

Hand Over Responsibility

We've made our bets and now it's time to start the next cycle. How does the team get started?

Assign projects, not tasks

We *don't* start by assigning tasks to anyone. Nobody plays the role of the “taskmaster” or the “architect” who splits the project up into pieces for other people to execute.

Splitting the project into tasks up front is like putting the pitch through a paper shredder. Everybody just gets disconnected pieces. We want the project to stay “whole” through the entire process so we never lose sight of the bigger picture.

Instead, we trust the team to take on the entire project and work within the boundaries of the pitch. The team is going to define their own tasks and their own approach to the work. They will have full autonomy and use their judgement to execute the pitch as best as

they can.

Teams love being given more freedom to implement an idea the way they think is best. Talented people don't like being treated like "code monkeys" or ticket takers.

Projects also turn out better when the team is given responsibility to look after the whole. Nobody can predict at the beginning of a project what exactly will need to be done for all the pieces to come together properly. What works on paper almost never works exactly as designed in practice. The designers and programmers doing the real work are in the best position to make changes and adjustments or spot missing pieces.

When teams are assigned individual tasks, each person can execute their little piece without feeling responsible for judging how all the pieces fit together. Planning up front makes you blind to the reality along the way.

Remember: we aren't giving the teams absolute freedom to invent a solution from scratch. We've done the shaping. We've set the boundaries. Now we are going to trust the team to fill in the outline from the pitch with real design decisions and implementation.

This is where our efforts to define the project at the right level of abstraction—without too much detail—will pay off. With their talent and knowledge of the particulars, the team is going to arrive at a better finished product than we could have by trying to determine the final form in advance.

Done means deployed

At the end of the cycle, the team will deploy their work. In the case of a Small Batch team with a few small projects for the cycle, they'll

deploy each one as they see fit as long as it happens before the end of the cycle.

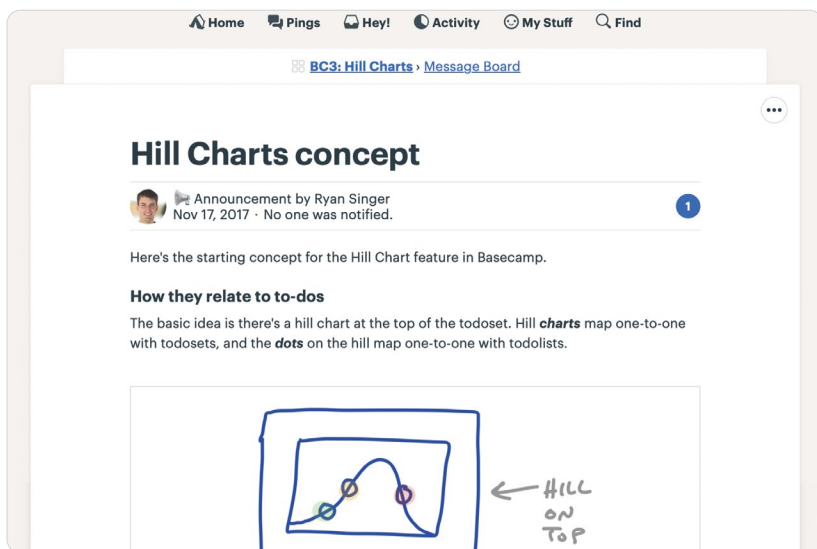
This constraint keeps us true to our bets and respects the circuit breaker. The project needs to be done within the time we budgeted; otherwise, our appetite and budget don't mean anything.

That also means any testing and QA needs to happen *within* the cycle. The team will accommodate that by scoping off the most essential aspects of the project, finishing them early, and coordinating with QA. (More on that later.)

For most projects we aren't strict about the timing of help documentation, marketing updates, or announcements to customers and don't expect those to happen within the cycle. Those are thin-tailed from a risk perspective (they never take 5x as long as we think they will) and are mostly handled by other teams. We'll often take care of those updates and publish an announcement about the new feature during cool-down after the cycle.

