, rethink your processes.
9. Don't let the technical debt build up. You'll be busy extinguishing fires and won't be able to respond to changes on time when you really need to.
10. Say no to feature requests customer doesn't really need.
11. Self-organizing teams are the most efficient organizational structure.
12. The team should reflect on their past work at regular intervals and improve.

Elements of agile project management

Culture where change is expected

Agile isn't about using Kanban boards, having daily standups, or anything similar (those are elements of specific agile methodologies). Agile, at its core, is mindset where everyone, from employees to clients, expects change. You can't promise your client everything at once or a firm deadline because both you and the client know that's unrealistic. But you can promise them that you'll give them something they can use and listen.

Incremental development

Each iteration builds on previous work, making the product better gradually. Also, you don't wait for completed work to pile up before releasing it all at once - you release it as soon as it's finished. An iteration might not add enough features to warrant a marketing campaign, but that doesn't matter because the ultimate goal is to give customers value.

Frequent releases

Because software is developed incrementally, you can have shorter cycles, where at the end of cycle you ship new features/updates. This way, customers can get value as soon as possible and validate it. If the work doesn't satisfy their needs, you can learn that before you spend more time on development.

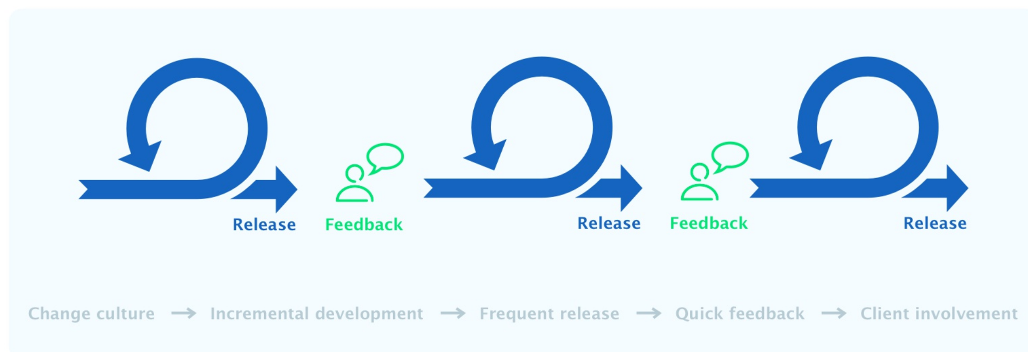
Short feedback loop

Because releases are more frequent, you can get feedback faster. And because you can get feedback faster, you can more quickly change the product and give value.

High level of client involvement

In order to reap the full benefits of short feedback loop and frequent releases, you need a high level of client involvement. You need to talk to your client after each cycle and see how they use the software and if it they derive value from it.

Sometimes the client is not available to give you feedback. Some agile methodologies (like Scrum) solve this problem by having a special role on the team called Product Owner. This person serves as a customer representative and act on behalf of the customer. If a developer has any question, they ask the product owner instead of the customer. Product owner also reviews progress and re-evaluates priorities at the end of each iteration.



When and who can use agile

Today, agile is such a buzzword that teams outside software development try to incorporate it into their workflow. But agile is not for everyone.

For example, a marketing agency can never implement agile because clients don't want to pay for a half-finished marketing campaign and iterate. There are revisions, but their number is clearly specified in the contract. Plus, there are no such thing as a "working increments" - you either have the deliverables or you don't.

Agile isn't the right approach for every software project either. If you don't have access to customers, can't iterate, or if you have a complex organizational structure, it's very difficult to adhere to agile principles.

Agile works best when:

- You can't estimate the time you'll need and don't know the full scope of

requirements

- You don't know whether there's a need on the market for your software
- You can't map the business needs and the design needs to emerge through trial and error
- You have unlimited access to your customer who's ready for extensive involvement
- You can afford to iterate and don't need to deliver a fully functional software at once
- Neither you nor your client have a complex bureaucracy that delays decision
- Clients don't have a fixed budget/schedule
- You need to capture the market before there's any competition
- Your customers don't have trouble updating their software (or don't even notice it, eg. they use a web app)

As you can see, agile is more suited for small-to-medium size organizations than corporations. The reason is simple: the less people there are, the easier it is to make a decision and respond to change. Also, agile is more suited for product companies over consultancies.

Agile is also great for startups, where "fail fast" is the dominant mantra. Venture capitalists encourage startups to try crazy ideas and let the markets do the work. Most of the ideas will fail those few that succeed will change the world.

Advantages of agile

- You can deploy software quicker so your customer can get value sooner rather than later
- You waste less resources because you always work on up-to-date tasks
- You can better adapt to change and respond faster
- Faster turnaround times

- You can detect and fix issues and defects faster
- You spend less time on bureaucracy and meaningless work
- There's a big community of agile practitioners with whom you can share knowledge
- You can get immediate feedback (which also improves team morale)
- Developers can improve their coding skills based on QA feedback
- You don't have to worry about premature optimization
- You can experiment and test ideas because it costs are low

Disadvantages of agile

Agile has strong advantages but it's important to know the limitations and risks it brings.

- Documentation tends to get sidetracked, which makes it harder for new members to get up to speed
- It's more difficult to measure progress than in waterfall because progress happens across several cycles
- Agile demands more time and energy from everyone because developers and customers must constantly interact with each other
- When developers run out of work, they can't work on a different project because they'll be needed soon
- Projects can become ever-lasting because there's no clear end
- Scope creep and experience rot
- Clients who work on a specified budget or schedule can't know how much the project will actually cost, which makes for a very complicated sales cycle (until iteration ends is not something clients like to hear)
- Product lacks overall design, both from UX and architecture point of view, which leads to problems the more you work on the product.
- Teams can get sidetracked into delivering new functionality at the

expense of technical debt, which increases the amount of unplanned work

- Features that are too big to fit into one or even several cycles are avoided because they don't fit in nicely into the philosophy
- You need a long term vision for the product and actively work on communicating it
- Products lack cohesion and the user journey is fragmented because the design is fragmented. The more time passes, the more disjointed software ends up.
- Short cycles don't leave enough time for the design thinking process so designers have to redevelop the experience over and over due to negative feedback.
- Check here for some more Scrum sprint planning anti-patterns and product backlog and refinement anti-patterns

> *Workers in tech don’t usually feel like they have the ability to focus on craft — especially when it comes to visual design. When you're constantly iterating, constantly pushing new versions out, you can’t invest time in seemingly unnecessary details that will be lost in tomorrow’s update. - Jessica Hische*

Agile project management in enterprises

Big companies generally have a problem of being too slow, with too much WIP, and too many features in the backlog.

Traditionally, big companies would need 6 months for gathering requirements, another 6 for development and testing, plus worry if they have an opening in their development schedule and capital for the feasibility study to get things going.

There's also another problem. Big companies need new features to stay competitive but features are always a gamble. About only 10% of new features get desired benefits. The faster you can put out features on the

market and test them, the faster you can differentiate the heroes from the zeroes and recoup the invested capital.

> *"Products need to ship in six months. Otherwise, some Chinese company will steal our idea, have them on our competitor's store shelves, and take the majority of the market.*
>
> *In these competitive times, the name of the game is quick time to market and to fail fast. We just can't have multi year product development timelines, waiting until the end to figure out whether we have a winner or loser on our hands. We need short and quick cycle times to continually integrate feedback from the marketplace.*
>
> *But that's just half the picture. The longer the product development cycle, the longer the company capital is locked up and not giving us a return. The CFO expects that on average, R&D investments return more than 10%. That's the internal hurdle rate. If we don't beat the hurdle rate, the company capital would have been better spent being invested in the stock market or gambled on racehorses.*
>
> *When R&D capital is locked up as WIP for more than a year, not returning cash back to the business, it becomes almost impossible to pay back the business." - The Phoenix Project: A Novel About IT, DevOps, And Helping Your Business Win*

When big corporations decide to be more agile, they need methods that work at a large scale. They also need a lot of cross-functional coordination so Development, Operations, and Sales & Marketing can work together efficiently.

