ZenHub

GitHub Project Management

Building better software and stronger teams

By Matt Butler and Paige Paquette

GitHub Project Management

What we know about building better software and stronger teams.

ZenHub helps software teams do exactly this. Find out how.

To read more, visit our blog where we cover productivity, project management, and exceptional teams.

Thanks to Bryce Bladon for helping put together this book and adding all the funny bits. Thanks also to the experts who provided feedback and thoughtful contributions.

Design by Stefano Tirloni

ISBN 978-0-9952747-0-9

Index

Intro

Much has been said about “the right way” to execute a software project.

Project managers and business types love to debate the merits of one approach over another, from Waterfall development to modern agile methodologies like Scrum and Kanban. How many meetings should we have? Should there be a pre-project project? What should a Scrum master do? A product owner?

Software developers, on the other hand, typically just want to get back to the damn code.

At ZenHub we spend a *lot* of time thinking about the best way to run a software project. Believe us, we’ve tried everything.

Those who use our products are familiar with the *tools* we believe help teams improve, but we spend considerably less time talking about the thing that connects it all – that is, the **best practices, values, and workflows** that help software teams do their best possible work.

In this book, we answer our two most common project management questions:

How do you run projects? Why do you run them that way?

We don’t spend much time debating one methodology over another. Our team generally stays away from capital-letter methodologies –

that is, Scrum, Kanban, or even “Agile” – and this book does, too.

Our own philosophy for product development is closer to post-agilism (small “a”), whose manifesto is pleasingly simple:

If it works, do it. If it doesn’t, don’t.

The purpose of this book is to arm you with strategies that could work for your team.

We use GitHub and ZenHub as our frame of reference, but the principles can be applied to whatever you’re using.

Who is this for?

This book is for software developers.

Specifically, software developers who want to ship better software in less time.

It isn’t about splitting hairs on Project Management Theory; there are tons of great resources out there if you want them. The purpose of every chapter is simple: to help teams ship better software. In less time. (Period.)

These strategies apply to enterprise software and open source projects in equal measure.

This book is for managers.

Whatever it says on your business card, you’re the person whose job it is to remove roadblocks from the path of your software team, at any

cost. If it's your mission to help engineers be as productive, collaborative, and *happy* as they possibly can be, this book will help you do that.

This book is for anyone involved in a software project.

Teams are becoming increasingly multi-disciplinary, but we have so far to go. We need more tools and initiatives that break down the walls between developers and "everybody else."

We believe the best way to do that is to centralize everyone in a single place, to break down silos of information, and to create a single source of truth. If you're involved in a technical project, this book will help you become a better software team member.

What we'll cover

This book covers the lifecycle of a software project from end to end.

We'll start by explaining why GitHub (and version control systems in general) are actually the perfect place to manage software projects. You'll learn the basics of setting up your GitHub project for success, including a sane label system and a task board that helps you spot bottlenecks early.

You'll learn how to manage and execute a product backlog that results in something your users actually want to buy, and understand how to use epics, Milestones, and GitHub data to bring speed and focus to your project.

We'll give you strategies to estimate software tasks more accurately, and show you how to use those estimations to deeply understand the

way your team works together – ultimately reducing guesswork and miscommunication.

Finally, we'll take you through exactly what to do before, during, and after a sprint to make sure you're making continuous improvement a habit. Along the way, we'll provide detailed examples from our own workflow to give you a better idea how it plays out in the real world.

And we want to hear from you. I invite you to tweet us with your own tips and experiences, or email us directly at founders@zenhub.com. Every team has different needs, and we'd love to hear what works for yours. Let's learn from each other.

Enjoy the book.

– Matt Butler

Co-founder and CEO, ZenHub

CHAPTER 1

Doing agile inside GitHub

What is Agile?

Agile development – or simply agile – is an iterative approach to software development that emphasizes flexibility, interactivity, and a high level of transparency.

Agile projects involve the frequent release of useable code (revenue), continuous testing (quality), and acceptance that whatever you think you know now, it'll change (reality!). With a few tweaks and some basic understanding of best practices, you can turn GitHub into a powerful agile platform...and reap the sweet, sweet benefits of the GitHub data your team is constantly creating.

GitHub is actually perfect for agile project management

GitHub reinvented what it means to collaborate on code. They championed a new era of open source development, which naturally transitioned into a profitable business driven from the ground up by developers who love the platform.

If something has code, there's a good chance it was developed or refined on GitHub. It's where the world's top software teams write, collaborate on, and ship amazing products.

The issue of focus: How most project management tools fail developers

As GitHub took over the development world, project management companies rushed to integrate with it as an attempt to bridge man-

agement and development teams. But every solution shared the same problem.

They all “live” outside GitHub, forcing developers to jump from tool to tool to update tasks, tickets, and reports. The result is “context switching”, and it’s much more costly than we expect.

People, as a rule, are terrible multitaskers. A study by the Journal of Experimental Psychology found that people who multitask see a 40% drop in productivity. When interrupted from a task, it takes 20-30 minutes to refocus. Switching between low-performance tasks – say, texting a friend while watching TV – isn’t too difficult. Getting back to where you were before (zoning out in front of Netflix) takes nearly no time at all.

But when tasks are complicated, the cost of context switching sky-rockets. Consider a developer in the middle of coding a new feature. Here, a five-minute interruption takes much longer than five minutes.

Steve Fenton puts it this way:

> **“If a developer is interrupted with a question, it could put their work back by hours for each interruption. This is because of the amount of stuff they are holding in their immediate memory while they are working on a feature. They will have parsed lots of code in the area they will be working on and will have projected their changes in their mind before they start coding and the interruption will impact their focus on this information. Once interrupted the developer may decide to batch other interruptions like email into the break – but this elongates the**

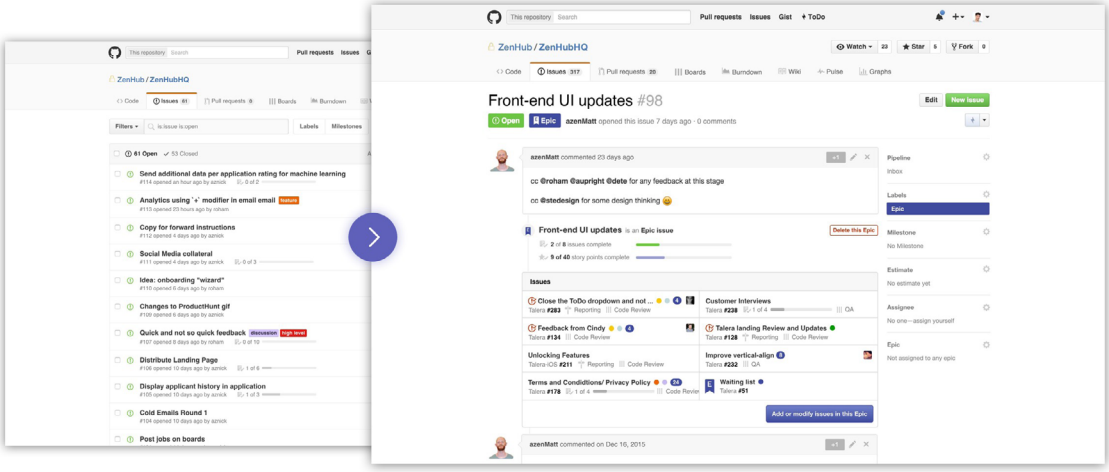
gap and makes it more likely that the information will be squirreled away into harder to access places."

In short: context switching bad. Make developer :(.

As a team lead, your role should be to shield your development team from distraction away from the code. Your business will be better for it – trust us.

"It's important to respect that the type of work we expect devs to do requires some degree of recurring isolation to do it well," says Christopher Hawkins, project management blogger and founder of Cogeian Systems. "A good manager runs interference against anything that might compromise developer focus, while actively soliciting any resources the developer might need."

How ZenHub Helps



There's a reason we built ZenHub, and it relates to why we wrote this book.

It’s not enough to instruct, hope, or pray engineers stay focused. As software-driven companies, everything – down to the tools we use – should reinforce our commitment to the software first. This is why we created a tool that brings the project management process inside GitHub. Not *close to*, not *“integrated with”*, but *inside*.

Centralizing your team in one system carries other benefits, like:

More accurate metrics

- Third-party tools result in silos of information. By creating a single source of truth, your data is always up to date.

Focus on product, not process

- Project managers shouldn’t spend their day reminding coworkers to update tickets. When everything is centralized, project managers spend more time managing the project, and less time worrying about the people building it.

Collaboration as a habit

- GitHub issues were built for collaboration. Collaboration not only helps reduce error and technical debt, it actually keeps teams happier. In one study, enterprise companies that identified collaboration as the top priority significantly improved employee satisfaction, customer retention, and productivity (Aberdeen Group, 2013).

More than issues: Setting up an agile process in GitHub

Like a message board for your ideas, issues are GitHub's task-tracking system, used to log bugs and scope out new features. Issues provide a place to talk about the amazing software you're building.

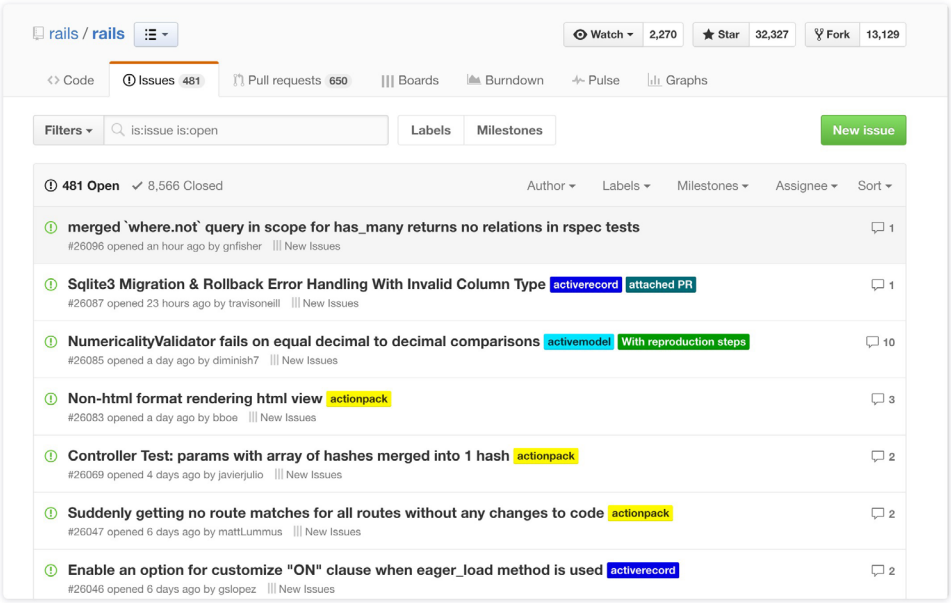


Image via GitHub.com

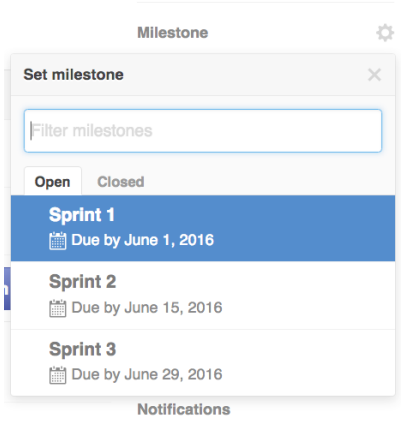
But in their current state – organized in a list – it's difficult to glean important information about issues in a glance. You can't easily understand their progress or their priority.

ZenHub's extension adds a crucial layer of prioritization and planning to GitHub issues.

Mapping Agile concepts into GitHub

Sprint → Milestone

In Scrum, sprints are a fixed length of time during which an agreed-upon chunk of work is completed and ready to be shipped. Sprints, or “iterations”, are mirrored in GitHub with Milestones. Simply set a start and end date (typically two or four weeks), and add user stories to begin sprinting.

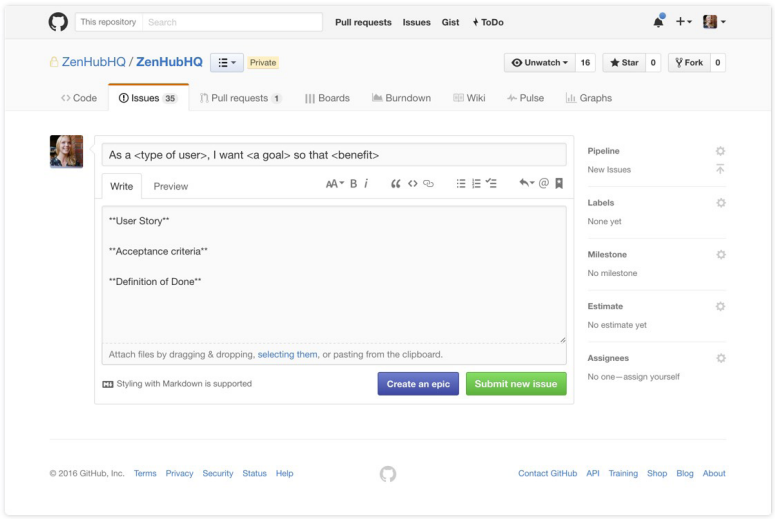


Teams should decide together how much work they commit to tackling every Milestone. Things get easier when you have some historical data – in the form of burndown or velocity charts – to back up your assumptions. You’ll learn how to do this later on.

User Stories → GitHub Issues

User stories are high-level feature descriptions that help define benefit from a customer’s perspective. They’re often written in this format, which is intended to keep things simple and focused on business value:

As a <user type>, I want <a goal> so that <benefit>.



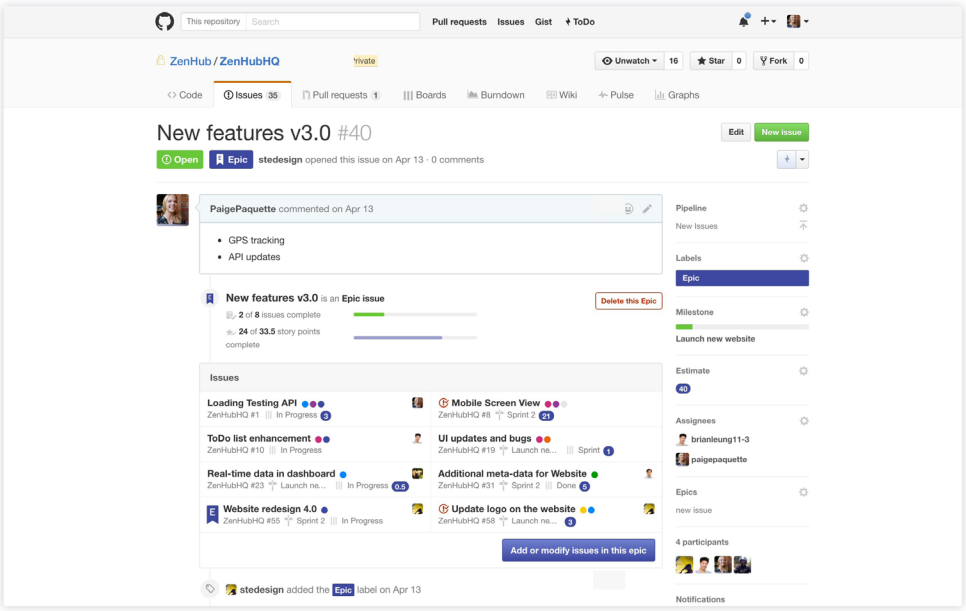
If you're adhering to this structure, make it your issue's title. You can also set up GitHub issue templates to add detail or acceptance criteria when the time comes.