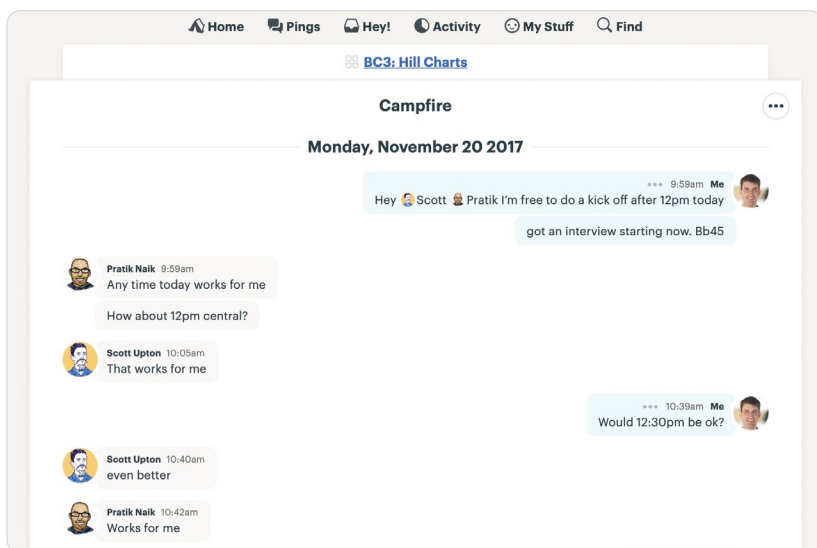
Kick-off

We start the project by creating a new Basecamp project and adding the team to it. Then the first thing we'll do is post the shaped concept to the Message Board. We'll either post the original pitch or a distilled version of it.



The first thing on the Basecamp project is a message with the shaped concept

Since our teams are remote, we use the chat room in the Basecamp project to arrange a kick-off call.



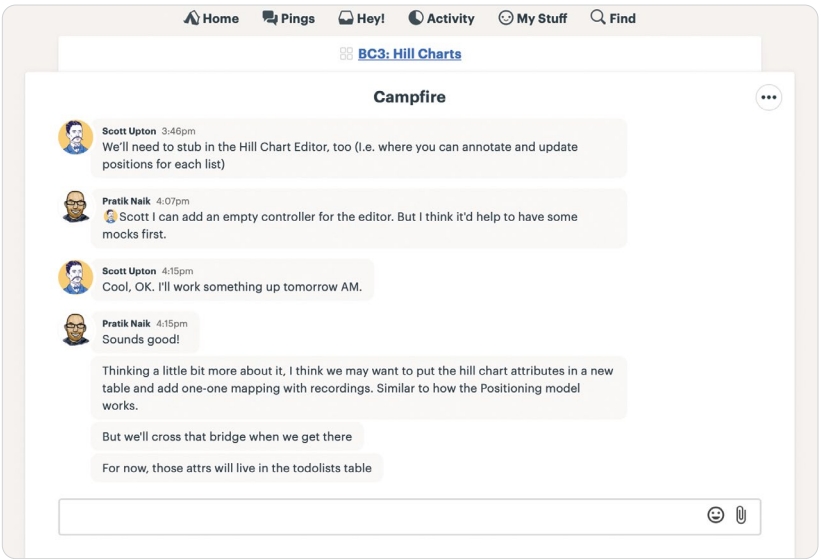
Arranging a call with the team to walk through the shaped work

The call gives the team a chance to ask any important questions that aren't clear from the write-up. Then, with a rough understanding of the project, they're ready to get started.

Getting oriented

Work in the first few days doesn't look like "work." No one is checking off tasks. Nothing is getting deployed. There aren't any deliverables to look at. Often there isn't even much communication between the team in the first few days. There can be an odd kind of radio silence.

Why? Because each person has their head down trying to figure out how the existing system works and which starting point is best. Everyone is busy learning the lay of the land and getting oriented.



The team figuring out where to start

It's important for managers to respect this phase. Teams can't just dive into a code base and start building new functionality immediately. They have to acquaint themselves with the relevant code, think through the pitch, and go down some short dead ends to find a starting point. Interfering or asking them for status too early hurts the project. It takes away time that the team needs to find the best approach. The exploration needs to happen anyway. Asking for visible progress will only push it underground. It's better to empower the team to explicitly say "I'm still figuring out how to start" so they don't have to hide or disguise this legitimate work.