The best approach that enables corporations to be more agile is the DevOps movement, which usually consists of:

- Work visualisation with Kanban boards
- Faster time to market for new features
- Lower failure rate of new releases
- Shortened lead time between fixes

- Faster emergency changes and recovery
- Small and frequent releases
- Controlling WIP
- Faster feedback loops
- Ensuring quality from the start
- Continuous delivery and automated deployment
- Aligning cycle time of operations with cycle time needed to keep up with customer demand (called takt time)
- Standardized environments across development, QA, and production
- Short sprint intervals and reduced planning horizon
- Scrum-like team roles

Big companies often combine Agile and Waterfall, where they use practices from agile methodologies (like sprints and roles) and the efficancies of waterfall, thus creating an Agile-Waterfall Hybrid.

Agile is not a silver bullet to late projects. Big companies who choose to go agile face a whole new set of problems and risks. Here are some side effects that happened at Microsoft once they switched to agile mindset:

> *"The problem happens when the entire company is completely and totally focused on developing an absurd number of new features and products, giving them completely unrealistic deadlines, and then shipping software on those deadlines no matter how buggy it is.*
>
> *The idea is that everything is serviceable over the internet now, so they can just "fix it later", except they never do. This perpetuates a duct-tape culture that refuses to actually fix problems and instead rewards teams that find ways to work around them.*
>
> *The talented programmers are stuck working on code that, at best, has to deal with multiple badly designed frameworks from other teams, or at worst work on code that is simply scrapped. New features are prioritized over all but the most system-critical bugs, and*

teams are never given any time to actually focus on improving their code." - Microsoft employee

Agile methodologies and frameworks

Agile is a philosophy and not a methodology. There are some methodologies (like Scrum) that happen work well with it, but if you don't embrace the philosophy, the methodology won't do anything.

Agile doesn't say how you should manage project; instead, it only provides guidelines and is open to interpretation.

Methodologies by their nature prescribe how you should do things. It may seem Agile rejects all methodologies, but that's not true:

- Agile embraces documentation but rejects hundreds of pages of never-maintained and rarely-used tomes.
- Agile embraces planning but recognize the limits of planning in a turbulent environment.
- Agile embraces processes but only as long as they serve a real purpose and not for the sake of filing some diagram in a dusty corporate repository.

There are a lot of methodologies and frameworks that are inspired by agile principles, but the line that separates agile and waterfall is often blurry.

Methodologies aren't 100% agile or 100% waterfall. Instead, they occupy different places on adaptive-predictive continuum:

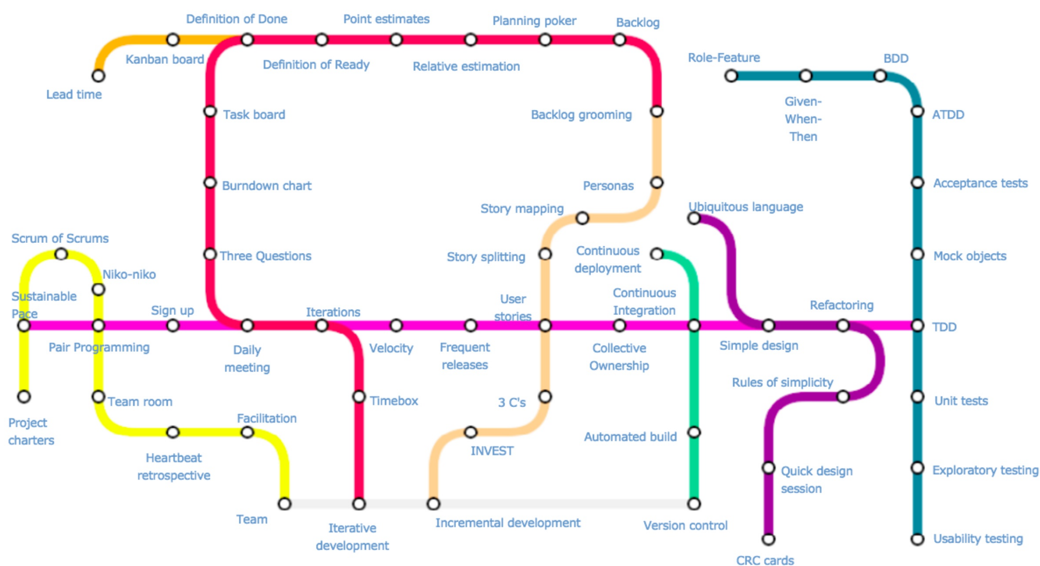
- Agile methodologies lie on the adaptive side (they favor responding to change)
- Waterfall methodologies lie on the predictive side (they favor planning)

Every methodology needs planning and responding to change. But what separates them is what they favor more.

Also, you can divide methodologies based on what they focus on:

- Some provide best practices (like Extreme Programming and Pragmatic Programming)
- Some control flow of work (like Scrum and Kanban),
- Some support activities for requirements specification and development (like Feature-Driven Development)
- Some cover the full development life cycle (like DSDM and RUP).

Agile Alliance has a very handy map that lists the most common practices from different agile methodologies.



Lines represent practices from the various Agile "tribes" or areas of concern:

“Subway Map to Agile Practices” by Agile Alliance

The most popular methodologies that are consistent with agile principles are Scrum, Kanban, XP, Lean, and DSDM. Scrum in particular is so popular that it became synonymous with Agile.

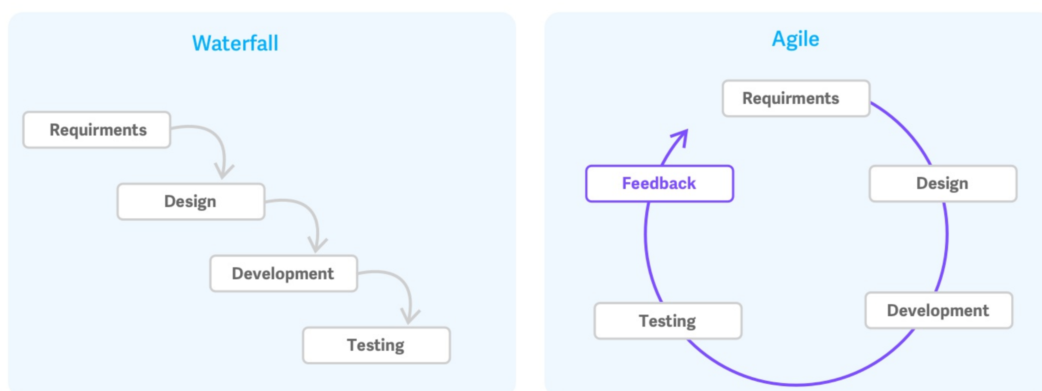
Agile tests

There are several of informal tests you can take to asses whether your organization is agile (or complies to Scrum's best practices):

- Karlskrona test http://mayberg.se/learning/karlskrona-test
- 42 Point Test http://www.allaboutagile.com/how-agile-are-you-take-this-42-point-test/
- Kniberg's Scrum checklist https://www.crisp.se/wp-content/uploads/2012/05/Scrum-checklist.pdf
- The Nokia Test http://agileconsortium.blogspot.rs/2007/12/nokia-test.html
- The Joel Test https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/

Is Agile Really That Different From Waterfall?

Waterfall is always mentioned as the antithesis to Agile, which makes sense. After all, waterfall projects have a hard time dealing with changes while agile projects welcome change. At least in theory.