Epics → Epics

No need to map this one: ZenHub provides epics inside GitHub. Epics help teams plan and collaborate on product backlogs; they're essentially just big user stories (for example, "Dashboard Redesign" would be an Epic, while "Change CTA colour" would be an issue within that Epic.) Using ZenHub, you can map out big chunk of work using Epics, then add related issues to flesh it out.

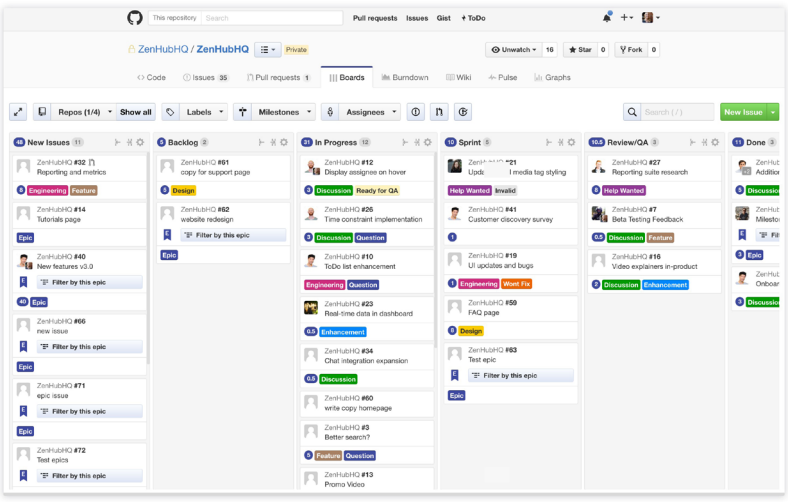
Epics have flexible scope, so you can add, edit, and remove issues as needed. Once you've set up an epic, you can track it alongside your other work in your task boards. Or, filter by epic to track only those issues.

In contrast to Milestones, Epics are *not necessarily related by time*. It can take several Milestones to complete an Epic, and that's okay!



Product backlog → Open issues without a Milestone

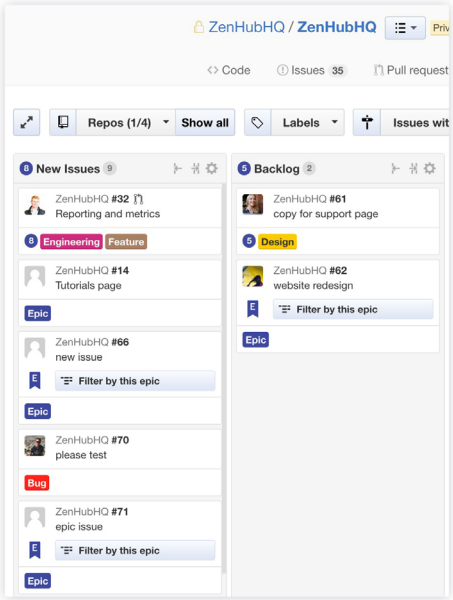
Your product backlog (sometimes referred to as a master story list) includes, well, pretty much everything. Typically ordered vertically by each issue's business value, ours is a collection of user stories, half-baked feature ideas, things we *might* do in the future, technical work, and anything else. The point is to make a habit of getting stuff out of your head, and into GitHub. Be honest with yourself, though: if it's *never* going to get done, it's best to close the issue.



Sprint backlog → Issues with a Milestone

Your sprint backlog contains the work your team has committed to tackling in a given Milestone. Issues here should be estimated, include detail, and have a Milestone attached. They should be vertically arranged by priority in this pipeline.

As we'll discuss in the product backlog chapter, you can use the **Icebox** pipeline to "freeze" stories that aren't a priority, and the **Backlog** pipeline to prioritize issues for multiple Milestones. When a team member is ready for a new task, they simply filter the board by Milestone, then drag issues from the Backlog to In Progress pipelines.



Note: Beware of scope creep. Once a sprint begins, your issues should remain fixed.

CHAPTER 2

Smarter labels, better task boards

A task board is an at-a-glance “information broadcaster” about your software projects. It’s the easiest way for teams to break down barriers that divide people working on a project, powering communication, efficiency, and a higher-quality product. In this chapter, we’ll explain how to combine custom labels and pipelines to create your perfect project dashboard.

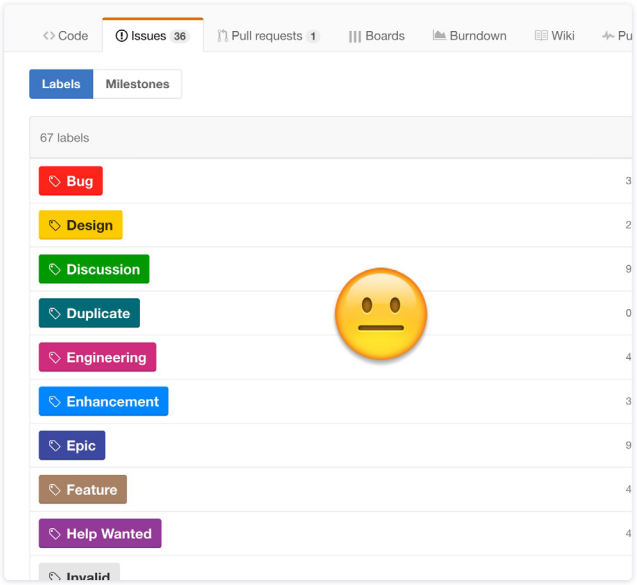
First things first

A successful project foundation starts with a good task board, and a good task board starts with repos, understood pipelines, and a sane labelling system.

- **The task board** (or “board”) is how you view, prioritize, and manage your product backlog.
- **Repositories** (“repos”) are how projects are divided in GitHub. GitHub now provides unlimited private repositories on paid plans. Since ZenHub allows you to connect repositories together, don’t be shy – divide your projects into a generous number of repos to help things stay organized.
- **Pipelines** are columns on your board that organize your issues (tasks). Your issues move between pipelines from left to right as they progress through the development cycle. We’ll discuss customizing your pipelines later, but ZenHub’s default structure works well as a starting point.
- **Labels** are the tagging system used to convey more information about each issue. Naturally, you can filter your boards by label.

A note on labels

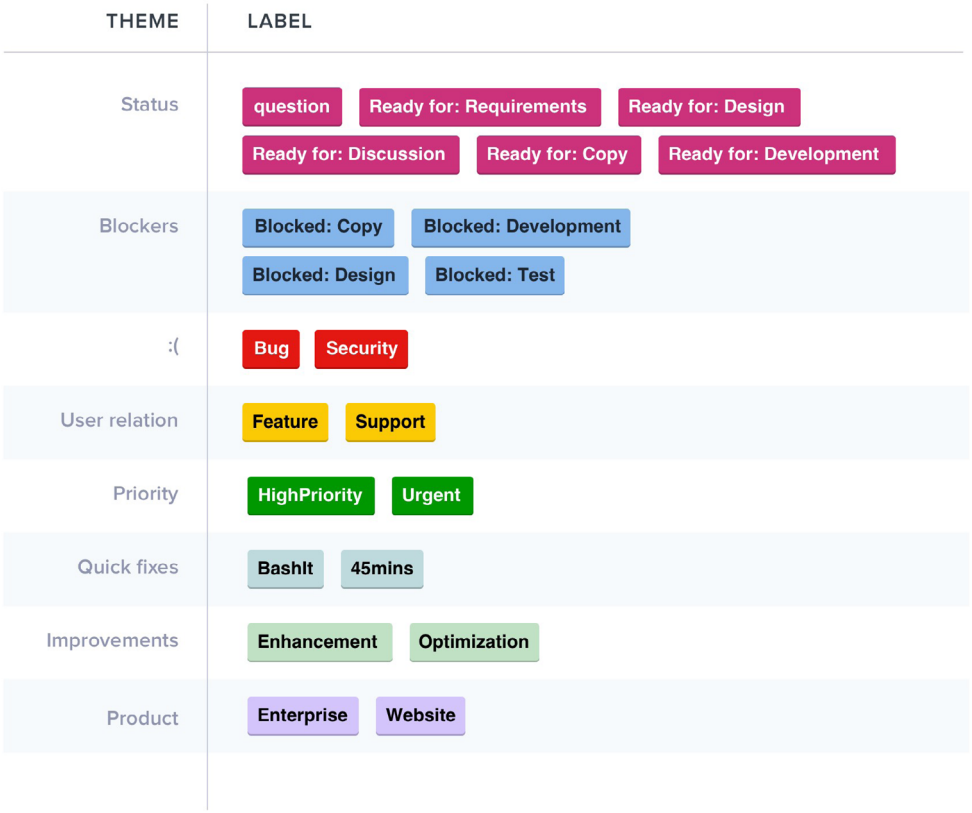
Don’t settle for GitHub’s seven default labels. Take a minute to set up a label style guide that communicates more than the issue type.



Our team set up custom labels that communicate **status, blockers, complexity (for example, we mark quick fixes with the label “Bash It”), and other factors**. Together, they communicate a lot in a single glance. Check out our example style guide on the next page.

Our label style guide

Labels don’t just add a pop of colour; if you use them consistently, you’ll never have to sift through your issues on the board. (For more ideas around labelling best practices, check out this post.)



Your new home at work

Did you know the brain processes visual information 60,000 times faster than text?

Kanban is a methodology that capitalizes on this fact to create an efficient and improvement-orientated system.

Basically, Kanban boards create a picture of project work and processes. By visualizing work this way, you can create focus, establish flow, and continuously improve.

Before tools like ZenHub, task boards were created using post-its and stickers. **The *one* advantage a physical board has over digital? Constant visibility to keep it top of mind. Keep your board displayed on a big monitor somewhere obvious, and bookmark it as**

your go-to workspace.

Using pipelines like a boss

ZenHub boards come pre-set with six pipelines: **New Issues, Backlog, To-Do, In Progress, Done** and **Closed**.

Let's quickly go over each of them.

New Issues: New issues land here automatically. They should be dragged to another pipeline as soon as possible.

Backlog: Backlog issues are not a current focus, but you will act on them at some point. If they don't have a GitHub milestone, consider them part of your **product backlog**. Once you add a Milestone, they're part of your **sprint backlog**.

To Do: Populated with well-defined issues, this pipeline is your team's current focus. Issues here will flow into the In Progress pipeline, so order them by priority and assign keepers using the Assignee feature.

In Progress: This pipeline answers the question, "What are you working on right now?". Ideally, each team member should be working on just one thing at a time.

Done: Depending on your team's definition of "Done", an issue in here may mean it is on production or staging, or is ready for testing. Make sure your team agrees what "Done" means!

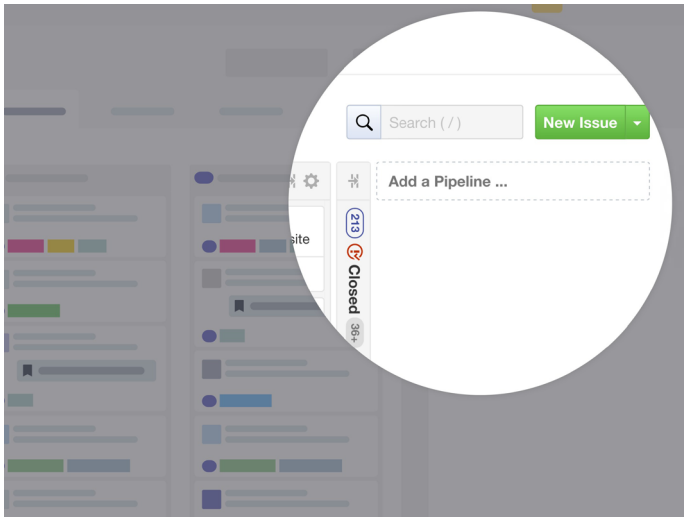
Closed: These are completed issues. Dragging issues into this pipeline will close them, while dragging them out will re-open them.

To keep ZenHub Boards clutter free, you can hide this pipeline via the checkbox at the top of the board. When the box is checked, the pipeline appears on the far right.

Making it all come together

- **Issues should be arranged by priority, with the most pressing issues at the top.** Issues get more detail as they move up in the pipeline.
- **It's not a one (wo)man job.** A well-functioning team has *all* its members dragging and dropping issues as they progress, and a well-organized task board is the key to making this possible.

Building your perfect workflow



While the default board is a great starting point for any project, you should consider customizing it to suit your unique team.

Customization can help you uncover bottlenecks faster, and provide more information for other stakeholders in the project.

Auditing your board

Ask yourself these questions:

- Does this board contain *only what we need* and nothing more? Could my boss look at this and understand everything she needs to about the project?
- **Is every group being represented here?** Can someone in design, marketing, or QA look at this board and know exactly where his or her help is needed?
- **Think about how your team builds products. Are any recurring "stages" being missed here?** It's all about keeping issues flowing through the board. If your website has its own repo, for example, you might need a "Copyediting" pipeline.
- **Is "Done" actually defined here?** And does my team know and understand our definition? Remember: defining "Done" is crucial in any agile project.

CHAPTER 3

Taking GitHub issues from good

Issues are a powerful wrench in any developer's tool belt. But while their simple structure makes it easy for others to weigh in, issues are really only as good as you make them.

Without some process, your repository can become unwieldy, overflowing with duplicate issues, vague feature requests, or confusing bug reports. Project maintainers can become burdened by the organizational load, and it can become difficult for new contributors to understand where priorities lie. It's a challenge for any team, but becomes especially difficult when that project is large or open source.

In this chapter, you'll learn how to take GitHub issues from good to great.

The issue as user story

We spoke with software expert Jono Bacon – former Director of Community at GitHub and XPRIZE, author of The Art of Community, and strategy consultant.

"High-quality issues are at the core of helping a project to succeed. While some may see issues as merely a big list of problems you have to tend to, well-managed, triaged, and labeled issues can provide incredible insight into your code, your community, and where the problem spots are."

He continues, "At the point of submission of an issue, the user likely has little patience or interest in providing expansive detail. As such, you should make it as easy as possible to get the most useful information from them in the shortest time possible."

A consistent structure can take a lot of burden off your team.