Generally speaking, if the silence doesn't start to break after three days, that's a reasonable time to step in and see what's going on.

Imagined vs discovered tasks

Since the team was given the project and not tasks, they need to come up with the tasks themselves. Here we note an important difference between tasks we *think* we need to do at the start of a project and the tasks we *discover* we need to do in the course of doing real work.

The team naturally starts off with some imagined tasks—the ones they assume they're going to have to do just by thinking about the problem. Then, as they get their hands dirty, they discover all kinds of other things that we didn't know in advance. These unexpected details make up the true bulk of the project and sometimes present the hardest challenges.

Teams discover tasks by doing real work. For example, the designer adds a new button on the desktop interface but then notices there's no obvious place for it on the mobile webview version. They record a new task: figure out how to reveal the button on mobile. Or the

first pass of the design has good visual hierarchy, but then the designer realizes there needs to be more explanatory copy in a place that disrupts the layout. Two new tasks: Change the layout to accommodate explanatory copy; write the explanatory copy.

Often a task will appear in the process of doing something unrelated. Suppose a programmer is working on a database migration. While looking at the model to understand the associations, she might run into a method that needs to be updated for a different part of the project later. She's going to want to note a task to update that method later.

The way to really figure out what needs to be done is to start doing real work. That doesn't mean the teams start by building just anything. They need to pick something meaningful to build first. Something that is central to the project while still small enough to be done end-to-end—with working UI and working code—in a few days.

In the next chapters we'll look at how the team chooses that target and works together to get a fully integrated spike working.



Get One Piece Done

As the team gets oriented, they start to discover and track the tasks they need to do to build the project. It's important at this early phase that they don't create a master plan of parts that should come together in the 11th hour. If the team completes a lot of tasks but there's no "one thing" to click on and try out, it's hard to feel progress. A team can do a lot of work but feel insecure because they don't have anything real to show for it yet. Lots of things are done but nothing is *really* done.

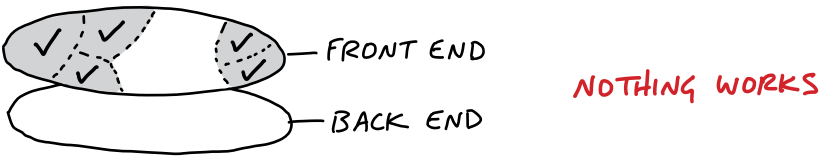
Instead they should aim to make something tangible and demoable early—in the first week or so. That requires integrating vertically on one small piece of the project instead of chipping away at the horizontal layers.

Integrate one slice

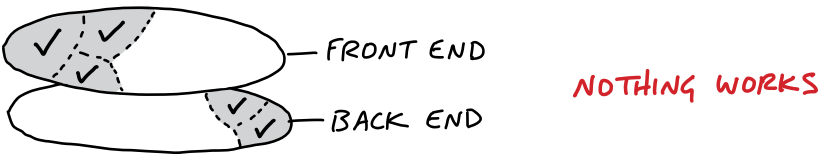
We can think of projects in two layers: front-end and back-end,

design and code. While technically speaking there are more layers than this, these two are the primary integration challenge in most projects.

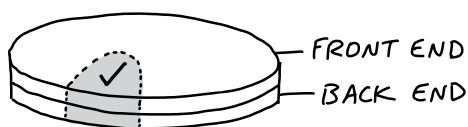
Suppose the project starts with a lot of design. The team could design a variety of screens and even implement them as templates or views. But until they're wired to a backend, nothing does anything. The work remains hypothetical and speculative.



Same with the backend. A lot of tasks could be checked off, but without any UI—what can you do with it? How do you judge if the work on a specific piece of business logic is really right without interacting with it?



What we want instead is to pick off one slice of the project to integrate. Then when that's done, the team has something tangible that they've proven to work (or not work and reconsider). Anyone can click through the interaction and see if the feature does what it should and if what it does is what they want.



SOMETHING WORKS!