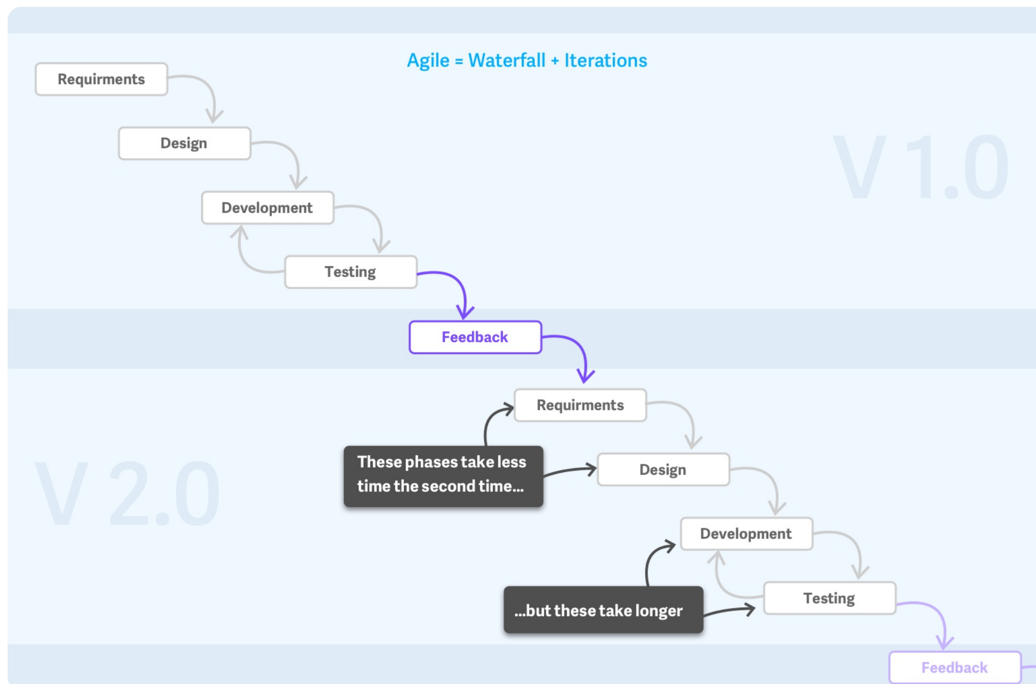


The truth is, no matter what methodology you use, change is not a good thing. Change always means additional scope, delay, and expenses. Agile is better at minimising the effects of change, but they still happen. Also, agile teams have the culture where change is OK, which is maybe the most important benefit of being agile.

But once you scratch behind the surface and look both from purely process perspective, waterfall is very similar to agile.

Once you break down any agile workflow, you'll still get a set of activities that follow one another, which eerily resembles Waterfall. And if you treat waterfall projects as smaller phases within a big project, you'll end up with agile.

In other words, activities on a project are waterfall and if you treat the whole project as a series of iterations, it’s agile.



Whether you're agile or waterfall ultimately depends on whether your client expects the first version to be bad. And waterfall projects are projects where the client decided on zero iterations.

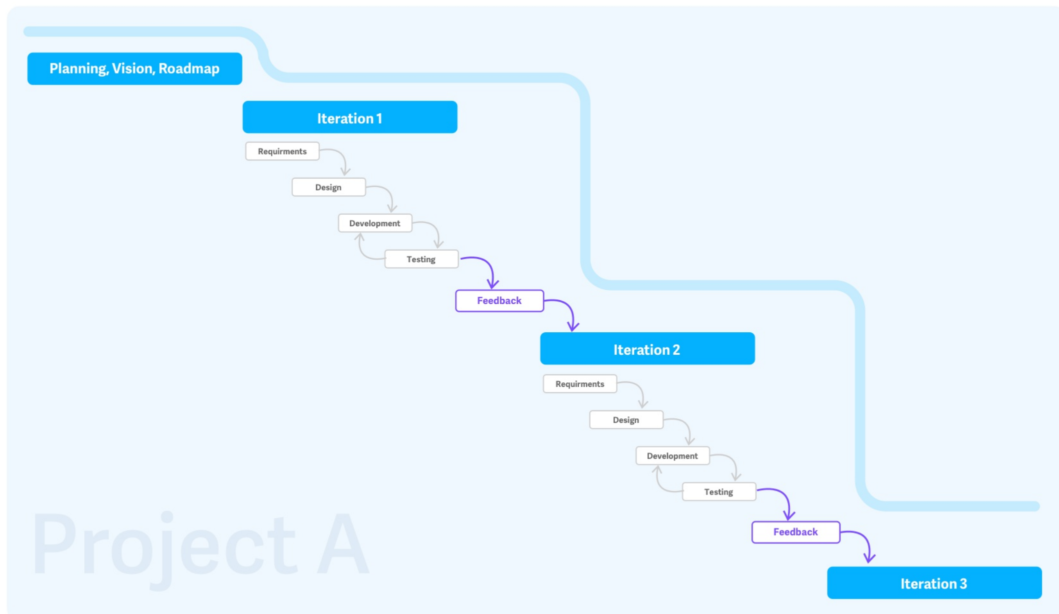
> *"In agile projects, the number of iterations is decided on by the customer. Because things are all done within an iteration in agile, the logical assumption was that an iteration equaled a project. But an iteration is more properly referred to as a phase or subphase of the project." - PMI*

As you can see, agile still fits in the traditional project management, only the point of view changes. Instead of treating each iteration as a separate project, iterations are just phases in one big project.

The real difference between the waterfall method and agile is that in waterfall the clients is heavily engaged at the beginning of the project and then their engagement declines; while in agile, the client is constantly engaged.

So what this all means in practice? It means no organization is purely agile

or waterfall. Agile and waterfall are more about the culture and type of work the organization does than how they do it. You'll find that most organizations divide the project into waterfall milestone but work according to agile principles between those milestones.



"A problem common with comparing agile and waterfall is the labeling. Few, if any, companies are purely "agile" or "waterfall". They are more mindsets that encompass a wide variety of practices and approaches to development. Labels are convenient for helping make an argument, often with cute little straw-man statements to help reinforce preconceived notions." - Clinton Keith

Case Study: How We Implement Agile Into Our Workflow

When businesses rely on your app for their day-to-day work, you have to be agile enough to quickly address their needs. If you don't, others definitely will. In the unforgiving world of SaaS, delaying a critical feature (or rushing a bug-ridden piece of code) will mean losing clients.

Therefore, the development process needs to run smoothly and up to a standard, with delays reduced to a bare minimum. Before any change makes its way to the end user, it goes through five crucial phases: feedback, design, development, quality assurance and deployment. Here's what we've learned (the hard way) about each of the stages from over ten years of software development.

How we used to do things

Before we get into the details, let's look at how this all came about. After years of adding features without enough scrutiny, our app was suffering from feature bloat. We've added so much functionality over the years that new users were intimidated by the sheer complexity of the UI (never to return again). We knew we had to rebuild from the ground up, even if that meant rewriting every single feature from scratch.

Then came the delays. The features for new versions were constantly lagging behind schedule. For example, a junior developer was supposed to develop an integration with QuickBooks. We didn't accurately predict the complexity, skills or knowledge needed. Plus, he was the only one assigned to that task, and no one could quickly jump in to help him out. As a result, what was supposed to be a two-week job ended up taking five. Those were a few red flags.

It was clearly time to switch to a more agile approach. We took what we liked from various agile (and not so agile) methods, combined it with experience, and came up with our own version, which has been delivering

great results ever since.

This is the road a feature must travel before it's offered to the end user; to ensure quality, a new feature is introduced only after it has gone though all of these stages:

From suggestion to feature

In our workflow, a new feature starts taking shape long before it reaches the development team, and it's usually born of user feedback. This is no coincidence — we used to release features no one needed, usually just because one user was particularly loud or we simply thought something would be great to have. Instead of imagining what features our users might need, we now make decisions based on actual demand.

If you're into lean manufacturing, you'd say that our customers "pull" certain features by requesting them more often than others. Our support and sales teams are a big part of the process because they're constantly in touch with users who share their needs and ideas.

Based on the feedback, our teams update a form. Feedback collected and saved using this form is essential for deciding which features make their way onto the road map.

When we don't have all of the information we need, we'll reach out to customers directly. This is usually done with targeted surveys on a carefully selected sample. The point is that we listen. No feedback is lost on us. It's always acknowledged and documented.

AC Customer Feedback

Category

- Projects
- My Work
- People
- Invoicing
- Calendar
- Reports
- Notifications
- Cloud
- Self-Hosted
- Mobile
- Integrations
- Competition
- UI
- Друго:

Subcategory

Projects, Tasks, Templates, Discussions, Files, Time, Expenses, Updates, Activity, Email, Morning Paper, Roles, Companies, Teams, Invoices, Estimates, Budget, Payments, Events, Timer, API, Webhooks, Basecamp, Trello, Teamwork, Android, iOS, Translations, SupportYard, Plans, Customization, Calendar, Enterprise, Settings, Notes, Labels, Text Editor, Chat, Desktop, Repositories, etc. (pls copy/paste),

Ваш одговор

Feature

Global Gantt, Task Dependencies, Enterprise, Branding removal, Recurring tasks, Documentation, Free plan, Client+, Chat, File preview, Real Time, Configurable notifications, Timer for Linux, Quickbooks, Xero, Google Drive, Dropbox, SupportYard, Zapier, GitHub, Proofing, Private comments, Russian, sharing, Workload management, Estimate items to project tasks, Task List Change Notification ...