Start with consistency

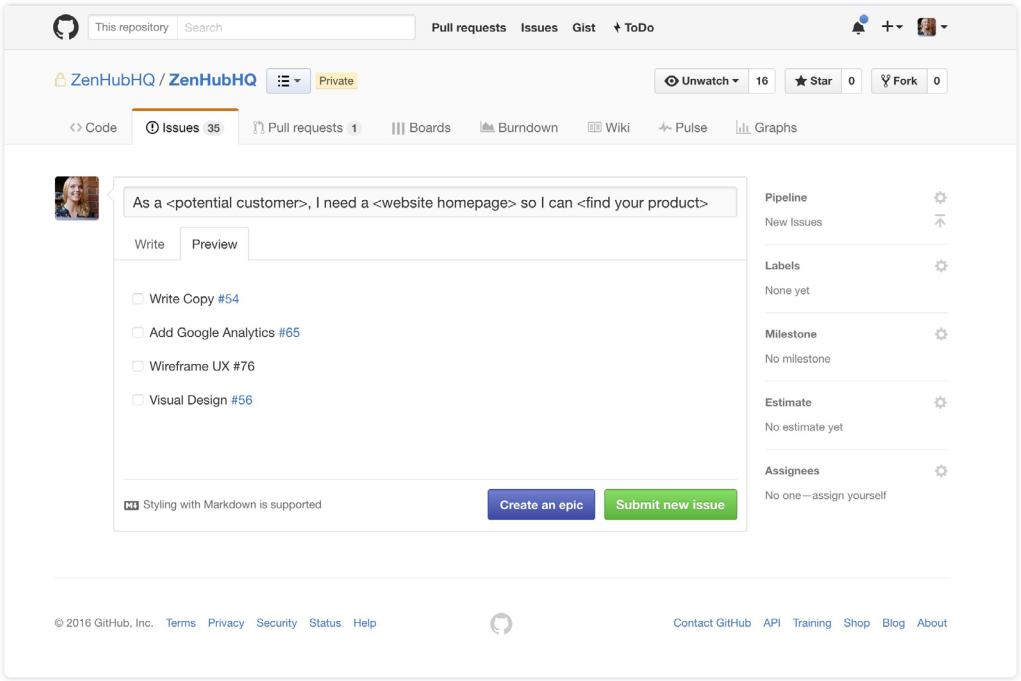
Remember: user stories address the “who, what, and why” of a feature, like this.

“As a <user type>, I want to <task> so that <goal>.”

For example:

“As a <customer>, I want to <create an account> so that <I can make purchases>.”

We suggest sticking that user story right in the issue’s title. You can set up issue templates to keep things consistent.



The point is to make the issue well-defined for everyone involved: it identifies **the audience** (or user), **the action** (or task), and **the out-**

come (or goal) as simply as possible. There's no need to obsess over this structure; other alternatives have been proposed. As long as the what and *why* of a story are easy to spot, you're good!

Qualities of a good issue

Not all issues are created equal. A well-formed GitHub issue achieves the criteria detailed in The Agile Samurai.

Ask yourself if it...

- is something of value to customers.
- avoids jargon or mumbo jumbo. A non-expert should be able to understand it.
- "slices the cake," which means it goes end-to-end to deliver something of value.
- is independent from other issues if possible. Dependent issues reduce flexibility of scope.
- is negotiable, meaning there are usually several ways to get to the stated goal.
- is small and can be easily estimated in terms of time and resources required.
- is measurable; you can test for results.

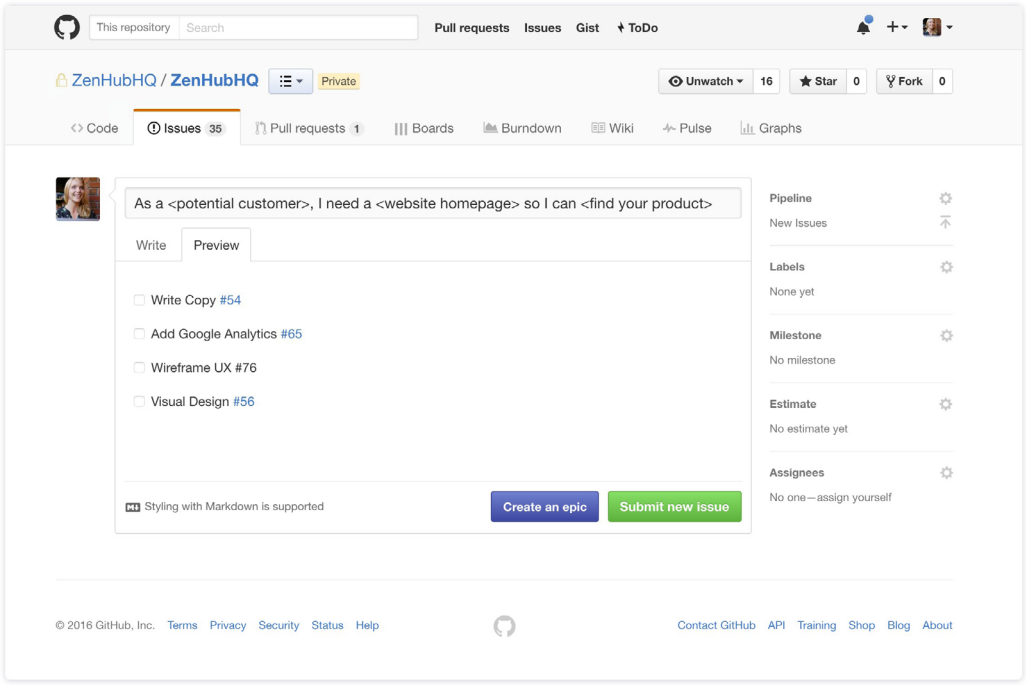
What about everything else?

If an issue is difficult to measure or doesn't seem feasible to finish within a short time, you can still work with it. Some people call these "constraints."

For example, "the product needs to be fast" doesn't fit the story template, but it is non-negotiable. But how fast is *fast*? Vague requirements don't meet the criteria of a "good issue", but if you further define these concepts—for example, "the product needs to be fast" can be "each page needs to load within 0.5 seconds"—you can work with them more easily. Constraints can be seen as internal metrics of success, or a landmark to shoot for. Your team should test for them periodically.

What's inside your issue?

In agile, user stories typically include acceptance criteria or requirements. In GitHub, you should use checklists (in markdown) to outline any sub-tasks that make up an issue.



Say we're creating an issue for a new website homepage. The sub-tasks for that task might look something like the above.

If necessary, link out to other issues to further define a task. (GitHub makes this really easy.)

Defining features as granularly as possible makes it easier to track progress, test for success, and ultimately ship valuable code more frequently.

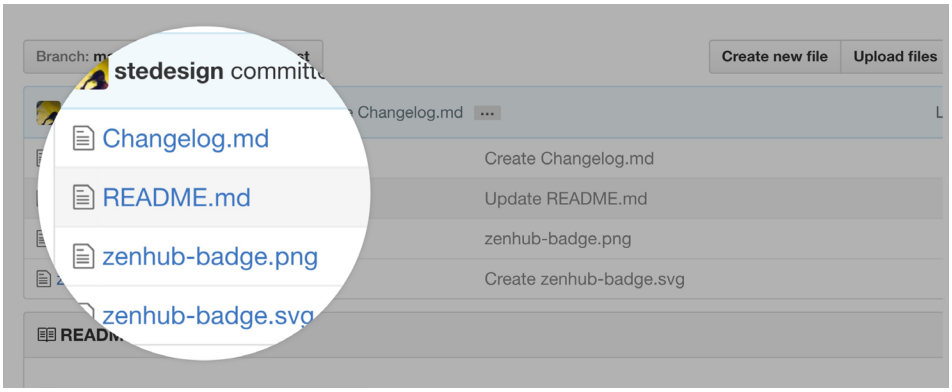
Once you've gathered some data points in the form of issues, you can use APIs to get even deeper insight into your projects.

"The GitHub API can be hugely helpful here in identifying patterns and trends in your issues," says Jono. "With some creative data science you can identify problem spots in your code, spot active members of your community, and [gain] other useful insights."

(ZenHub's API adds extra context, providing data like issue estimates, history, and task board information.)

Getting others on board

Once your team decides on an issue structure, how do you get others to buy in? Think of your repo's **ReadMe.md** file as your project's "how-to." It should clearly define what your project does (ideally using searchable language) and explain how others can contribute (by submitting requests, bug reports, suggestions, or by contributing code itself.)



The ReadMe is the perfect spot to share your GitHub issue guidelines.

If you want feature requests to follow the user story format, share that here. If you use a tracking tool to organize your product backlog, consider adding its badge so others know how to access it.

“Issue templates, sensible labels, documentation for how to file issues, and ensuring your issues get triaged and responded to quickly are all important” for your open source project, says Jono.

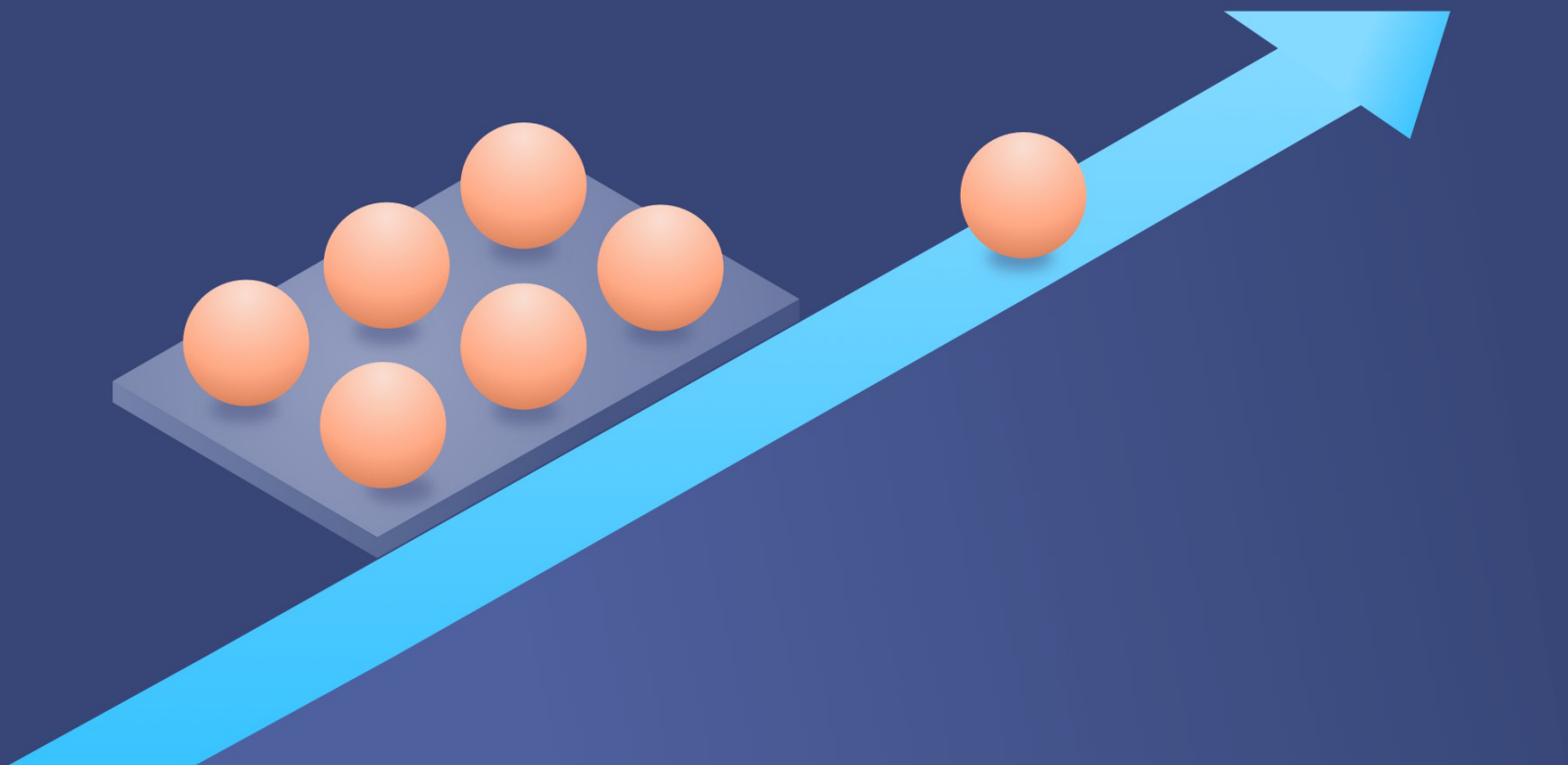
Remember: it’s not about adding process for process’ sake. Some simple guidelines can help others discover, understand, and feel comfortable contributing to your community.

“Focus your community growth efforts not just on growing the num-

ber of programmers, but also [on] people interested in helping issues be accurate, up to date, and a source of active conversation and productive problem-solving."

CHAPTER 4

Mastering your product backlog



A product backlog is a list of all the stuff that needs to get done in your agile project. It is a culmination of feedback from multiple sources, like the development team, prospects, management, marketing and, most importantly, your customers.

It's your job to take in that feedback, prioritize it, manage it, and work it into the future of your product. Is it easy? No. But by sticking to a few best practices, it gets a lot easier.

Why does it matter? A (well-managed) backlog leads to more productive and more meaningful work. The result is a better product, and one that your customers actually want to buy.

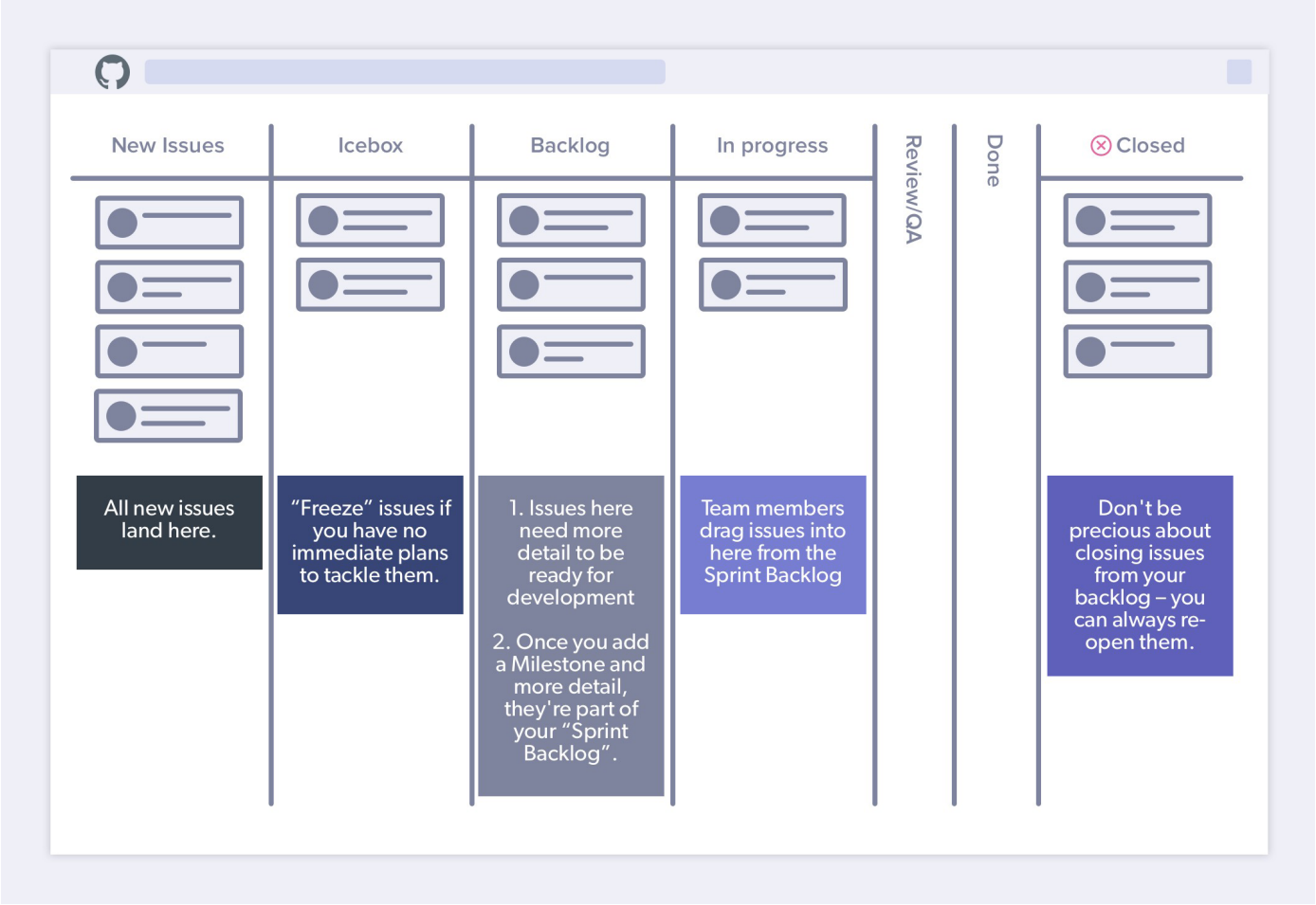
Do I need a product backlog process?