Case study: Clients in projects

We built a feature in Basecamp 3 that allowed service firms to invite clients to their projects and share chosen documents, messages, or to-do lists with them. The concept, defined in the pitch, had a variety of moving parts:

- **Client Access:** Before this feature, Basecamp's access model was all or nothing. We needed a way to invite some people to see just some parts of a project. This had major back-end and caching implications.
- **Client Management:** We needed a way to add clients to projects and the ability to manage clients separately from team members.
- **Visibility Toggle:** Each piece of content in a project should have a toggle to expose it to clients or not.

The team had one designer and one programmer. After they got oriented and familiar with how the existing code worked, the designer chose the visibility toggle as the best place to integrate first. This was the most central piece of UI in the project. It's the one that would appear in demo videos and the interaction customers would use most.

The designer didn't make a pixel-perfect mockup. Instead, he experimented with different affordances and placements in the app's HTML templates. Should the toggle be two radio buttons, a

checkbox, or a custom button that changes state?

Meanwhile, the programmer wasn't waiting around. He had enough guidance from the pitch to start spiking the access model.

As soon as the designer felt confident in the basic direction of the UI, he pinged the programmer and showed him the stubbed toggle. Stepping away from the access problem for a bit, the programmer wired the toggle enough so that it would appear on all the supported content types, change state when clicked, and save its state in the database.

At this point, the toggle didn't actually change the visibility of the content. But it worked from the service firm's point of view. The designer could click it, feel it, and judge how well it worked with live data on a staging server.

There was still more design work to do on the toggle. But the programmer didn't need to be involved anymore. With the affordance wired up, the designer could continue to experiment with copy, placement, color, mobile view rendering, and more. Meanwhile, the programmer could get back to the access model or whatever else was most important to tackle next.

About three days after the start of the project, the designer demoed the working toggle to a manager. Their conversation led to a few more tweaks and then they were able to call the toggle "done." One important piece of the project was designed, implemented, demoed, and settled. The team felt good about showing tangible progress. And the team and management both felt confidence in the project by seeing a working piece. By clicking through a core interaction early, they were able to validate that what they hoped would make sense in theory did indeed look right and make sense in practice.

This short example illustrates a few points about how the teams integrate over short periods to finish one piece of the project at a time.

Programmers don't need to wait

Because the important moving parts were already defined in the shaping process, programmers don't need to sit idle waiting for design when the project starts. There's enough direction in the pitch for them to start working on back-end problems from the start. They won't be able to take a piece of functionality to completion without knowing where it leads on the front-end, but there should be enough information in the pitch to inform foundational modeling decisions.

Affordances before pixel-perfect screens

Programmers don't need a pixel-perfect design to start implementing. All they need are endpoints: input elements, buttons, places where stored data should appear. These affordances are the core of a user interface design.

Questions about font, color, spacing, and layout can be resolved after the raw affordances are in place and hooked up in code. Copywriting, basic affordances, and some wiring are all we need to try a live working version in the browser or on the device. Then we can answer the fundamental questions early: Does it make sense? Is it understandable? Does it do what we want?

That means the first interface a designer gives to a programmer can look very basic, like the example below. It's more like a breadboard than a visual design or a polished mock-up.

Please choose an option

☒ Private Room
☐ Dorm Room

Arrival and departure

When do you arrive?
Friday May 24, 2019 in the Afternoon

When do you leave?
Monday May 27, 2019

Your information

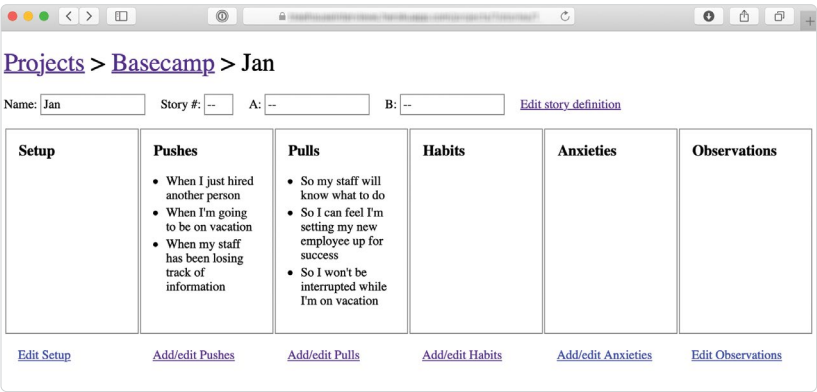
First name

This screenshot is from a registration app for multi-day courses. The designer made it in HTML by hand. There's barely any style — just enough visual hierarchy to feel confident that the layout is usable and amenable to future layers of styling.