Ваш одговор

Description

Ваш одговор

The next step is analysis. We tally the scores each month to see which feature got the most votes. Also, with proper categorization, we get a broader perspective on which parts of our software need work. For example, if the calendar is getting many complaints, we'll consider revamping the entire section, rather than introducing the feature that got the most votes (such as calendar exporting).

However, even when the results are in, the decision to introduce a feature isn't final. Before it makes it onto our to-do list, we always do a thorough sanity check:

- What benefits will this feature bring to users?
- Does it fit our product philosophy?
- Will it add unnecessary complexity?
- Does it blend in nicely with the existing interface and functionality?
- Do we have the resources to deliver it in a reasonable timeframe?
- When a feature passes the checklist and the product owners approve it, it's time to go to the drawing board.

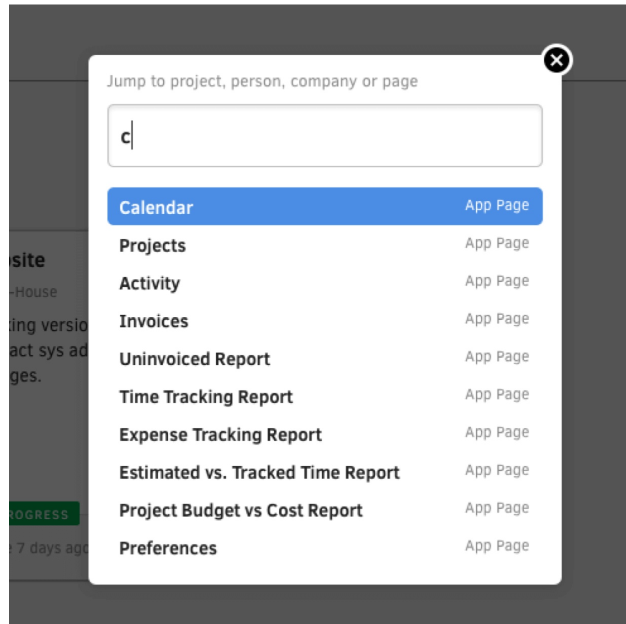
Designing for the user

Experience has taught us that adding a new feature doesn't just mean sticking it on top of the UI — you have to blend it with the existing design, with the user in mind. If you don't do this, you'll soon end up with an app so complex that only a brave few will make it through the first five minutes of the trial (yes, we've been there). Aesthetics are also important for a good first impression, but our main concern is ease of use. A feature needs to be added in a way that feels natural to the user.

We keep things in perspective by asking: where would the user expect this option to be?

For existing users, it's important that the new design follows the patterns they're familiar with and doesn't disrupt their workflow. Also, there are industry standards and conventions that we're all (unconsciously) used to. Never expect your users to change their habits completely. They'll more likely look elsewhere if the interface is not intuitive.

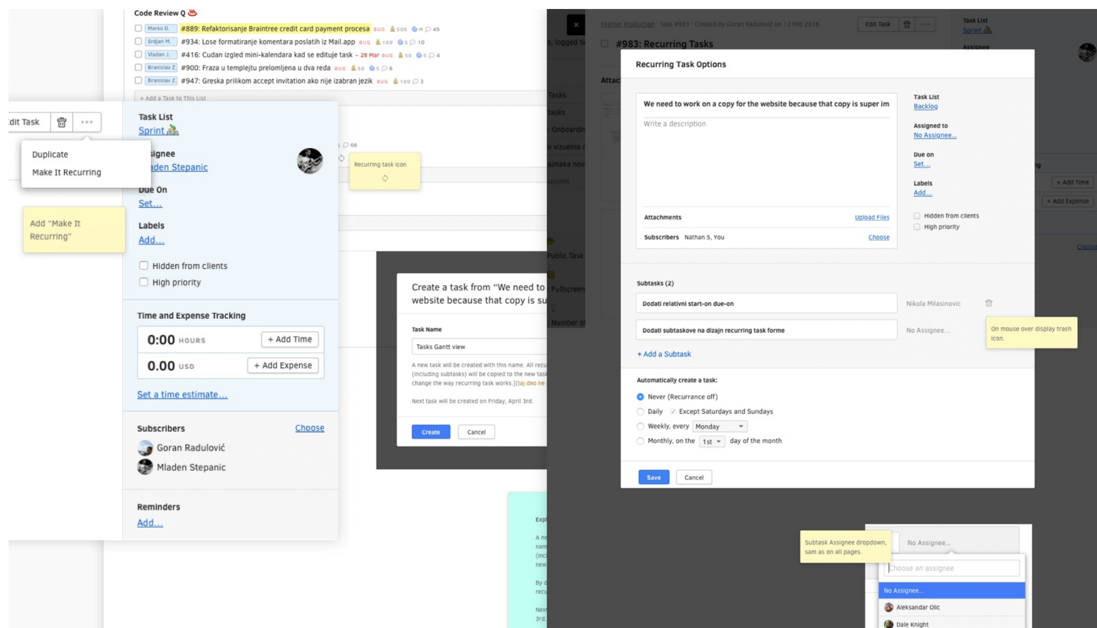
Take keyboard shortcuts for example. You could make your own set of shortcuts and expect users to learn them (they probably won't). Or you could add ones that people already use. A lot of our users already use Slack, for example, so we added a shortcut they are already familiar with (Command + K for quick jumps works in Active Collab as well).



When the user flows are in place, our UX designer mocks up the design in Sketch. We rarely use HTML in the early stages. Well-thought-out Sketch visualizations give us enough flexibility that we don't have to do any backtracking when we start coding. The initial design usually ends up matching about 80% of the final product. The rest is added and adjusted along the way.

Another important step of the design process is copy. Our copywriters devote a great deal of attention to every single word. We even have our own style guide, to sound as approachable and to be as easy to understand as possible. For example, saying "security certificate" instead of "SSL certificate" conveys in layman's terms something an average user might not be familiar with.

When all this is done, the feature is assigned to a development team. The team is made up of a project manager, a lead developer and a number of back- and front-end developers, depending on the workload. The project manager makes sure everything runs smoothly and on schedule, while the lead developer takes care of the technical side of things. They also coordinate and mentor junior developers.



Kicking things off

Our kickoff meetings aren't boring motivational get-togethers. They are opportunities for a development team (consisting of junior and senior developers) to meet with the project manager and product owner.

Instead of allowing empty monologues, we focus on putting everyone's words into actionable tasks. Throughout the day, three important meetings take place:

- The product owner presents the given feature to the team, the ideas behind it, the initial designs and the expected results.
- Then, the team has its own meeting in which it discusses the action plan, potential problems and the testing schedule.
- The final meeting is attended by everyone involved, and the plan takes its final shape. At the end of this meeting, the team gives estimates for demos and a final due date.

Three meetings might sound like a lot for one day, but that's how we make sure problems are solved early on. Filling in the blanks at this stage saves

our developers a lot of time, false starts and backtracking. The meetings also encourage teamwork and make everyone feel that their contributions are welcomed.

After the meetings, the team will have all of the necessary information:

What?

This is the scope of the feature and includes everything that needs to get done, as well as potential blockers and bottlenecks. We try to anticipate as many scenarios and edge cases as possible. Predicting all of them is not easy; they often come up as we go.

Why?

The product owner estimates the business value of a feature and explains why it's worth the effort. This way, the team gets a clear picture of the benefits to customers and the product. The prime motivator here is that everyone should understand why their work matters and how it contributes to the company as a whole.

How?

By outlining the steps required to complete a feature, we make sure our developers never lose their compass. We also set realistic expectations by adding a complexity tag. We went with t-shirt sizes — S can be solved within a few hours, while XXL takes weeks or more to complete.

Who?