Yes. Don't fight us on this one.

Without one, your backlog will become clogged with inactionable feedback. Double-work is created, visibility goes down, and focus is eventually lost; it all adds up to a demoralizing experience.

Don't worry, though. Building a process is really easy.

How we run our backlog



Here's how the ZenHub team does it:

1. New feedback and ideas automatically land in the **New Issues** pipeline. Our product owner scans each issue and quickly decides whether each task is actionable. *Note: When we say "product owner", we mean whoever makes the ultimate call as to what is built and when.*

2. If we intend to complete an issue but it's yet not ready for development (ie. it needs more detail or the team has no capacity for extra

work), the issue is dragged the **Backlog**. Issues here don't yet have a Milestone. You'll add one at the beginning of a sprint.

- If it's a valuable issue but we have no plans to add it to a Milestone, we'll freeze it in the **Icebox**. This keeps our backlog from getting bogged down.
- If the issue isn't going to get done, we **close it completely**. Don't get too precious about this: you can always re-open it.

3. Is the issue ready for action? At this point, our team has added details (like acceptance requirements) and a user story (the what and the why.) Our engineers have added an estimate. Once an estimate and a Milestone have been attached to an issue, it's formally part of our **"Sprint Backlog."**
4. When the sprint begins, we simply filter the Board by Milestone. Individuals grab whatever is on top of our backlog and drag it to **In Progress** to indicate it's being worked on. Simple.

What makes a "good" product backlog?

Focus on making it DEEP.

Detailed appropriately.

The closer the issue is to the top of your backlog, the more detail it should have.

Estimated.

Your product backlog is more than a to-do list; it's a planning tool. Issues at the top should have precise estimates (by time, complexity, or whatever works for your team), whereas tasks further down don't need to be as specific.

Emergent.

In a backlog, time, budget, and quality are all fixed variables. Scope is not. Backlogs are emergent, meaning issues will be "frozen" in the **Icebox**, closed, added, or edited as you learn more.

Prioritized.

Issues should be vertically arranged according their business value.

What's the customer's role in a product backlog?

You know the expression "the customer is always right"?

Besides being a source of agony for folks in retail jobs, the expression applies non-negotiably to your product backlog. Customer feedback should be the biggest influencer of what you work on, and when you work on it. In agile development, the customer's needs, wants, and feedback informs all the requirements on a project.

Your ideal customer is an expert in your subject matter. They should be...

- **Familiar with your business**
- **Deeply invested in what your product does**, how it looks, and

how it works. After all, your product is solving a significant need in their life.

- **Committed to guiding your team** and answering questions thoughtfully.

Naturally, only your development team knows how other factors, like potential technical debt, will affect how that feedback should be prioritized.

Keeping your backlog healthy

A well-maintained product backlog might be the single biggest gift you can give to your users – and your sanity. The point of "backlog refinement" isn't to create needless process, but to make space for a more focused product. Think of it as clearing smog from your team's North Star.

Backlog refinement accomplishes a few things:

1. **It provides prioritization**, giving clarity and direction to everyone involved.
2. **It ensures customer feedback is observed and integrated** into the product.
3. **It prepares issues for the development team**, giving each issue a clear, concise, and testable foundation for developers to work from.

Our better-refined backlog resulted in shorter planning meetings, an increase in planning efficiency, and smaller, more focused issues.

Whose job is it?

A single point person – the product owner – should be in charge of backlog refinement. They can pull in developers, stakeholders, and customers as needed during the discovery process.

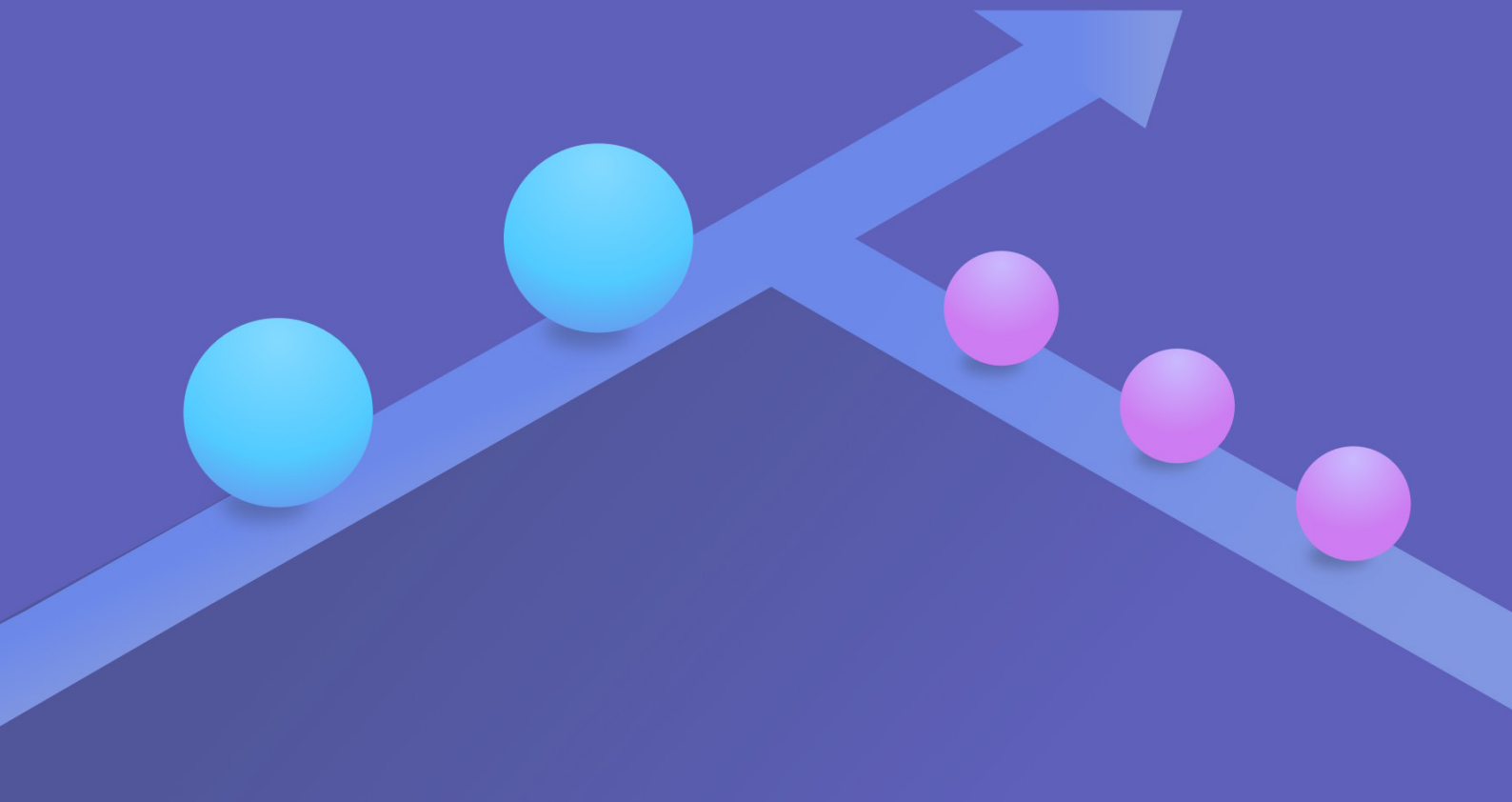
We find team-wide product backlog meetings laborious and generally unproductive. We think developer efforts are better spent actually *developing*, and that refining by committee can introduce clashes, misaligned objectives, and other potential time sinks.

When should you refine a product backlog?

We try to do it a little every day, but as a focus, we refine it around two days before a Milestone begins. There's no right answer here; your team should develop a cadence that works for you.

CHAPTER 5

A workflow for Epics and Milestones



Next, we'll show you how to create a lightweight and powerful workflow using Epics and Milestones.

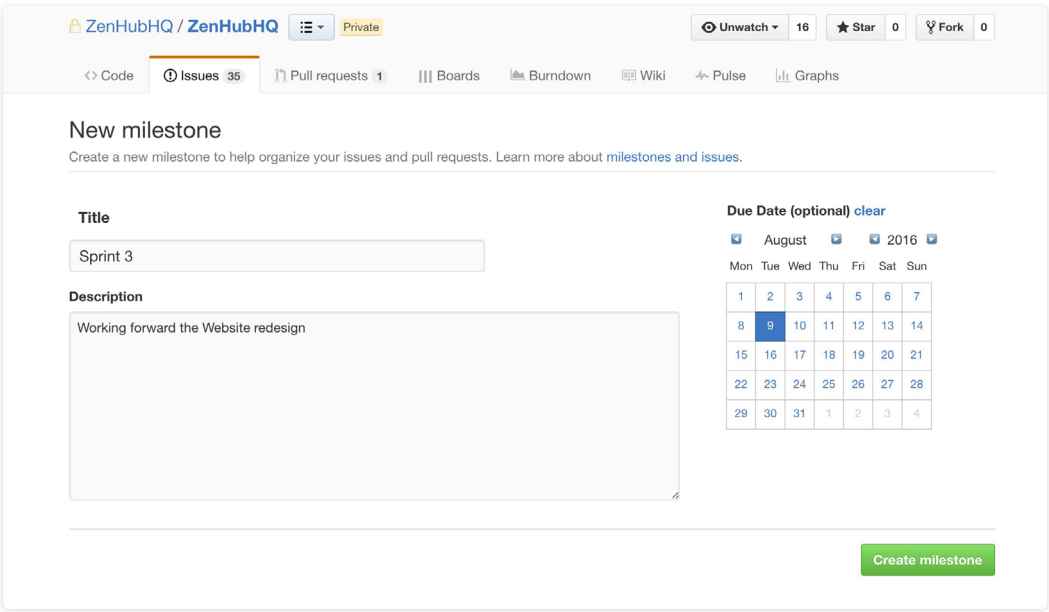
By applying some basic agile principles, you can combine Epics, issues (as user stories), markdown checklists (as sub-tasks), and GitHub Milestones to ship better software with less overhead. Read on to learn how we do it.

But first...

The difference between Epics and Milestones

One of our most common questions is how Milestones and Epics interact. Sometimes they're even confused for one another. Milestones and Epics aren't interchangeable; in fact, they work best when used together.

Remember, a sprint is typically two to four weeks of work with the goal of shipping workable code by the end. These iterative changes are a cornerstone of agile development. Here, GitHub Milestones represent sprints.



Milestones are used to track the progress of issues and pull requests. They contain issues related by *time*, and not necessarily related by *subject*. The scope of work is fixed once a sprint begins.

Epics contain issues related in *subject*, and the scope is flexible. Whereas epics can span multiple repositories, GitHub milestones do not.

In order to use epics and milestones effectively, we need to step back and look at what they're made up of: GitHub issues.

Going back to the heart of the issue

As we've discussed, we suggest using a user story as the title for your GitHub issues – that is, high-level feature descriptions that help define benefit from a customer's perspective. Remember, once a sprint (Milestone) begins, their scope should remain fixed.

But why do we keep coming back to the traditional user story format? To learn more about why it matters, we spoke with product management expert Alexander Cowan – faculty at the University of Virginia Darden and Managing Director at Synapse Partners.

"It's important to remember that user stories are there to drive team discussions about what is a valuable *outcome* for the user," he says. "They're not just another format for a specification; specifications are about *output* and that's why *spec-driven* products miss the mark so often with the user."

To illustrate this concept, he suggests a simplified example of a website that sells something. In this example, a related story might be:

```
As a [buyer persona], I want to [see my final items
```

```
and charges] so I can [make sure I have what I
wanted, and agree with the applicable charges.]
```

Alex elaborates, “That’s a good starting point for a team to design solutions for that part of the buying experience. A specification-driven approach would probably say something like ‘The page must show item descriptions, quantities, costs as well as tax and shipping.’ It seems fine too, doesn’t it?”

“A developer given that specification will probably build something that does all those things. But is it easy for the user to understand? Does it play well with the rest of the experience? Those are the kind of questions where specifications don’t perform as well. You usually end up with something that’s not wrong, per se, but just is not very good. And that’s not good enough anymore in today’s hyper-competitive marketplace.”

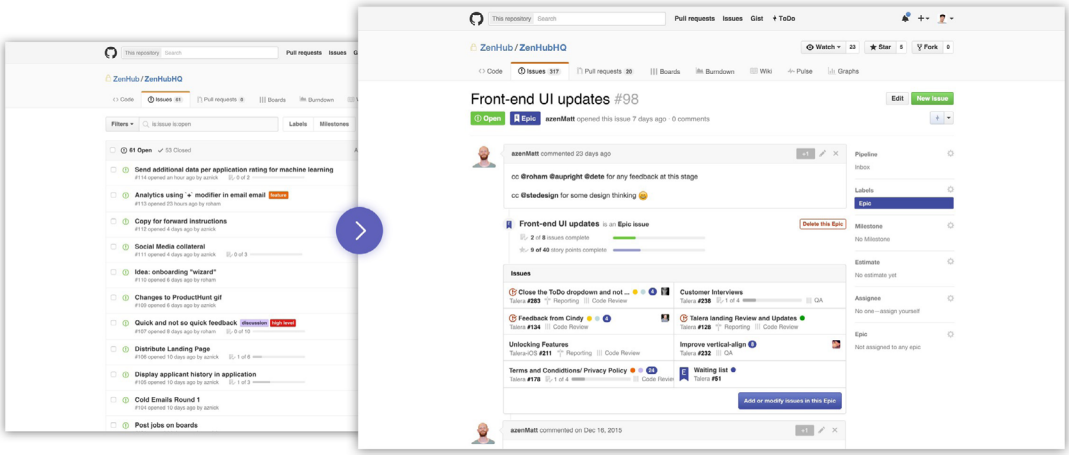
Defining agile in GitHub

To review, this is how we map agile concepts to GitHub/ZenHub:

Agile	GitHub
Epic	Epic
User Story	GitHub issue
Sub-Task	Markdown Checklist

Note: Without ZenHub, your GitHub issues are simply a list with no indication of dependancies. ZenHub’s Epics add this crucial layer of

hierarchy.