While the design looks simple, a lot of decisions are reflected in it.

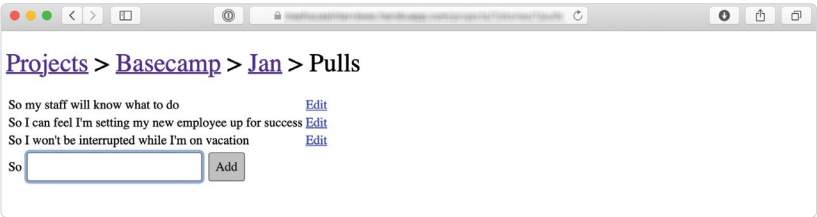
- The decision to ask for arrival time but not departure time came from detailed discussions about the business logic and pricing model.
- The specific options in the arrival time pulldown correspond to rules that had to be worked out about when to charge for meals and overnight stays. The designer's first sketches used a calendar-style date picker for the arrival and departure days. But that led to UX problems. Some courses were long (multiple weeks) with different phases. There wasn't room in a standard calendar-style date picker to label the phases on the day boxes. With a pulldown, she could use option groups to label groups of dates when needed. That way users wouldn't need to reference a schedule elsewhere to be sure they were selecting the right dates.

Here’s another example. This is the first working piece of an app for capturing data from customer interviews.



At this early stage the project name (Basecamp) and interview subject (Jan) were hard-coded and most of the links didn’t go anywhere.

Look at how raw this design is. The actions are plain text links in the default blue and purple browser colors. The boxes containing the data points are barely styled with plain black borders. As rough as it is, this design tests some important trade-offs. The designer chose to show as much data as possible above the fold so it would be easy to review interviews. That didn’t leave enough room within each section for UI to add, edit, or remove data points. That led the designer to create separate screens for adding and editing data per section.



This is the first design for adding and editing “pulls” — a type of data in this interview technique. Again, look at how raw it is. There’s just enough design here to quickly wire it up and test it. The team can click through this to judge whether navigating to a separate screen to record data is acceptable or not. If it works, they can layer on additional styling later. If it doesn’t work, they didn’t waste a lot of time implementing a pixel-perfect design.

Beautiful alignment, colors, and typography don’t matter on the first pass. Visual styling is important in the end product, not in the early stages. The biggest uncertainties are about whether it will work, whether it will make sense, and how hard it will be to implement. After the elements are wired up, they can be rearranged, restyled, and repainted to improve the work that’s already done. First make it work, then make it beautiful.

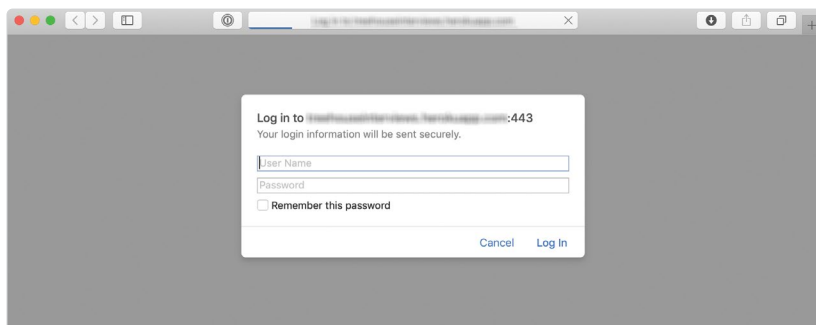
Program just enough for the next step

The same is true for back-end work. It doesn’t have to be all or nothing. Sometimes a designer just needs some scaffolding—a couple fields that save data or some code to navigate from one stubbed screen to another. Other times she needs to populate a variable in the template with a collection of real data so she can iterate on different displays (rows, columns, media boxes, etc) to find the best design.