The team lead is responsible for finishing a feature or task on time and for updating the product owner on the progress. Other team members are assigned to their own set of subtasks, so that it's perfectly clear who is accountable for what. Keeping this as transparent as possible helps us to see whether someone is struggling or causing delays.

When?

With everything taken into account, a due date is estimated. If a task is delayed, we analyze the reasons and use that experience the next time.

That way, a missed deadline becomes a learning opportunity and not a source of stress.

Kickoff day can get hectic, but it's also very fruitful. Everyone pitches in with ideas and comments. A task transforms from an empty slate to a hub for comments, edits and updates. By the end of the day, the development team has a clear picture of the work ahead and solid ground to build upon.

All important information is available in the task. This is also where team members communicate and post updates on their progress.

We go through this checklist before beginning work:

✓ Feature explained

✓ Steps for completion outlined

✓ Business value assigned by product owner

✓ Complexity assigned by team

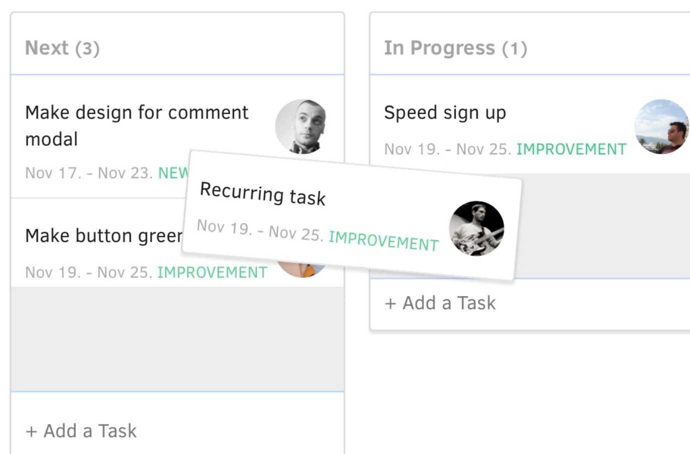
✓ Feature and bug dependencies identified

✓ Performance criteria identified (if needed)

✓ Demos scheduled

✓ Start and end dates set by team leader

This is the moment when a task moves to the “In progress” column. When a feature is kicked off, the task moves into the “In Progress” column.



Teamwork on display

Our developers never work alone — it’s always a team effort, and it’s by far the most efficient way to release new features. Before teams were introduced, a (junior) developer would get stuck with a problem and might have gone round in circles for days (without anyone realizing it). Or, if they weren’t the lone-ranger type, they’d constantly be distracting colleagues and managers.

On the other hand, with teams, we mix people with different skill sets and levels of experience. This means that everyone is assigned a set of tasks appropriate to their level. Plus, seniors get the benefit of learning how to manage and coach junior teammates — and juniors get to ask for help.

Because it’s a team effort and everyone is working towards the same goal, questions aren’t regarded as distractions, and the team can tackle any issue much more efficiently. The team becomes a self-organizing entity, splitting

work into phases (or sprints) and presenting their work during demos.

Show And Tell

According to the schedule, the team will demo for the product owner. The purpose of the demos is to show how well a feature is performing in its current state and where it needs more polish. It's a reality check that doesn't allow for excuses like, "It just needs a few finishing touches" (touches that would take another month). Also, if things start to take a wrong turn, product owners get to react quickly.

Weekly Meetings

We started off with regular short daily meetings attended by all developers. Each developer would briefly describe what they were working on, their problems, their blockers and whether they needed help. It soon became obvious that these meetings needed to be more focused and to provide tangible results. So, we switched to having meetings with individual teams about once a week. This is how the product owners and project manager are kept in the loop and all potential problems are dealt with on the spot.

Putting it to the test

The code is written, the demos have been successful, everything seems to be wrapping up nicely... until you release the feature and see that the app crashes. That's why every feature we make goes through quality assurance (Q/A). Always. Our tester meticulously goes through every possible scenario, checking all options and running tests in different environments. A feature rarely passes Q/A on the first go.

To increase productivity, we used to let developers start on new features during this phase, but that just resulted in a lot of delayed, half-finished features. So, now the development team keeps busy by working on small maintenance tasks while their feature is being reviewed. If Q/A finds a problem, the developers immediately fix it and resubmit. The process is repeated until the feature passes Q/A and moves on to code review.

This is when a senior developer makes sure the code is written according to

our standards. One final step before the release is writing the documentation — you don't want to get swamped by support emails after releasing a feature that no one knows how to use. Our copywriters also update the release notes and write marketing materials, such as email announcements and blog posts.

Here's our definition of "done":

✓ Auto-tests written

✓ Q/A done and all resulting tasks completed

✓ Code review done and code merged to master

✓ Release notes updated

✓ Feature and bug dependencies identified

The task has reached the final stage, called "Release Q."

Release day

When choosing a day for new releases, we immediately decided against Friday and Monday. Friday is not good because any potential issues wouldn't get resolved until Monday; plus, the support team was already pretty busy then. Monday is not the best time because any major update requires preparation, which would have to be done on the weekend. It's always good to leave a day for final touch-ups. This narrows down the options to three days — and we went with Tuesday. Here's why:

- We have Monday to review the release and add finishing touches before deploying.
- If there are any untranslated phrases (our web app is available in seven languages), we have enough time to finish the translation.
- The marketing team has plenty of time to send out newsletters and announcements via social media.
- We are available to provide upgrade support quickly and efficiently for

the rest of the week.

- If a deadline has passed for whatever reason, we still have Wednesday or Thursday to complete the work.

Free Activity Day

The day after a major release is a free day for the team. This is when they learn new skills or do anything work-related that they find interesting. Even though everyone's eager to know which feature we'll be kicking off the following day, the team doesn't discuss that just yet. Instead, they'll relax and maybe watch a presentation about the history of Erlang, or finish reading that article about why PSR-7 and middleware are important to modern PHP development, or have their own retrospective discussion. It's a well-deserved day to recharge their battery and celebrate a job well done.

Bug Hunt Day

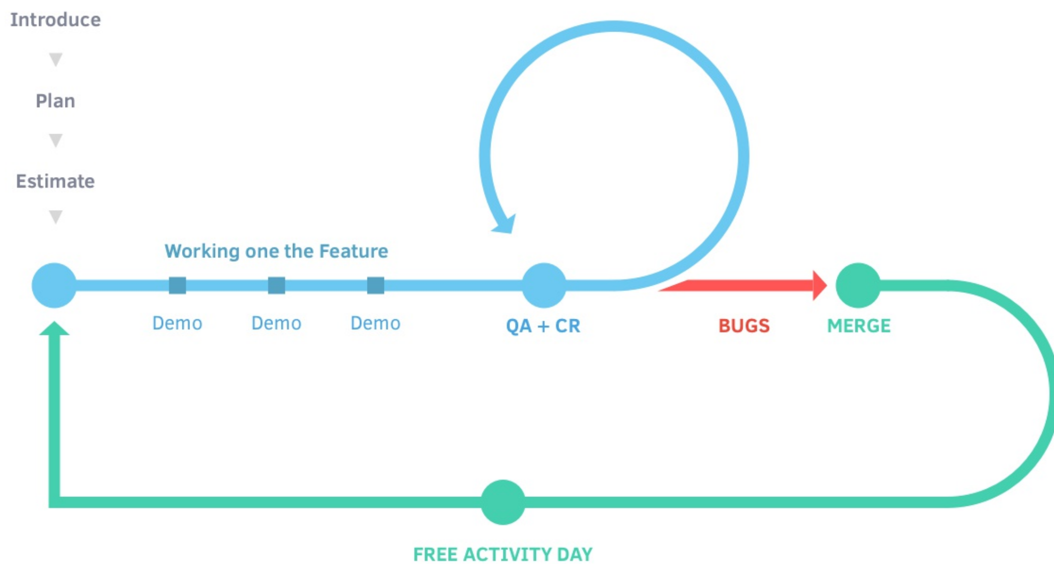
Apart from developing major new features, there is always work to be done on existing ones. Whether it's a bug fix, a design update or a small change in functionality, the team needs to take care of it in a reasonable time.

Clearing the backlog of bugs is much faster when a day is dedicated just to that.