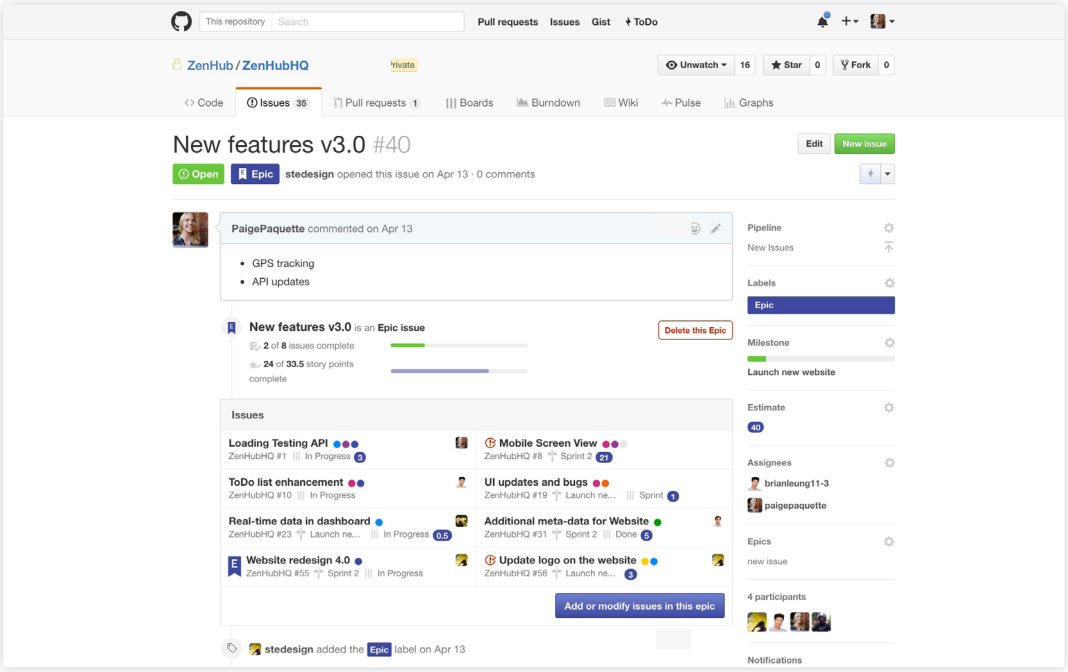


Working with Epics in GitHub

If a user story is the smallest unit of work, an Epic is essentially a “big” user story. If “as a customer, I want to be able to create an account” is a user story, the entire account management feature may be an Epic.

In ZenHub and GitHub, Epics are a theme of work that contain issues (stories) needed to complete that larger goal. They keep product backlogs coherent and organized while providing greater control end-to-end over the release process.

As you can see, we now have three three hierarchical layers: **epic, issue, and sub-task** (which can be linked to their own issues if necessary.)



So: Issue, or Epic?

When deciding whether an issue should become an Epic (or vice versa), consider the time and complexity.

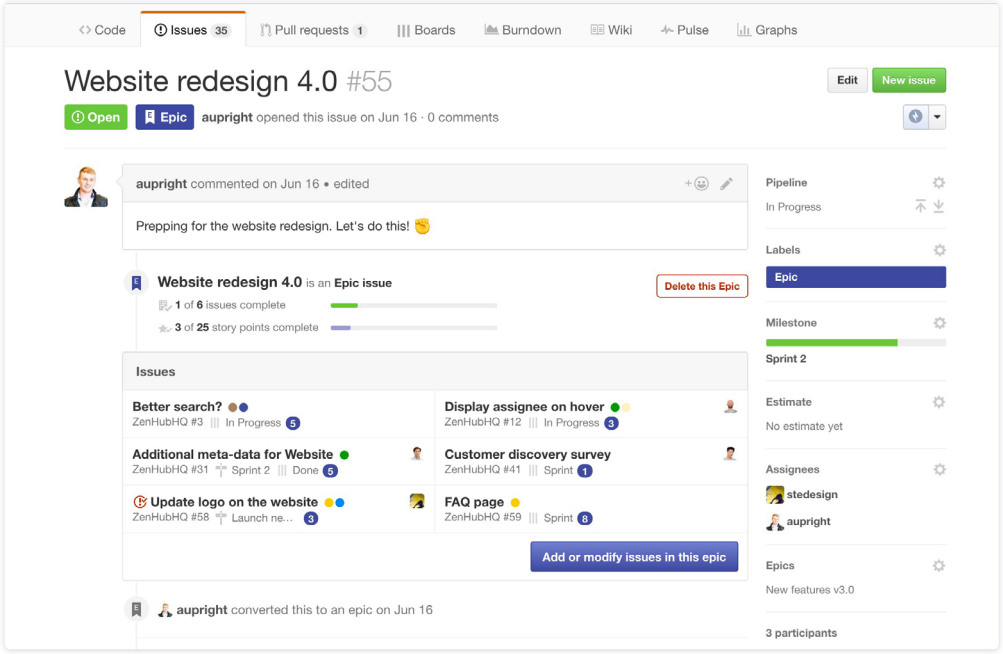
Issues should be completed in the smallest amount of time possible. If an issue will take weeks or months to finish, it should probably be an Epic. **Likewise, if an issue becomes too complex – if there are several tasks required to complete it – it's likely better off as an Epic.** Splitting these tasks into easily-completed chunks of work helps reduce technical debt, and ensures you can ship impactful changes more frequently.

A sample workflow using Epics and Milestones

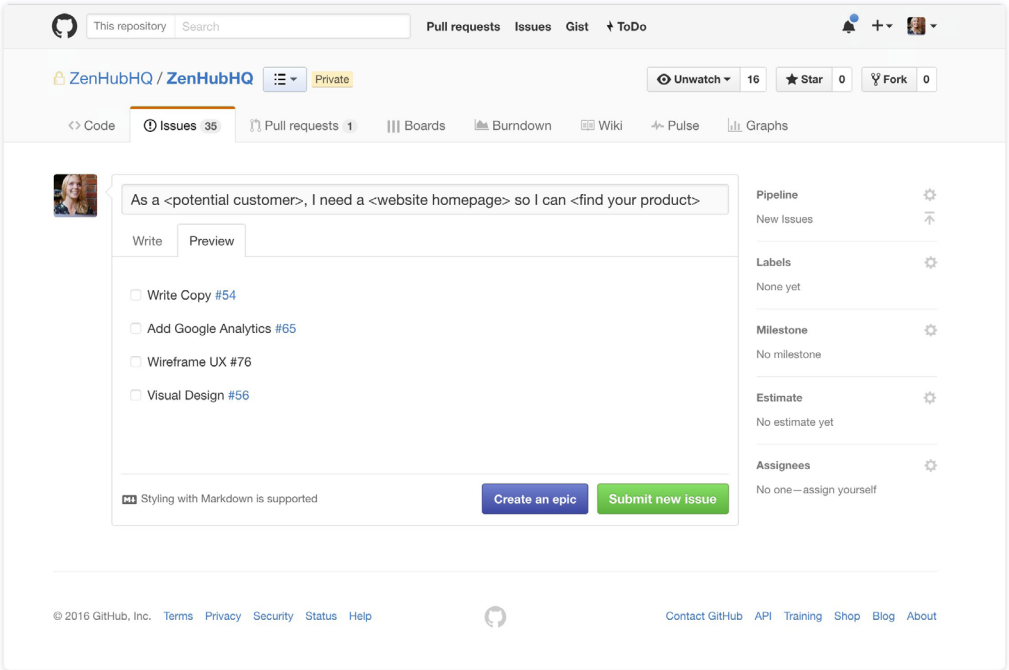
Here's an example of how Milestones and Epics interact. Let's say

we’re redoing ZenHub’s website as part of a rebranding effort.

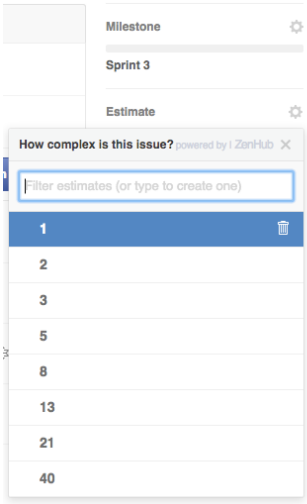
An Epic might be called “website redesign.” Inside that Epic, I create one issue for each page on that website:



As I add issues to my Epic, I’ll also create markdown checklists of tasks needed to complete that page.



At the beginning of a sprint, my team will create a Milestone (ie “Sprint 3”.) Having already estimated complexity on the issues in our backlog, we’ll decide as a team which issues should be included in our Milestone. We use Burndown Charts, which give us a pretty good idea of how many story points we can tackle in a given sprint.



We clearly can’t complete the entire website in two weeks, but finishing the homepage seems reasonable.

We’ll add the Homepage issue to our Milestone, and perhaps a couple unrelated technical fixes. Together, these issues form a chunk of work that, based on our team’s projected velocity, we believe we can finish, test, and ship over the next two weeks.

Milestones and Epics aren’t swappable, but they are complementary.

The benefits of pairing Epics and Milestones

Once you understand the differences between Epics and Milestones, you’re ready to use both of them effectively.

Epics are intended to give you a broad understanding of larger initiatives. When paired with Milestones, you're able to work towards these bigger goals in manageable iterations. This means you deliver more business value (workable code) more frequently.

If you use Epics without Milestones, you might get bogged down in details as you pursue big chunks of poorly defined work.

If you use Milestones without Epics, it's hard to effectively plan or pursue bigger goals. It's also more difficult to visualize how big chunks of work are progressing alongside all your other tasks. The lack of direction could lead to unclear priorities.

However, pairing Milestones with Epics gives you a granular way to plan and achieve your product backlog. It clarifies both the big picture and the minute details that make it up, providing everything necessary to ship better projects faster. Once you do that, you'll be ready to add estimates and get sprinting.

CHAPTER 6

Software estimation: The art of guessing badly

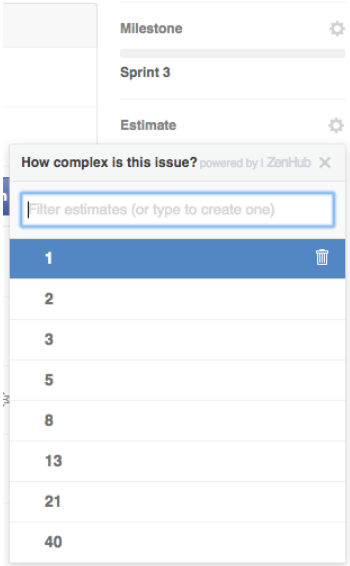
When will it be done?

Most developers dream of a world in which they could tell their boss, “the feature will be done when we finish it.” It’s not that they’re cur-mudgeonly (well, they might be – but that’s not the whole story.)

It’s because developers are *smart*. That answer is the only correct answer. Any other is a guess.

But in the real world, the correct answer isn’t always the most useful one.

Software estimation – Your guess is as bad as mine



Sometime around the late 1500s, we started using a word that sounds better than guesses: estimates.

The bigger a project is, the less accurate an estimate will be. The further away the deadline or delivery of that feature is, the less accurate

its estimate will be.

And yet, those are the situations business types and customers care most about. Nobody wants to plot their course without a map.

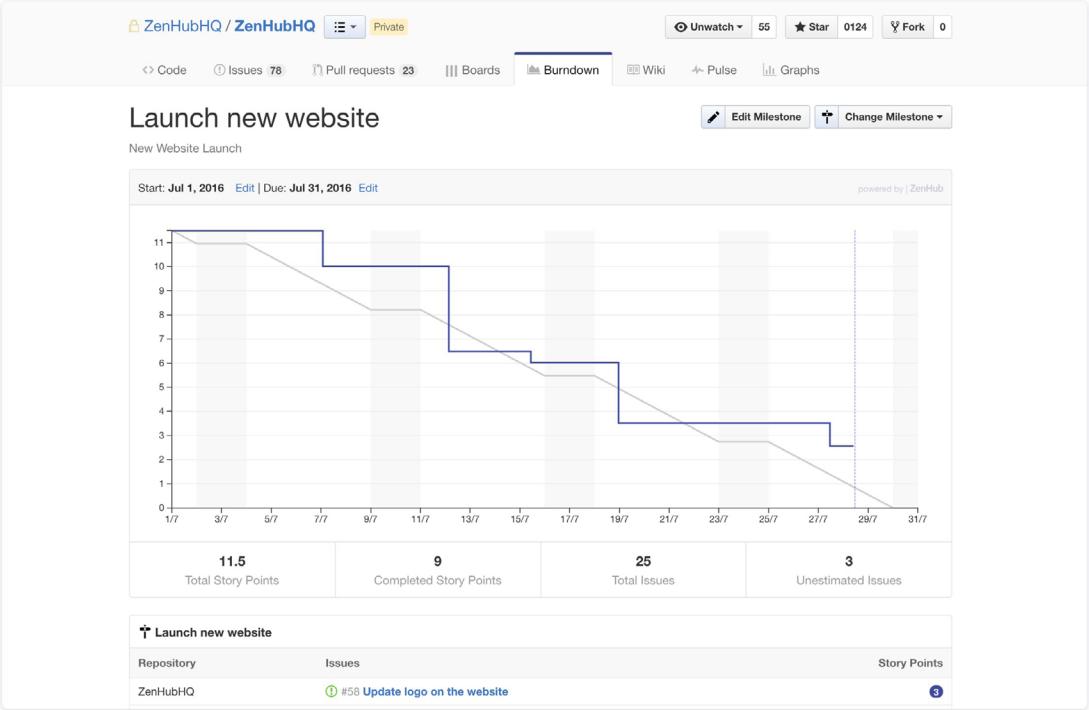
At the beginning of a project, estimates are, at best, bad guesses. In contrast to front-loaded waterfall development, most teams now add detail about tasks as they discover more *about* the tasks and the project. This means that we won't know much of anything at the beginning of a project – which is fine.

Then why bother estimating development tasks?

Estimating a task is helpful when putting together your sprint backlog. Given a set amount of time – say, two weeks – and a set budget, how can you tell which GitHub issues to tackle if not for estimates?

When paired with historical data in the form of burndown or velocity charts, your team can see how fast you're actually

going, which is an invaluable planning tool.



There's a big difference between plotting a satellite's course to Jupiter's orbit and estimating a timeline for software development tasks. The former requires dozens of people with MENSA-level IQ scores and billions of dollars in funding; the latter requires something more akin to a homemade compass and a gas station map. You just need to have a general idea of where you're going and the route you're going to take to get there. Estimates are how you do that.

How should you estimate?

There are two main types of estimates: story point and time.

Story point estimation

People are mediocre at guessing how big something is in absolute terms, like hours or days. But we're surprisingly good at sizing something up in *relation to another thing*.

For example, if you give a person two piles of beans, they probably won't be able to tell you how many beans are in each pile. However, they should be able to say one pile is about twice the size of the other pile.

That's the basis of story points. It's unitless scale of measurement in which "units" are sized in *relation* to each other. Unlike hours or days – which are specific measurements based on consistent values – they rely on arbitrary measurements that serve no purpose other than to compare a tasks' size with the size of your other tasks.

Agile estimation uses a point-based system because it's fast and you don't need to stress about how your *actuals* compare with your *estimates*. Point estimates also remind us that estimates aren't a deadline, but the best guess.

Your priorities when estimating

Whatever your estimation method, you should prioritize two things.

The team.

Never estimate in a vacuum. Harness the wisdom of the crowd in your estimations. Your estimates will be better, and your team will like you more.

Speed.

Don't get caught up in the details, because you're still just guessing.

Three ways to estimate software

There are a few main ways you can tackle the challenge of estimation:

Planning poker

Everyone on the team gets a set of cards with point estimates. When an issue is read, each engineer holds up an estimate card they feel is appropriate. The benefit of this method is discussion, not compromise: if Sarah and Tim hold up one and seven respectively, don't give the task an estimate of four. Talking about why your estimates are so different can reveal miscommunications about scope, which is helpful.