The early back-end work can be strategically patchy. There might be a controller to render templates but no model. Or a controller and bits of a model with mock data but no support for creating or updating the data. Screens that aren’t wired yet could at least be connected with routes for navigating between them.

When it was time to test the first piece of the interview app, the

team knew there would be sensitive data from real interviews going into it. They needed to protect it with some kind of authentication. Rather than building full username and password support—or even integrating a third-party solution—they just used plain HTTPAuth to hard-code a password.



This allowed the team to try adding data from real interviews very early in the cycle, without slowing down to hook up some authentication code that wasn't going to teach them anything about the problems they were trying to solve.

The point is to create a back-and-forth between design and programming on the same piece of the product. Instead of one big hand-off, take turns layering in affordances, code, and visual styling. Step by step, click through the real working feature-in-progress to judge how it's coming and what to do next.

Start in the middle

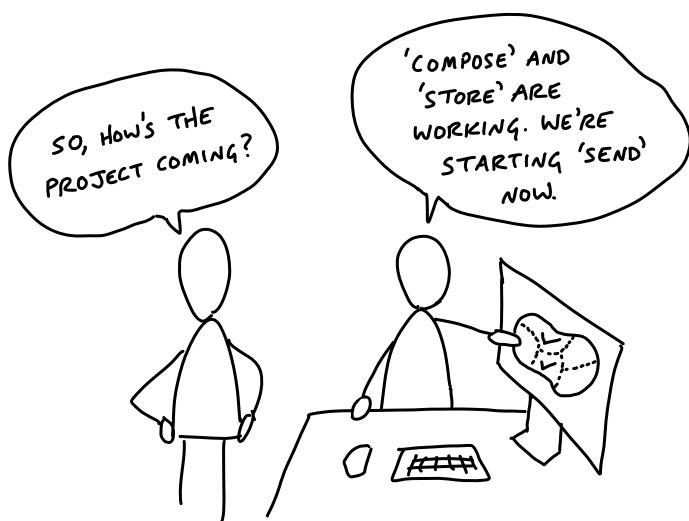
In the examples above, the team didn't build log in first. They didn't build a way to create an interview project and an interview subject before solving the problem of adding interview data. They jumped straight into the middle where the interesting problem was and stubbed everything else to get there.

To expand on this, here are three criteria to think about when choosing what to build first:

First, it should be **core**. The visibility toggle was core to the Clients in Projects concept. Without it, the other work wouldn't mean anything. Contrast that with a more peripheral aspect of the project, like the ability to rename a client. Both were "required," but one was more central and important to prove out early in the cycle. In the interview app, recording interview data was more core—more in the middle—than setting up a new research project.

Second, it should be **small**. If the first piece of work isn't small enough, there isn't much benefit to carving it off from the rest. The point is to finish something meaningful in a few days and build momentum—to have something real to click on that shows the team is on the right track.

Third, it should be **novel**. If two parts of the project are both core and small, prefer the thing that you've never done before. In the Clients in Projects feature, the UI for adding clients was mostly the same as the UI for adding regular users. Starting on that would have moved the project forward, but it wouldn't have taught the team anything. It wouldn't have eliminated uncertainty. Starting with the visibility toggle boosted everyone's confidence because it proved that a new idea was going to work.



Map The Scopes

In the previous chapter, we started the project by finishing one integrated slice early on. That practice belongs to a more general technique that the team can use throughout the project.

Organize by structure, not by person

When asked to organize tasks for a project, people often separate work by person or role: they'll create a list for Designers and a list for Programmers. This leads to the problem we talked about in the previous chapter—people will complete tasks, but the tasks won't add up to a finished part of the project early enough.

To take an example outside of software, consider someone organizing a fundraising event. They could create a list of tasks for each of their three volunteers and track the work that way. But then there'd be no way to see the big picture of how the event is coming together—what's done and what's not done at the macro

level. Instead, they should create lists based on the *structure* of the project—the things that can be worked on and finished independently of each other. To do that, they would create lists for Food Menu, Venue Setup, and Light/Sound. Then the organizer can easily see which areas are done and which areas have outstanding work.