This is why we have bug-hunting days at least once a month. It's when all developers stop working on their main projects and turn their attention to maintenance. This focused effort has proven to be a great success. When everyone works solely on bugs on the same day and is available to help each other, the team solves a huge number of issues.

The result

Releasing a big feature now takes about three weeks on average, from the kickoff to development, testing, code review, documentation and final release. A feature of equivalent complexity used to take us 45 days. From our perspective, that's a 100% increase in productivity. We accomplished it with the same resources and people, the only difference being an improved workflow.



To us, Agile is not a destination. We see it as thinking, learning, and constant improvement that yields measurable results (measured in how well we are meeting customer demands). That's what it's all about.

Dave Thomas says that agility is what matters, and that it's a really simple framework (no book or coach needed):

1. Find out where you are,
2. Take a small step towards your goal,
3. Adjust your understanding based on what you learned,
4. Repeat.

Some may not like what he is saying, and may prefer prescribed management and development processes more than the one that's in constant motion, but saying that such a process (and thinking that got the team to devise it) is not agile would be wrong. The process itself may be 'waterfally', but that is how flow works (it flows through work-cells) and it's great when you are building a known solution to a known problem (Active Collab turns 10 this year, so we know our customers, what they are looking

for, and how to built it).

If you start from the perspective that all Waterfall projects are Deathmarches and that Waterfall is bad would seriously limit the team's ability to think how to improve and better do what they are hired to do. They should be thinking about the customer, value stream, bottlenecks, not getting burned out etc, not what their process is called. Maybe this is "just a bunch of small Waterfalls", but that alone is hugely different from a Deathmarch, from our customer's, team's and business perspective.

Our workflow here is just a snapshot of our process at the moment when this was written. There are many lean and agile concepts that are built into the process, like:

- One-piece flow (capacity set to 1, no set sprint length, team works exclusively on one feature, even when it hurts)
- Need to balance the team (so we don't get to the hurt part in #1),
- Pull (customers say what is going to be worked on, not Marketing or Management)
- Continuous integration and dogfooding (we start using a feature as soon as it reaches level of usefulness for our team, usually weeks before it is well rounded to be presented to customers)
- Waste reduction (part of the improvement process: take a bit of time to think and discover waste, see if it can be removed, remove or reduce it, learn)
- Focus on lead time and throughput (who cares how it is called, if it delivers customer demanded results twice as fast and gives you the option to improve even further)
- Bottlenecks (find them, understand them, see how they affect the value added work, address when needed)

So, if we have one takeaway, it's this:

Nurturing a company culture that eliminates fear of change will make your team better at what it does.

Whether you call it Scrum, Kanban or Lean doesn’t matter, as long as it helps your company grow. Experimentation and innovation lie at the heart of any agile approach. Don’t be afraid to test different solutions, measure results and, based on the results, modify existing practices. Good results will follow.

Case Study: Agile Reporting

When Active Collab was starting out, I - Ilija Studen, the cofounder of Active Collab - wore a lot of hats. My areas of responsibility were backend development, customer support, blogging, community moderation, and many other smaller activities. With such a hands-on approach, it was hard to let go of some of these things and let others do them instead. But it had to happen, because it became obvious that my most impactful work no longer took place in the trenches (even though I still love the trenches).

First step: surround yourself with great people you enjoy working with. How to find such people, direct them (and, even better agree on a direction) and keep them, is a massive topic that we'll get into in a series of blog posts later this year. For now, let`s just assume that you managed to find them and that they turned up for work next day. Can you really let go, get out of their way, and leave them to do their best work?

In areas such as development, marketing, and customer care, I found a method that worked really well: answer-oriented reports that are quick to compile. They can be set up in Active Collab as recurring tasks; colleagues prepare them, and send them over to my partner and me. Apart from that, I'm mostly hands off and intervene only when I get asked for help, or notice something strange.

Answers rather than raw data

Nobody ever told me that a report can be anything other than a massive pile of data that you're expected to draw conclusions from by yourself. I always pictured reports as huge spreadsheets, full of data points and charts, formulas and rainbow colors everywhere. The fact that I've never worked for an organization that was heavy on reporting helped this Hollywood-style image stick in my head for years.

Instead of raw data, reports should be focused on conclusions and opinions.

While some of the reports that I get really are spreadsheets with charts, formulas and rainbow colors, the ones I like best are the simple documents that mix data points with bullet lists and paragraphs of text that offer conclusions, opinions and suggestions.

In such reports, the focus is shifted away from raw data, toward what we get out of the raw data. I hate when someone just sends me a link to an article, or an Excel file, and expects me to divert from what I was doing to focus on something else, and then draw my own conclusions. Why does that very same work need to be done twice – first by the person who read it through and thought it might be of interest to me, and then again by yours truly? When a new colleague does that (old ones know me better than that), they get a simple reply: “And?” or “What am I looking at here?”, but only if I’m in a good mood.

I love when people send me key data points alongside their own conclusions and opinions…and the raw data that led them there is attached as an extra file (that I may, or may not, open for further study). The same goes for articles that cover the topic, or formulas used for calculations. They’re great additions to the report, but it shouldn’t be necessary for me to read or need them to get to the answers that report’s trying to communicate.

Design answer-oriented reports

In order to get a report that doesn’t focus solely on data, but captures the result of data analysis in a simple-to-digest form, you need to build it around anticipated, possible answers, not the data itself. Here are some techniques I used when I was designing a recent report:

Which questions do I want to have answered by this report?
Reports should give you answers, not mystify you and force you to figure out what the data’s all about. In order to get to these answers, start with questions - the crucial questions that can be answered by studying data that your systems and people are collecting, like: “How engaging was the content that we published this month?” or “What was the team’s velocity during this sprint?” or “What’s the churn rate during the month, and what are

the main churn causes?”

Reports should give you answers, not leave you even more puzzled then you were before.

Structure things in such a way that you get the data (“Churn rate during May was 2.5%, with 15 accounts, with $1500 MRR lost”) but so that you also get opinions and conclusions from your colleagues (“Upon studying the data, we found that main cause of churn remains X, yet we noticed that people are also stating Y as a reason why they closed their accounts. Our suggestion for how to approach this problem is to do Z.”).

How often do I need to check a report?

Some reports should be checked weekly, some monthly, some quarterly, and some are yearly retrospectives. Start with the least frequent option that you feel comfortable with. Later on, you can experiment with more, or less frequent, reporting cycles to see what works best for you and your team.

Am I the only person who can prepare this report, or can I legate it to someone else?

The answer should always be that you can delegate it to someone else. If not, reshuffle things so that someone prepares the report for you.

How long will it take for this report to be completed?

Some reports take more time than others, but I always try to shoot for a report to be completed in under 30 minutes. If your colleague needs a day or two to prepare a report for you, it’s pulling them from their actual work way too much. Answers and conclusions are what’s important, so focus on that. Also, consider as much automation for data collecting and number crunching as possible. There are some great tools out there that can help your with report automation.

For example, you can use tools like Zapier to listen to particular events from all sorts of apps and record these events as line items in Google Docs spreadsheets. Then you can pull the important data points from that spreadsheet, instead of going to different tools and compiling the data by

yourself.

Are we exceeding A3?

Toyota used to insist that all reports fit a single A3 page. The reason was simple and practical – A3 was the largest format that could easily be faxed between Toyota employees. Apart from that physical limitation, the folks at Toyota also love brief and to-the-point reports. That's why the A3 directive is a good rule of thumb when figuring out how much info to fit into a single report. While we don't write or print our reports, I try to lay down the data in my head on A3 paper, and cut the report if it can't "fit."

Should I store the reports for later use?

The reason I love wiki as a way to publish reports is because it helps you build up your company's knowledge base. With these reports, the conclusions presented in them, and the discussions they sparked, can help you induct new colleagues in the future and get to the "thinking behind," so you can disseminate you got to a process, not just the process itself.

When you're designing your new report, consider whether it's going to be worth keeping, or if you'll delete it once you've studied it. Some things can be forgotten because they hold no lasting value (like the weekly Twitter audience analysis report), but some will prove to be a valuable asset as they pile up and build your knowledge base (like development sprint, or marketing experimentation reports).