T-Shirt sizes

It's all about simplicity. Using T-shirt sizes (XS, S, M, L, XL) guarantees you won't over-analyze things, and they're yet another reminder that your estimates can't be equated to measures of time. Have your team come up with a range of story points that equal an XS, S, L, and so on. If something is an XL, consider breaking it down into several smaller tasks.

Triangulation

Triangulation is just a fancy word for "matching like with like".

Take your issues and choose one that's a definite *one*. Take another that's an estimate of *ten*. And another that's somewhere near the middle. Now you have a basis on which to estimate the rest of your issues comparatively. Match the rest of your tasks relative to these baseline estimates: *Is this issue bigger or smaller than my "one"?* Keep going until all your issues are estimated relatively.

What about time estimates, then?

Back to that conversation with your boss. “When will we release this feature?” she asks.

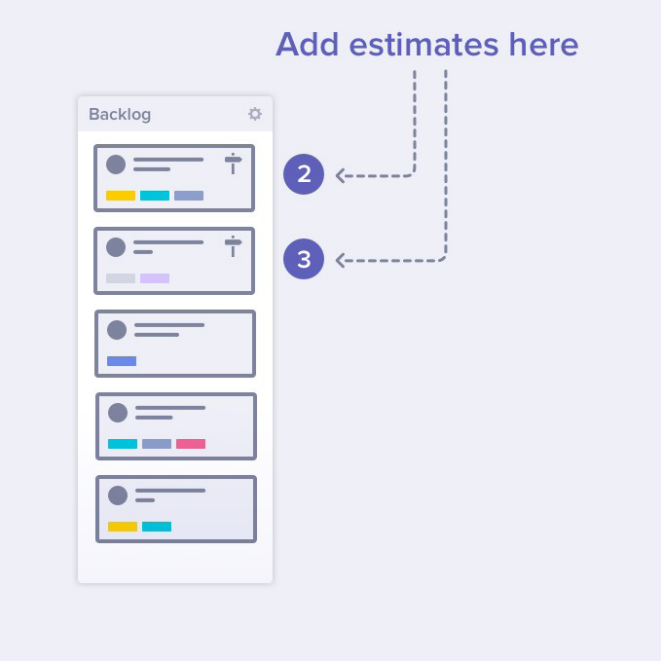
“Well,” you reply, “it’s an extra small.”

You haven’t exactly solved her problem, have you?

Time estimates are useful only when you have enough information – which is to say that you should use them when your development team has gathered some detail and done some discovery work.

In contrast to point estimates, time estimates refer to the **actual productivity of real living people**. What takes Sarah three hours could take Sam two days. That’s why nobody but the people coding should be estimating time. If you’re adding time estimates, consider the issue’s scope fixed.

When and how you should estimate



Some people use time estimates; others use story points. After trying a

couple methods, here's what we've landed on.

First, we don't spend time estimating tasks until we've done some discovery work. To keep overhead as low as possible, issues that are in our Icebox never get estimated.

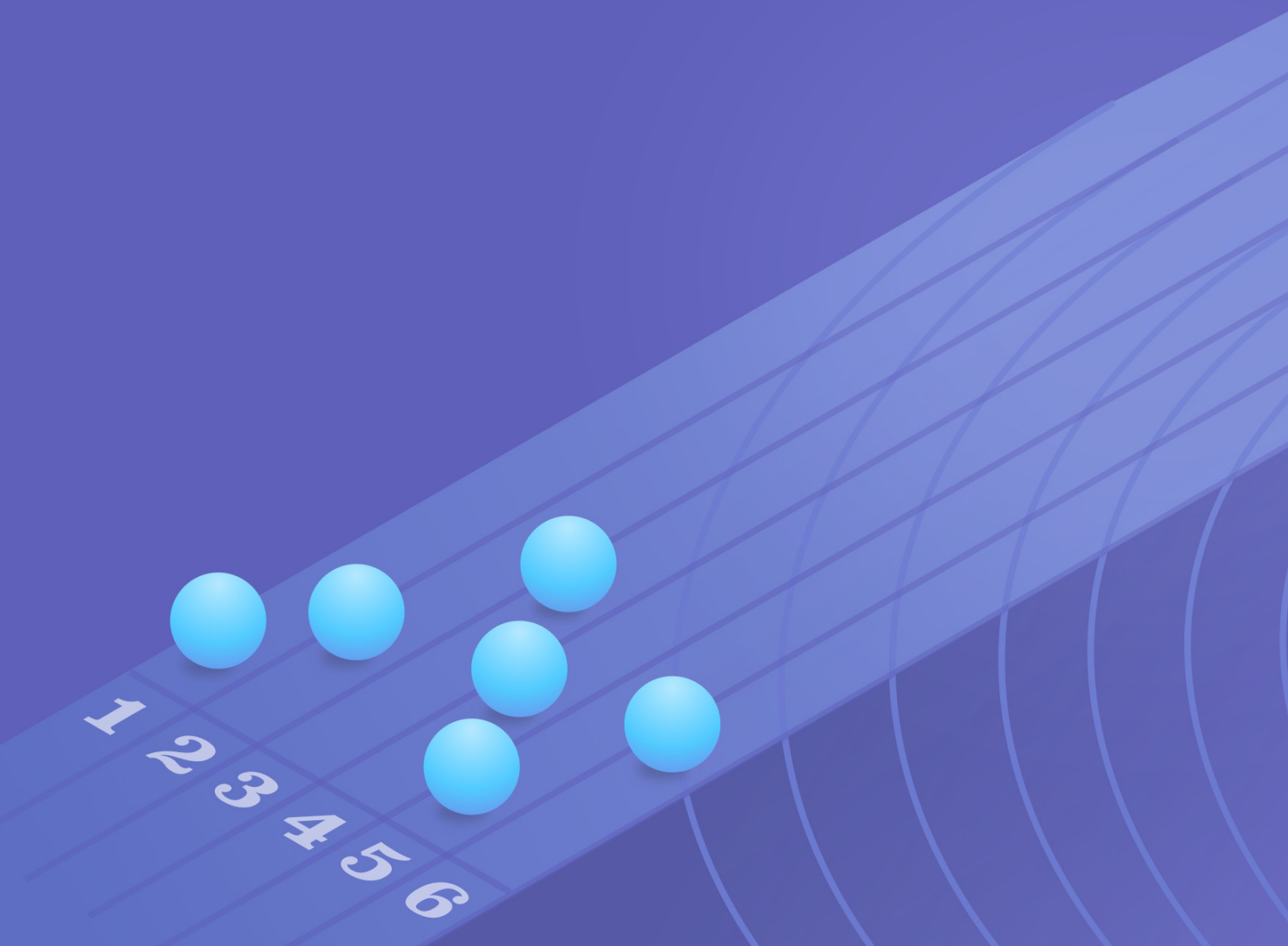
At this point, our issues have been broken down into consecutively smaller chunks as we learn more about them. This probably means they're sitting in our product backlog (the backlog pipeline with no Milestone attached.) By adding estimates at this stage, we'll easily be able to construct our sprint backlog (that is, which issues to add to our next GitHub Milestone.)

Issues that are estimated and Milestoned automatically appear in our Burndown charts. When an issue is closed, a new data point will appear on the chart. As a result, we can see how our *projected speed of work* compares with our *reality.*

Software estimates have long been considered a challenge – but with a little practice, some experimentation, and some historical data, your team will be rewarded with more predictability and greater confidence in your development process. Are estimates worth doing? We'd guess so.

CHAPTER 7

What to do before a sprint



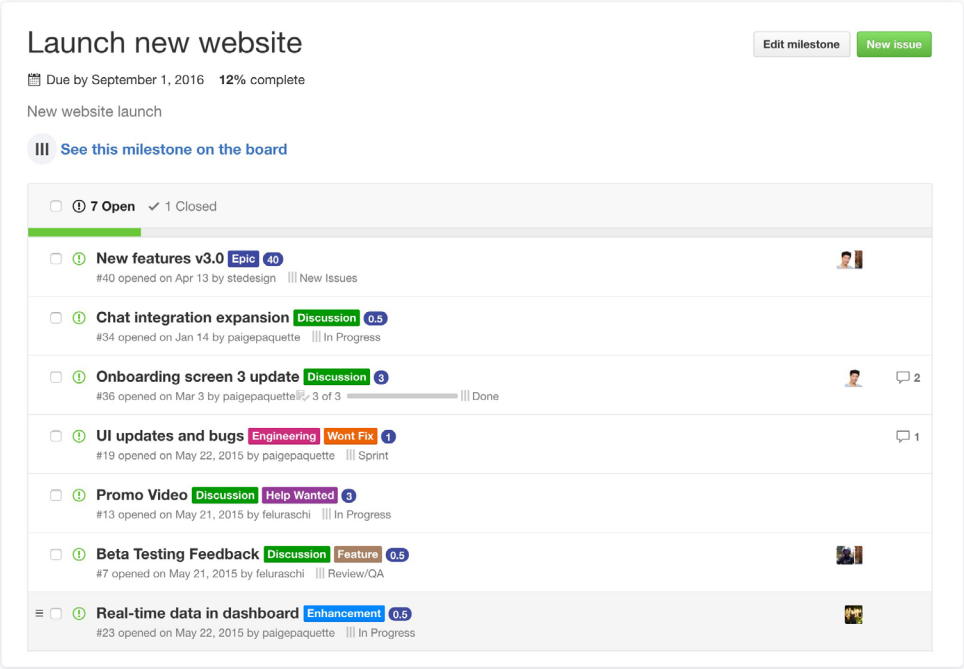
You've set up the perfect task board, customized your labels, refined your backlog, and conquered time estimations. Go you!

But important questions must be answered before you start sprinting. *How much work can you actually tackle? What issues should you choose?*

Accept that, especially if this is your first foray into an agile-ish process, you won't have all the answers. *Time*, *budget*, and *quality* are fixed, and now it's time to predict the the *scope of work* your team can accomplish. The only way to answer these questions is to talk to your team.

With this in mind, here's how we approach a sprint.

Milestones = sprints



Remember, when we talk about GitHub Milestones, we're talking about *sprints*. We use the terms interchangeably.

In GitHub, a *sprint backlog* is created by adding a Milestone to the issues in your Backlog pipeline. Issues in your backlog pipeline *without* a Milestone are your 'product backlog' – the things you'll get to eventually.

Choose a due date for your Milestones that aligns with the end of your sprint. You should be able to ship something useable in that time; if you can't, then your issues are too big and you should break them down into smaller pieces.

What to include in a sprint?

To figure out what issues should be added to a Milestone, you need to have a healthy product backlog.

"Having a prioritized list of your high-level requirements or your product backlog is a key input going into the first agile sprint," says agile consultant Yvette Francino.

In other words, your issues should have:

- **Estimates** that you and your team decide on together
- **A user story** that addresses the who and why of a task. Don't worry about adding much detail yet – you'll discover more when you start your Milestone
- **Priority** according to the issue's value to the user

Once an issue has a Milestone attached, it's ready to be assigned to your team. If you've done a good job organizing your backlog, team members can assign themselves; otherwise, the product owner can do it.

So how do you decide what issues belong in your Milestone?

How much work goes into each sprint

A “sprint goal” is the amount of work your team agrees can be completed within a sprint. The goal should be a collective effort; everyone should have a say.

As we discussed in the previous chapter, the unpredictable nature of time estimates is the reason why we prefer to use story points instead. Using story points, we are able to estimate *relatively*. Since your tasks aren't tied to real hours, you're able to reduce complexity and account for sick days or vacations.

There's no perfect answer for the amount of work to include in your first sprint. You're *almost definitely* going to get it wrong the first time. If the number of story points in your first sprint feels about right, then stop worrying and start sprinting. After a couple weeks, you'll have enough historical information to inform your next sprint. Don't stress about it!

However, if your team is standing firm on time estimates, you can use some simple math to put together your first sprint. First, add your estimated issues up until they fill up your sprint’s time allocation. For example – in an ideal world, each team member is capable of completing 70 hours of work during a two-week sprint. With five team members, that adds up to around 350 hours of work.

Because you don’t know what your average sprint velocity is, it’s easy to over-commit in the early stages. (Don't worry too much if that hap-

pens. If necessary, you can always adjust scope.)

To accommodate for variables like inaccurate estimates, unexpected requirements, and real-world realities, one-third of the sprint time is left unassigned. For a five-person team, this means you'd assign roughly *230* hours per Milestone – not the 350 hours quoted in the “ideal world” scenario.

Remember: your primary goal during a sprint is to make sure each assigned issue is fully delivered.

Your first sprint planning meeting

If you've done everything right, you should be entering your sprint with estimated issues and a prioritized backlog. Does that mean you have all the information needed to move forward? Hardly.

Only your developers have all the information about dependencies, conflicts, or the urgency of certain fixes. That's what your sprint planning meeting is for. It should be a chance for your team to advocate for the inclusion of one issue over another.

That doesn't mean your sprint is a free-for-all. Quite the opposite: you should make it a best practice to stick to some theme for each sprint. Not only do themes help set a clear goal to move toward, but they help keep your users happy.

Think about it. As a user, what would impress you more? A bunch of tiny improvements to a handful of features over the course of eight months, or big obvious improvements to one feature at a time? **The latter is certainly more noticeable, and gives a clear message: your team is frequently shipping high-value improvements.**

The value of sprint velocity

When you assign issues according to how long you think it will take, you are taking a guess at your team's *projected velocity*.

You only have a rough idea of your team's velocity when you first start a sprint. Over time, you can track your team's *actual velocity*: the actual rate it takes your team to get work done. Knowing your team's *actual velocity* enables more accurate estimates and, as a result, more effective sprints.

Unfortunately, a single milestone can't really be considered an accurate assessment of your team's velocity. It will take a few months before you have the data necessary to make useful assumptions.

When you've been through a few sprints and have some historical data, velocity charts become a useful tool.

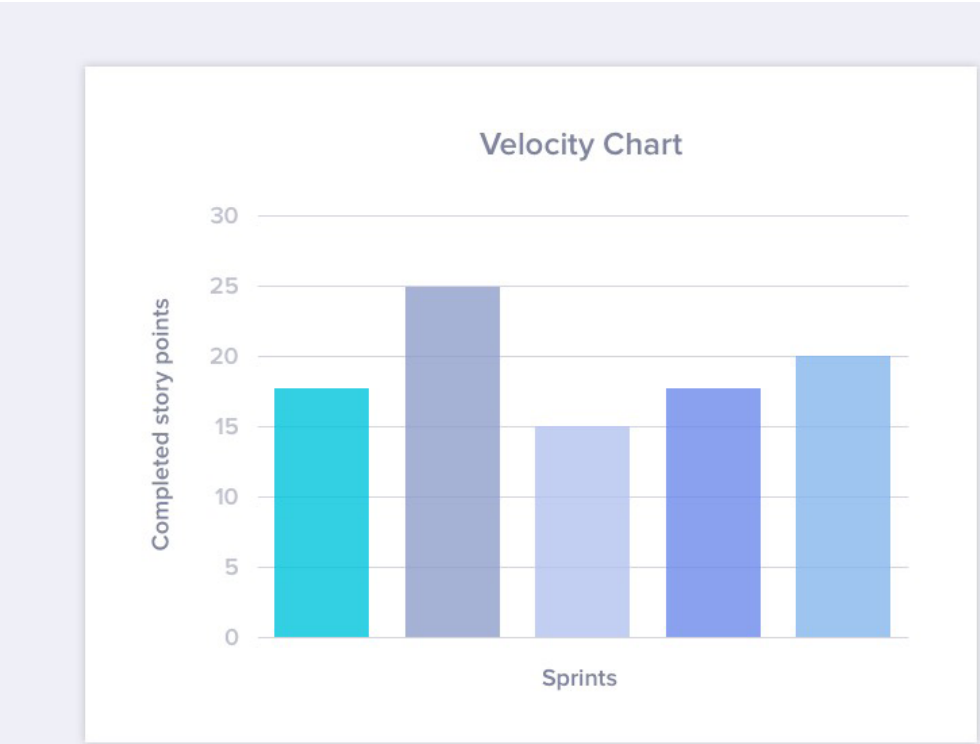
Using Velocity Charts

Where burndown charts show how things are going, velocity charts show you how things *went*.