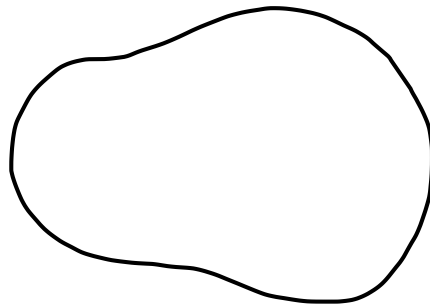
In product development, the categories aren't pre-cut for us. We usually build things we've never built before. Each project is a wild territory that we have to walk through before we can draw a map. By digging into the work, we figure out where the interdependencies are, how things are connected, and what we can slice apart.

As we saw in the previous chapter, the slices of work integrate front-end and back-end tasks. This allows us to finish one slice of the actual project and definitively move on. That's better than having lots of pieces that—fingers crossed—are supposed to come together by the end of the cycle.

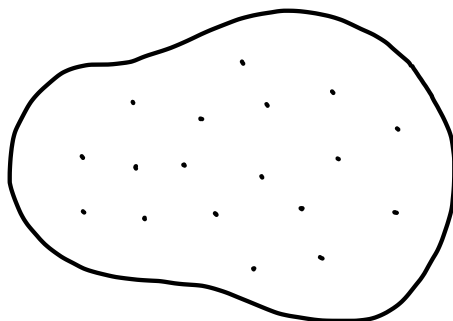
We call these integrated slices of the project scopes. We break the overall scope (singular) of the project into separate scopes (plural) that can be finished independently of each other. In this chapter, we'll see how the team maps the project into scopes and tackles them one by one.

The scope map

Imagine an overhead view of the project. At the beginning, there's just an outline from the shaping work that preceded the project. There aren't any tasks or scopes yet.

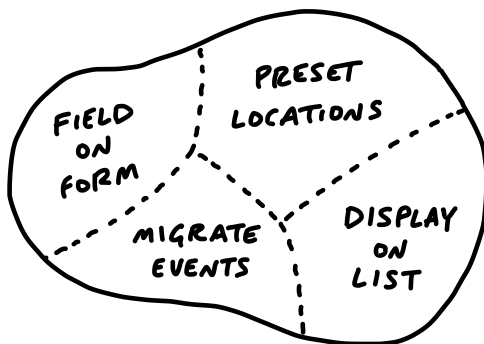


When the team members take over the project, they start discovering tasks. Tasks are a natural starting point because they're concrete and granular. It's too early to organize them into higher level categories. It would be artificial to try and group them arbitrarily. It's enough at the start just to capture a variety of things that need to happen.



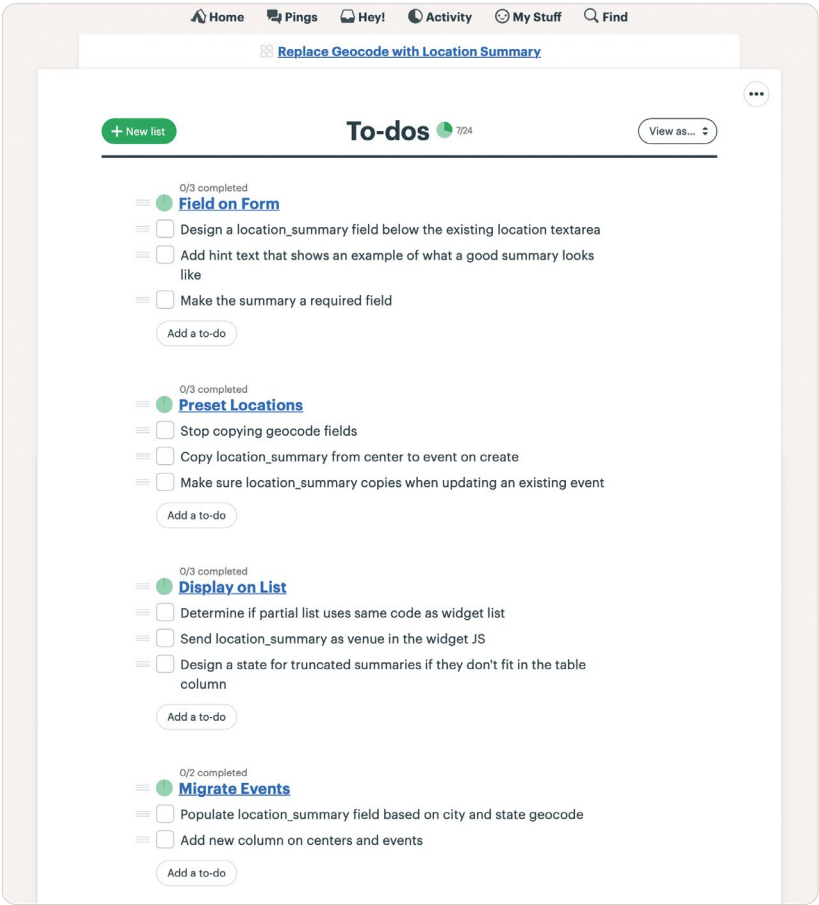
But we don't want to stay with this picture for long. It's too low-level. There's nothing visible from high altitude.