Implementing automated reporting

Now that you've got your list of questions that you want your team to answer on a weekly or monthly basis, it's time get things geared up so you actually get them answered. Consult the person who'll be preparing the report, and explain the questions and thinking behind the question. They'll probably have comments and suggestions, and you should consider them and improve the report.

Know that it's human nature to go with the easiest possible route, so never lose sight of your main objective (why you wanted this report in the first

place) when discussing technical implementation. Don't just discard a question because the data's hard to collect. Figure out a way to make it easier, instead. Consider automation, adopting new products that can produce the numbers that you need. Just don't give up at the first hurdle.

I see a report as a way to extend our processes with a mandatory reflection point on an important topic. As such, it brings value to the team preparing and presenting the report, to the company itself (as a knowledge building tool), and to you, as the business owner or manager. Take that into consideration when you decide how the report will be prepared. If it's hard for you to take it in, it's not going to be read (or watched). If it's hard for team to produce it in a timely fashion, it's going to cost you more than the value that you'll get out of it. Avoid both traps, and go for something that's reasonably easy to prepare, yet you know you'll give it your full attention when it arrives in your inbox.

I see reports as a way to get insights, keep things on track and get myself out of people's way.

You've noticed by now that I like wiki as a way to get reports. That's my personal preference for some types of reports. It goes hand-in-hand with my goal to build Active Collab into a company that collects and nurtures our collective knowledge, instead of letting it disappear as time goes by. Some reports that I get are Google and Excel spreadsheets or PDFs attached to tasks that Active Collab automatically creates and assigns to the team. That works really well for reports that I don't consider to be "knowledge building."

You may prefer something completely different, and you should go with that. In the beginning, fancy tools are of little relevance, and you should go with something that's convenient and easy.

When you pick the tooling, try to create one report by yourself (with assistance of the team, of course). It's okay if there's a lot of manual work in the beginning. By preparing a couple of reports manually, you'll get a feel for how much work goes into them, and what the general experience is. As you get a feel for the process, try to improve on it by automating as much as possible. Depending on the tools you've chosen, different options are

available: document templates, form builders, data aggregation tools....

If possible, keep in mind that 30-minute target. Some data points and records may already be prepared (for example, incident reports, web traffic analytics etc.) Putting these elements together, with brief conclusions, should take no more than a half hour. Elaborating on conclusions, and adding extra bits of information to present an opinion or suggest a course of action may take as much time as needed after that. That's the magic, so don't put a limit on it!

Now that your team knows what kind of report they'll be preparing for you, and why, you're ready to delegate it. Create a recurring task that assigns the report preparation to a colleague, sets the reporting frequency...and sit back and wait for the first report to roll in on the scheduled date.

Sprint report questions

We have a section in our company wiki called Control Tower, where project managers list features, releases, and sprints that are under development. Each big feature (that takes more than a month ti develop) is a separate

project. We develop them in biweekly sprints (like in Scrum).

After each sprint, the project manager creates a report that answers the following questions:

What was the goal of the sprint?

Before the project kick-off, we set clear targets for each sprint's goal. Not all projects lend themselves the two-week timeline, but they all have established targets. Teams need clear goals, expected outcomes, and timeframes so you can measure what you've achieved and test your estimating skills.

Who managed the team and who did the work?

We record who manages the team and who is responsible for what parts of the work. This helps us with performance reviews of the both project managers and the developers. It's also nice to check the wiki from time-to-time and remind yourself how you contributed to the company’s growth.

How long was the sprint, in work days?

We restrict sprints to 10-12 days, plus two extra days for the kickoff and retrospective. Days are recorded as actual dates, for future reference.

How many tasks did we plan to deliver? How many complexity points did they have during the estimate?

During the kickoff, the team estimates how complex each task is in the sprint. These are then used to gauge whether the workload can be actually achieved in those two weeks. A project manager monitors the team’s progress. If there's too much work, they remove some tasks from the sprint and put them for the next sprint.

How many tasks were finished? What's their complexity score?

We measure velocity (how much work we finish) by adding up the complexity points of each finished task. Only tasks that fully meet our definition of done are taken into the account. Knowing your velocity is important so you can better plan in the future. It's also an indicator of how

well we did our job.

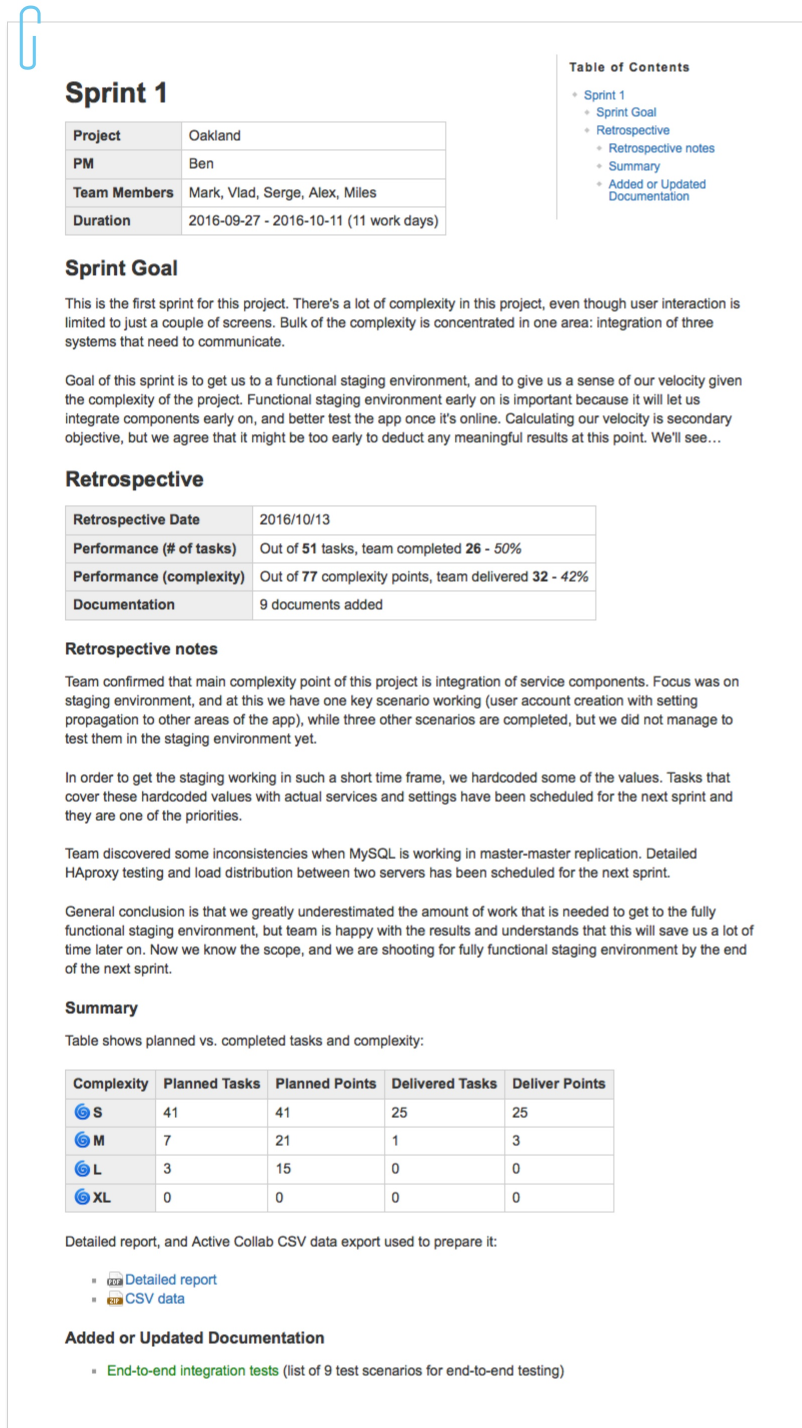
What were the conclusions from retrospective?

We always hold a retrospective after each sprint and note the conclusions in the team's knowledge base. This way, we capture learned lessons for future sprints and projects. This is especially helpful when someone new joins the team. We give them to read every sprint report before starting to work so they learn from our experience.

Which wiki articles were written or updated during the sprint's urse?
With the pressure to deliver working software, documentation is often an afterthought. But, it's extremely important because it helps us maintain projects, especially the old ones. So, writing development documentation is a mandatory part of the process. All new and updated documents are listed in the sprint report.