People, as a rule, overestimate how much work they can do and underestimate the complexity of the work they're doing. Velocity charts provide a precise picture of what your team can really handle.

Note that we measure team velocity instead of individual productivity

to sidestep these estimation shortcomings, and to build a culture of healthy collaboration.



In The Agile Samurai, Jonathan Rasmusson explains:

> **When we create plans based on our team's velocity, we are making a commitment as a team. We're saying, "We, as a team, feel we deliver this much value each and every iteration."**
>
> **This is very different from measuring individual productivity—which leads to the dark side of project management. If you want more bugs, more rework, more miscommunication, less collaboration, less skill, and less knowledge sharing, then by all means, promote, highlight, and reward individual developer**

productivity.

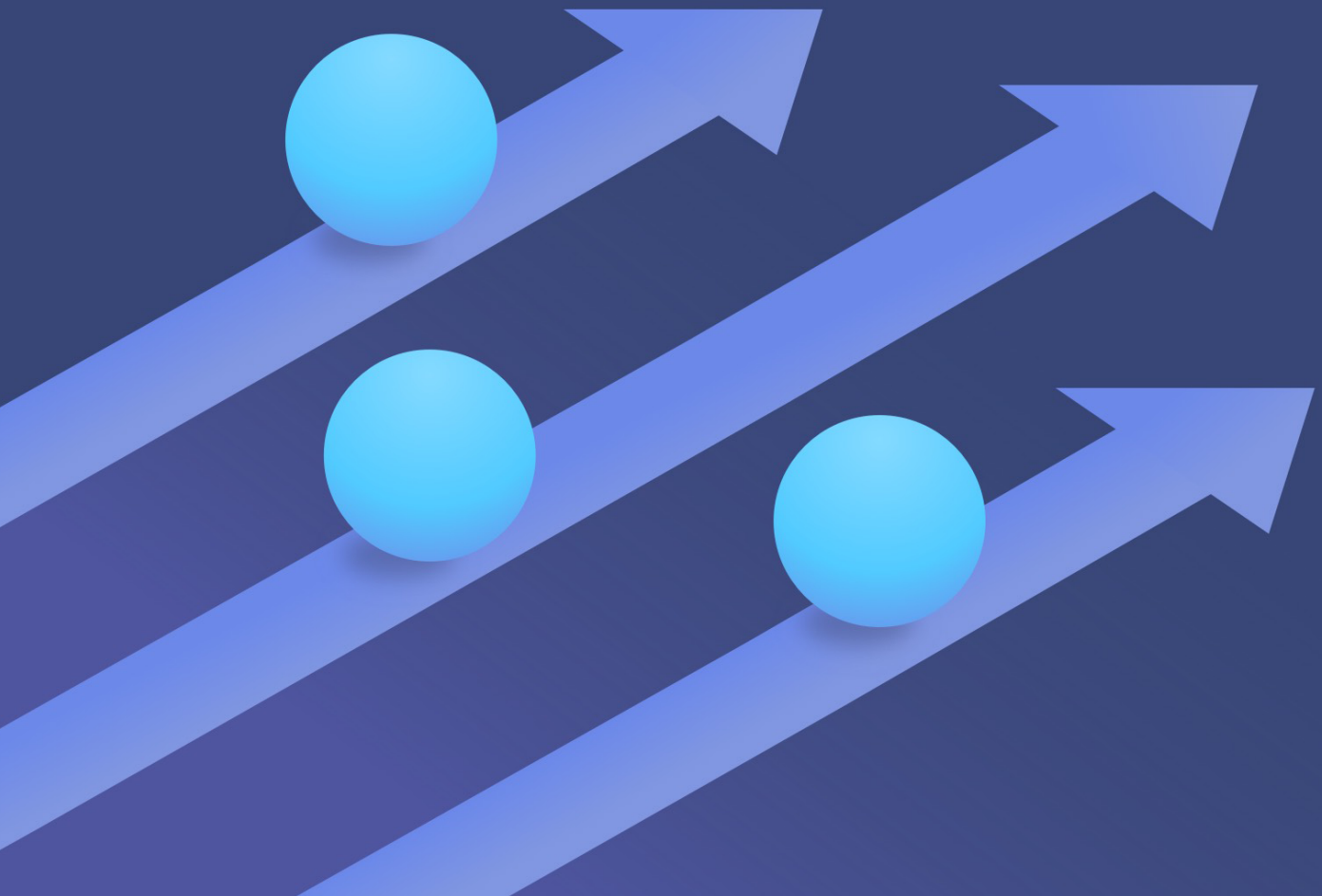
Just understand that by doing so, you are killing the very spirit and behavior we want to foster and promote on our projects: sharing ideas, helping each other out, and watching for things that fall through the cracks.

Reason enough for us.

Now the fun part begins: are you ready to ship some code?

CHAPTER 8

What to do during a sprint

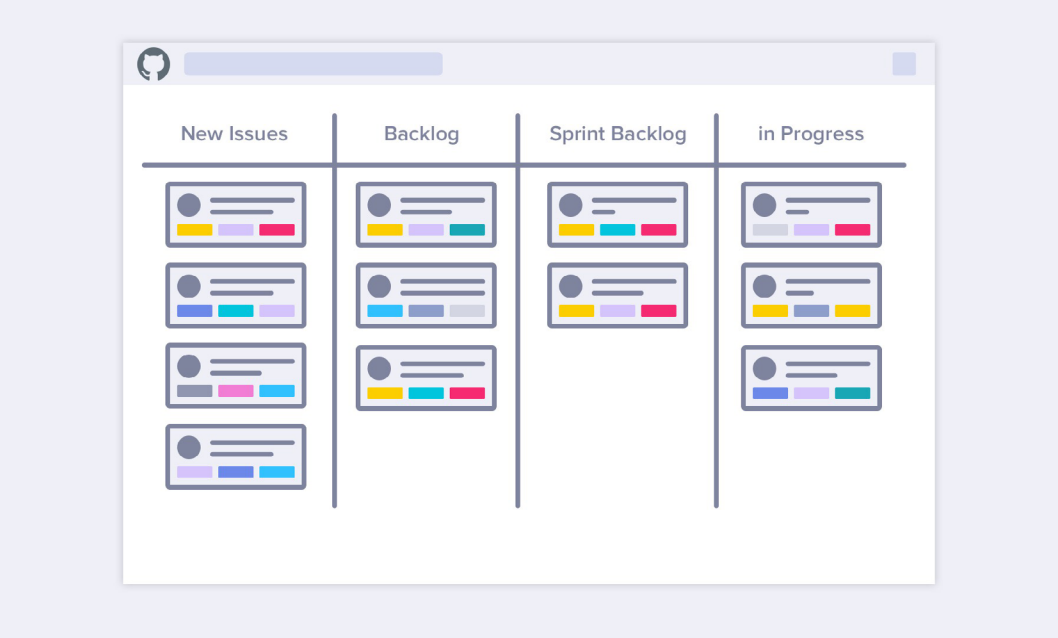


Start Sprinting

Once you've refined your backlog, set up Milestones, and established a sprint goal, you're ready to begin a sprint. Congratulations!

After you get started, it all about keeping the project in motion. Here's what to do during your sprint to keep everything copacetic.

The day-to-day of a development sprint



During a sprint, task boards are at the center of everything you'll do. Make use of labels and pipelines, and drag and drop issues into the correct places as you move forward.

If you've set up a task board that reflects how your team actually works, then they should portray exactly where things are at. That's part of the reason it's so important that *everyone* involved with the project uses the board.

It's important to emphasize that a lot of teams aren't used to this level of individual autonomy. It might feel unnatural at first. Many developers are accustomed to being told what to do, and many managers are used to owning every part of the project management process.

This isn't the fault of a bad manager or a lazy developer; it's because most project management workflows and tools are too distracting from the real work (the code) for us to reasonably expect developers get very involved. That's the reason why we built ZenHub in the first place. If your team isn't used to this system – or they're using it ineffectively – gather everyone together and formally introduce them to this new way of working.

With everyone actively using the Board, your developers will get more done in less time. The project lead will spend less time bugging people for updates. Everybody wins!

Daily standup meetings

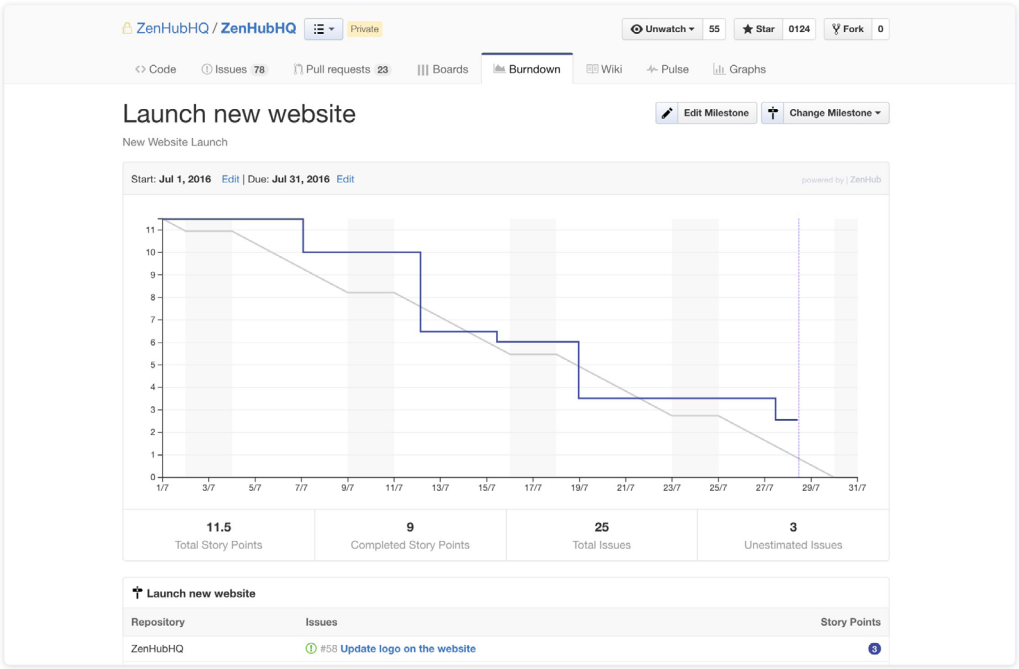
Every day, the ZenHub team conducts a short standup in which we quickly state what we did yesterday, and what we're doing today, along with any blockers or difficulties. Holding it in the morning (usually with a mug of coffee in hand) helps us set the stage for the day ahead. Everybody, including sales, design, and marketing, participates.

Besides being a way to share information and spot blockers, in-person standups are valuable in that they mandate public commitment: you're proclaiming, quite literally, what you'll be delivering. It makes everyone accountable over their day, every day.

Watch out for two common mistakes of daily standups. One: we tend to spend too much time talking about our work, and not enough talking about our blockers. Making sure to mention your own blockers is an act of generosity to your teammates.

And two: standups often devolve into technical discussions. They're meant to be quick, so if you start debating something, take it into a GitHub issue and let the day begin.

Know thyself: Using Burndown Charts



Burndown Charts display your completed work (as closed issues) in relation to your projected velocity (how much work you thought you'd get done.) Though they're often associated with Scrum – an agile methodology involving frequent releases, incremental progress, and a focus on customer requirements – they're a handly little tool for any team.

During your sprint, check in with your burndown charts every day to see if things are progressing as planned. They're an excellent way to keep an eye on scope. If you find you've overestimated how much you can get done, don't worry – it's a very common misstep. Simply stop starting new issues and focus on completing (at least) a couple of existing ones.

As an early indicator of how projects are progressing, Burndown Charts can help teams meet deadlines and track their velocity. Since they visually display complete and incomplete work, they're a user-friendly reporting tool – the quickest answer to the question, "How's that project going?"

Building out your Burndown Charts

In ZenHub, each Milestone (sprint) gets its own Burndown Chart. To build your chart's x-axis, choose the Milestone's start and end dates.

When issues with estimates are added to that Milestone, they'll show up on its Burndown Chart. When an issue is closed, a new data point will appear on the chart. You can be confident you'll achieve your goal if your team's completed work tracks closely to the diagonal timeline.

What's done is done. Maybe.

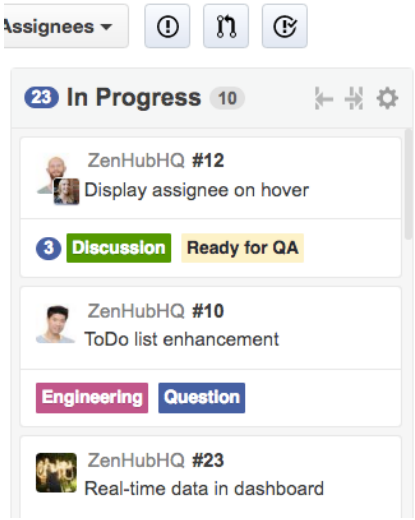
If you ask ten different people "when is an issue done?", you might get ten different answers.

The QA person says it's done once it's been tested.

A developer says it's done when the pull request has been merged to staging.

Your CEO might say it's done when it's in the hands of users.

It's critical to establish a definition of done that everyone on your team understands and agrees on.



Testing and QA during a sprint

If a feature isn't tested, it cannot be done. Thus, testing will be a continuous process during your sprint.

There's a reason we're so adamant about breaking tasks into small pieces. The sooner you can complete an issue, the sooner the testing process can start. The closer to the start of your sprint you begin testing, the better chance you'll have of being able to deliver valuable code by the end.

We suggest adding a "ready for QA" pipeline to keep issues visible and moving forward. Alternatively, use "`Blocked by:`" or "`Ready for testing`" labels to let other team members know where their help is needed.