Real-world example of agile report

Here's one of our sprint reports (numbers and descriptions have been tweaked a bit to better illustrate the point):



Sprint 1

Table of Contents

- Sprint 1
  - Sprint Goal
  - Retrospective
    - Retrospective notes
    - Summary
    - Added or Updated Documentation

Project	Oakland
PM	Ben
Team Members	Mark, Vlad, Serge, Alex, Miles
Duration	2016-09-27 - 2016-10-11 (11 work days)

Sprint Goal

This is the first sprint for this project. There's a lot of complexity in this project, even though user interaction is limited to just a couple of screens. Bulk of the complexity is concentrated in one area: integration of three systems that need to communicate.

Goal of this sprint is to get us to a functional staging environment, and to give us a sense of our velocity given the complexity of the project. Functional staging environment early on is important because it will let us integrate components early on, and better test the app once it's online. Calculating our velocity is secondary objective, but we agree that it might be too early to deduct any meaningful results at this point. We'll see…

Retrospective

Retrospective Date	2016/10/13
Performance (# of tasks)	Out of **51** tasks, team completed **26** - *50%*
Performance (complexity)	Out of **77** complexity points, team delivered **32** - *42%*
Documentation	9 documents added

Retrospective notes

Team confirmed that main complexity point of this project is integration of service components. Focus was on staging environment, and at this we have one key scenario working (user account creation with setting propagation to other areas of the app), while three other scenarios are completed, but we did not manage to test them in the staging environment yet.

In order to get the staging working in such a short time frame, we hardcoded some of the values. Tasks that cover these hardcoded values with actual services and settings have been scheduled for the next sprint and they are one of the priorities.

Team discovered some inconsistencies when MySQL is working in master-master replication. Detailed HAproxy testing and load distribution between two servers has been scheduled for the next sprint.

General conclusion is that we greatly underestimated the amount of work that is needed to get to the fully functional staging environment, but team is happy with the results and understands that this will save us a lot of time later on. Now we know the scope, and we are shooting for fully functional staging environment by the end of the next sprint.

Summary

Table shows planned vs. completed tasks and complexity:

Complexity	Planned Tasks	Planned Points	Delivered Tasks	Deliver Points
S	41	41	25	25
M	7	21	1	3
L	3	15	0	0
XL	0	0	0	0

Detailed report, and Active Collab CSV data export used to prepare it:

- Detailed report
- CSV data

Added or Updated Documentation

- End-to-end integration tests (list of 9 test scenarios for end-to-end testing)

As you can see, a report is just a regular wiki page with a couple of tables, lists, and some conclusions gathered by the team immediately after the sprint, while the memory is fresh.

But the real magic happens later on, when these report pile up. Then you can see how well your team performs over time, when you underestimate the work, etc. These reports are also very helpful when a new developer joins the team so they can learn from our experience.

Complexity Points / Workdays
(eg. 46 points in 13 work days)

Project	Sprint	Sprint goal	Timeframe	Output	Resp.
Shepherd	Sprint 1	Create trial in staging	17. Aug - 2. Sep	86/13	Ben
Shepherd	Sprint 2	Account lifecycle; new payment method	6. Sep - 22. Sep	81/12	Ben
Shepherd	Sprint 3	Staging stabilization; test procedures	27. Sep - 11. Oct	62/11	Ben
Shepherd	Sprint 4	Authentication - IdP and SP	14. Oct - 28. Oct	.../...	Ben
Shepherd	Sprint 5	Buy, upgrade, downgrade, switch plan	.../...	...	...
Shepherd	Sprint 6	Data migration, licensing	.../...	...	...
Shepherd	Sprint 7	Wrap up, test, get ready for deploy	.../...	...	...

More **reports** = More **insight**

Agile reporting is great when you do it right. It gives everyone a good point of reference from the past, helps us plan future growth and make better daily decisions, and prepares us on how to deal with challenging situations.

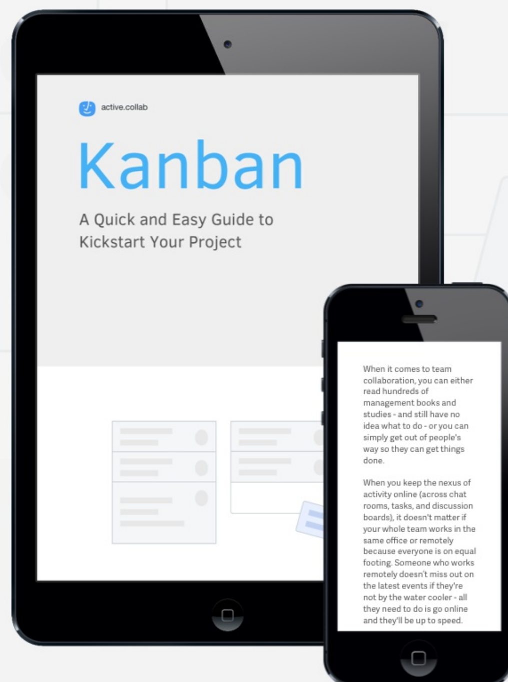
Working on your business, instead of in it is something that many of us continue to struggle with. Don't let that worry you – it's a learning process for everyone. Follow the recipe above, and enjoy the insights you'll get from business reports that are actually important to you.

What Next

This guide covered the basics of agile project management. To learn more about other aspects of project management and running a business, be sure to check out our other books.

Kanban: A Quick and Easy Guide to Kickstart Your Project

This book introduces Kanban and key principles of agile project management designed to improve your productivity. The book is very short and is geared towards beginners. The book will help you learn how to organize projects and how to introduce a simple and reliable process so you're more productive.



Download

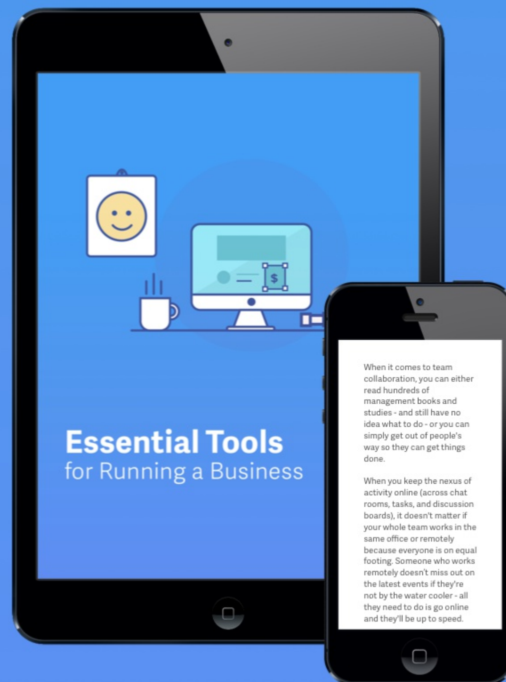
The Complete Guide to Managing Digital Projects

This book dives deep into project management. It covers everything from client collaboration and project management to invoicing and time tracking.
The book will teach you everything you need to know to successfully manage digital projects, get paid, and make your clients happy. Unlike the Kanban ebook, this will take you much longer to read but it's still very easy to understand.

Download

Essential Tools for Running a Business

Every growing business needs tools. This book lists every tools that helped us grow our company from 3 to 30 people (and beyond). We share behind-the-scenes insight, how we use every app, and how each app can help YOU become more productive. Every tool is illustrated with screenshots so you can see how it works.



Download

GROWTH: Everything You Need to Know Before You Can Grow Your Business

Growth is a double-edged sword. It's good if you are prepared and know what you are getting into, but terrible if you aren't ready. The book covers everything you need to know to avoid mistakes business owners commonly make when growing their small business.

Download

Active Collab is a powerful, yet simple project management software. It helps your team stay organized when you outgrow email and offers a one-stop solution for all your business needs.

Active Collab gives you an overview of your team's activity across projects and bring clients on board to collaborate more closely. WIth it, you can delegate tasks to your team, keep information in one place, estimate and track time, and issue invoices.

For more than 10 years, over 200.000 people have used Active Collab, ranging from small businesses to Fortune 500 members, universities and government institutions.

activecollab.com