Pull Requests and code review

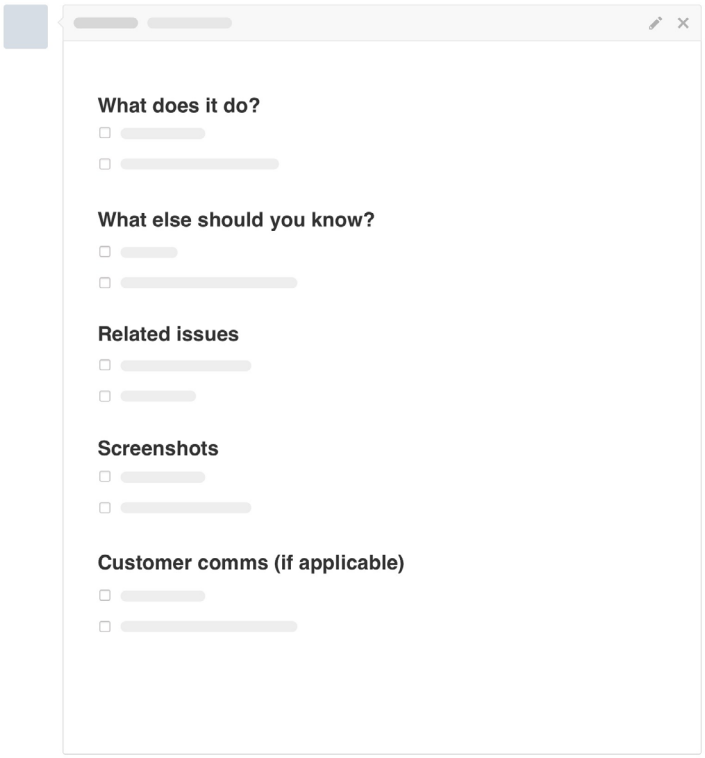
Pull requests are GitHub's way of telling your team that you're requesting a code merge. Others can review the proposed code, discuss different solutions, and add follow-up changes.

"Code review is essential to readable and maintainable code...it's to programming what editing is to writing," says Trey Hunner, Director of the Python Software Foundation and host of the Weekly Python Chat. "You can review your own code, but getting others involved is often more productive."

We maintain that having others review your code is a non-negotiable. Pull requests are an opportunity for learning, and we treat them as such. In fact, making pull requests the cornerstone of our developer onboarding allow us to get new team members shipping valuable code Day 1.

To make it work, it's helpful to have some standardization to what constitutes a "good" PR.

Just like issues, you can set up pull request templates. Here's an example:



(We include a section for customer comms so we remember to loop back with any users who originally requested the feature or submitted the bug.)

So, what else goes into a great pull request?

Do create “single responsibility” pull requests which change one thing only. There’s nothing less useful than a “kitchen sink” pull request.

Do encourage your team to add a title that describes what will happen when it’s merged.

Do make ample use of @-mentions to tag team members on specific parts which may need special attention.

And by all means, **Do** make it a point to also comment on the positives when reviewing another's PR!

At the end of your sprint, open Pull Requests should be merged, closed, or moved to the next Milestone. By taking the opportunity to make pull requests a hub for constructive dialogue as well as praise, PRs can be a powerful force for mentorship and even camaraderie in your engineering team.

CHAPTER 9

What to do after a sprint

You've done it. The milestone ended. Code that solved a specific problem was tested and shipped. One of your programmers grew a beard. No one expected mutton chops from Martha, but your team is being very supportive of her choices.

All in all, it was a successful sprint.

Now it's time to take stock, analyze your work, and fine-tune your process so that your next sprint is even more successful.

"Teams that don't have retrospectives never close the feedback loop," says Wes Garrison, a Rails consultant and Ruby conference organizer. "They say they want to improve their processes, but they don't have a system to actually put those changes into place."

Let's discuss what that system might look like for your team.

Show the thing off

If your team is centralized in GitHub and ZenHub – and therefore already highly collaborative – they are probably familiar with what's been accomplished during the Milestone. Regardless, demos can be a helpful practice to stay connected with your end user.

Try doing a bi-weekly or monthly demo from the user's perspective. Similar to how we emphasize a user story format in an issue's title, you should place an emphasis on how you're continually solving new problems for your customers.

This is the perfect time to bring in other people, like designers, executives, marketers, salespeople, and the customers themselves. Not only does it feel awesome to show off what you've been working on,

but it's also helpful to put your complex day-to-day work into terms that everyone can understand.

Tweaking your process with sprint retrospectives

After a Milestone is completed, you need to make time to reflect on it – especially if you're new to this workflow. Working constantly might feel good, but without reflection it's akin to being too busy mopping to turn off the faucet.

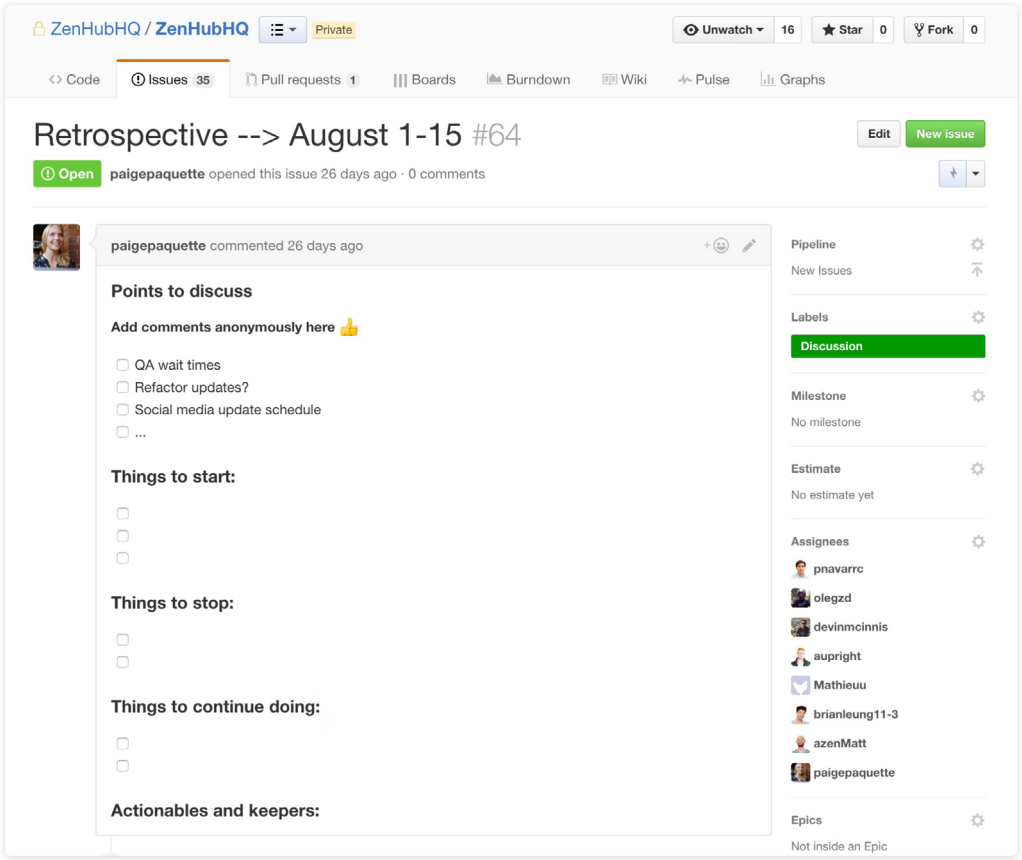
We spoke with Henrik Kniberg, author of Scrum and XP from the Trenches and agile coach for Spotify and Lego. He says that before a sprint, the number-one priority should be having the team agree on what success looks like. Then when the sprint finishes, says Henrik, "the team should follow up on whether the sprint matched their definition of success, and discuss what they can do to make the next sprint even more successful."

What does this look like in practice? At ZenHub, we schedule 30-45 minute check-in meetings every two weeks. To keep the conversation focused, the product owner or project manager will prompt us to come prepared with any points of discussion, which are collaboratively added to a GitHub issue.

Having a successful retrospective meeting without everyone's participation is difficult. But sometimes, group dynamics can make it hard for people to speak their mind – especially in close quarters.

To ensure even your shyest team member is heard, encourage the team to edit the GitHub issue anonymously. The issue template we

follow is below. *(Note that we created a Retrospective pipeline so our check-is are easier to reference.)*



Often, topics will come up that warrant more attention. Mark them separately as comments in that issue, then schedule separate meetings with relevant people to follow up.

The most important aspect of a retrospective

The core of any successful retrospective is honesty.

Unless you are transparent about where problems originate, you'll only be treating their symptoms.

If your team is anything like ours, nobody wants to point fingers. To address this problem of politeness, encourage a non-judgemental environment focused on improvement and information-gathering. Shaming or punishment have no place here. Set the tone by establishing a safe space; for example, open the meeting by giving out +1s for recent wins.

The intersection of caring personally and being willing to challenge directly is what Kim Scott calls radical candor. (We highly recommend watching entire keynote.)

Scott argues that radical candor – the practice of giving, receiving, and encouraging honest guidance – is actually your moral obligation as a team mate. She says:

> **Radical candor is humble, it's helpful, it's immediate, it's in person — in private if it's criticism and in public if it's praise — and it doesn't personalize.**

In contrast to radical candor, most of us fall into a dangerous space Scott calls *ruinous empathy* – which is to say that we're so intent on being "nice" that we end up ignoring problems and sabotaging our team in the process.

While feedback should be immediate and impromptu, regular retrospectives are an invaluable tool to building a healthy and radically candid team.

Here are a few ways you can improve your retrospective.

Retrospectives: Start, stop, or continue

During the meeting, ask each team member to name things they should start, stop, and continue doing. “Continue” items are the things that are helpful but aren’t yet habits. Mark these in your GitHub issue.

To stay accountable, open each retrospective with a 5-minute review of the previous sprint’s issue. Ask the team: did you stop, start, and continue the things you said you would?

Retrospectives: Keep it focused and actionable.

It’s always better to have a couple of high-value actionables than a bunch of vague ones. Identify a few things your team will do differently in the next sprint, mark them in GitHub, and remember to follow up next time to evaluate how it went.

We close our retrospectives by verbally repeating what our actionables are, and state who the keeper is. A lot can come up during a retrospective, so this final step helps cut through the clutter and ensures everyone understands who is responsible for what.

Retrospectives: Keep asking “Why”.

Of course, the value in reflection isn’t just about identifying what you did *well*, it’s openly discussing problems and areas of improvement.

Root cause analysis is a fancy term that means “tracing a problem to its source.” It can help you figure out what happened, why it hap-

pened, and what steps you can take to prevent it from happening again.

The first step of root cause analysis is to **identify the problem**: what areas of concern arose during the sprint? How and why did you recognize it as a problem?

Next, you'll want to **collect data**: how long has the problem existed? What was its impact?

After that, try to identify the **causal factors**: What events led to the problem? What other problems sprung up because of it?

By figuring out the causal factors, you can move toward nailing down the **root causes** of the problem. We need to go as deep as possible to uncover these root problems. Try asking "why" a few times.

- Why did *x* happen? (Because of *y.*) Why did *y* happen?
- Why is *x* a problem?
- Why did we not see *x* happening until now?
- Why did no one report on *x*?

It feels weird, but you'd be surprised how repeatedly asking "why" digs up root causes that nobody would have thought of.

Finally, figure out the actionables you'll take to prevent the problem from happening again.

Retrospectives: Dealing with leftover work

Your team may not close every issue attached to your Milestone –

that's just life. Are they open because your team didn't have enough time, or are they open because they're no longer relevant?

Whatever the answer, don't just leave your leftover work in your sprint backlog. You'd be surprised how often those issues just don't matter anymore because circumstances have changed.

If work is still relevant, look at the issue title (user story) and details (like acceptance requirements). Does the issue need to be re-written or clarified? If you're confident everyone still understands the goal, save yourself the time and leave it as-is.

Most importantly, make sure you understand why the work wasn't done. Did you over-estimate your velocity? Is there disagreement with the original time estimate attached to the issue? Unless it's really unclear, it's probably fine to leave this task to the product owner.

Pat yourself on the back

You did it! Not only did you properly prepare for your sprint, you successfully saw it through to the end, *and* you took the time to reflect and make sure future sprints go even further.

Now get ready to do it all again.

About the authors

Matt Butler is ZenHub's CEO and co-founder. As a software developer, then later as a project manager for Barclays Investment Bank, he became deeply familiar with the pain of standard project management tools. These experiences fueled him to develop a better way for teams to work together.

Paige Paquette is ZenHub's head of marketing, which means she spends a lot of time thinking (and writing) about what goes into an exceptional software team. She led marketing at the award-winning venture studio Axiom Zen before joining ZenHub's founding team.

With assistance from: Bryce Bladon is an award-winning writer and communications expert. Bryce believes in short sentences, big ideas, and a lexicon everyone can understand. His past clients include specialist development studios, award-winning design firms, and at least one federal government. He's also editor-in-chief of Clients from Hell.

APPENDIX

Everything else you need to know

Agile project management

We talked a lot about agile methodology, but we tried to make our teachings as concise as possible. If you're looking for a more in-depth guide, we have a few favorites. Here are the resources that inspired the writing of this book.

- Martin Fowler's (Free) Guide to Agile Software Development
- The Agile Samurai (A near-perfect intro to agile). Rasmusson's blog is also worth checking out.
- Coaching Agile Teams: A Companion for ScrumMasters, Agile Coaches, and Project Managers in Transition

The philosophy of ZenHub is underscored by a belief that developers need to avoid distraction at all costs.

- Steve Fenton's examination of the true cost of team interruptions
- Fast Company's own take on context switching

Customizing task boards

When it comes time to customize your task board, labels are a great way to communicate important information about tasks in a concise way.

- Creating a Label Style Guide
- Labelling Best Practices

Creating Quality GitHub Issues

Issue standardization can ultimately save you time and ensure a consistent foundation for each new user story.

- How to create an issue template in GitHub

Identifying trends – and ensuring your team's process evolves in an informed manner – is done by gathering data. ZenHub's RESTful API provides data on issue estimates, history, and task board information.

- Master your team's data using ZenHub's API

Product Backlogs

A quality backlog is the foundation of a quality product. We use the DEEP approach at ZenHub.

- Learn more about the DEEP method to product backogs

Milestones and Epics

Software Estimates

Unsure about the team-based approach to assessing your estimates?

- Learn more about The Wisdom of the Crowd

Sprints

Gearing up for your first sprint?

- TechBeacon's First Agile Sprint Survival Guide.

Team Meetings, Retrospectives, and Reviews

Jono Bacon, former community lead at Ubuntu and GitHub, offers advice for recruiting, motivating, and managing your software team.

- Learn about The Art of Community

Daily standup meetings are a great way to keep your team accountable and on track, but ensure you're doing it right.

- Avoid these common mistakes during the daily stand-up meeting

When it's time for code review, make pull requests into opportunities to teach and learn.

- We recommend setting up Pull Request templates for consistency.

During a retrospective, it's important that your offer support while actively challenging assumptions.

- Kim Scott's Guide to Radical Candor - Kim Scott's presentation on Radical Candor in video form

More on GitHub

For more on GitHub, we recommend:

- GitHub's Help Section
- Resources for Learning Git and GitHub

About ZenHub

ZenHub is how the world's best development teams work together – from flight engineers at NASA, to teams at Docker, Rackspace, Adobe, and Starbucks.

For managers

Fed up with guesswork? ZenHub tracks work where work happens, providing more accurate data and unparalleled insight into your technical organization. Never ask for a status update again.

Teams using ZenHub report a 31.5% increase in productivity and a 21% increase in achieved sprint goals. They use GitHub 52% more often, helping you make the most of your existing investment.

For software developers

ZenHub is the best-in-class collaboration solution for development teams. It's natively integrated in GitHub's UI, so wasted time and distraction is eliminated. Developers can stop jumping from tool to tool and focus on what really matters – shipping amazing products.

Visit ZenHub.com to download ZenHub's browser extension free.

Using GitHub Enterprise? Start a free evaluation of ZenHub Enterprise, our on-premise tool used by the Global 1000.

Want even more? Check out our customer stories.

How the world's best teams work together

zenhub.com