As the team starts doing real work on the project they learn how the tasks are related and what the structure of the project is really like. Then they become able to factor the project into scopes. This is like dividing the map of the project into separate territories.



The scopes reflect the meaningful parts of the problem that can be completed independently and in a short period of time—a few days or less. They are bigger than tasks but much smaller than the overall project.

The map is a mental image. In practice, we define and track the scopes as to-do lists. Each scope corresponds to a list name. Then any tasks for that scope go in that list.



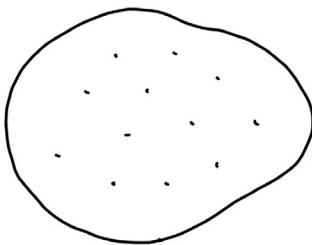
The language of the project

Scopes are more than just slices. They become the language of the project at the macro level. When we were building the *Clients in Projects* feature, the team used the language of the scopes like this: “After *Bucket Access* is done we can implement *Invite Clients*. Then we’ll *Update Recording Visibility* when people on the firm flip the *Visibility Toggle*.”

When it’s time to report status, the team uses the language of scopes to explain what’s done and what’s not done. It’s more satisfying to have the conversation at a high level and point to finished pieces of software, instead of going down into the weeds and defending the purposes and status of individual outstanding tasks. (We’ll see more in the next chapter about how to report on scopes using the Hill Chart.)

Case study: Message drafts

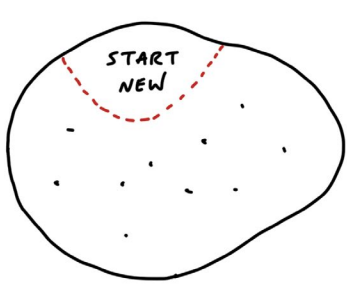
A designer and programmer were building a feature to create and save drafts of messages in a new app. After kick-off, they identified a bunch of tasks they would need to do at some point.



- ☒ **Unscoped**
- ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ☐ Handle draft message timestamps after sending
- ☐ Hook up Send from Draft edit state
- ☐ Remember draft content when editing draft
- ☐ Remember addresses when editing draft
- ☐ Hook up re-saving from Draft edit state
- ☐ Reduce duplication in Draft forms
- ☐ Design index of existing Drafts
- ☐ Hook up Draft deletion from Draft edit state
- ☐ Design "Only you can see this draft" wrapper/labels
- ☒ Allow "invalid" drafts to be created (without an addressee, for example)
- ☒ Hook up manual Draft creation

As the end of the first week approached, they had completed some of the tasks, but there wasn't anything to show for their work. In the spirit of "get one piece done" they focused on one key interaction they could integrate: creating a new draft.

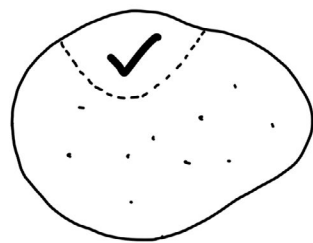
They called the new scope "Start New," created a to-do list for it, and moved to-dos into it. There was only one design task left for them to consider this scope finished.



- ☒ **Start New**
- ☐ Design "Only you can see this draft" wrapper/labels
- ☒ Allow 'invalid' drafts to be created (without an addressee, for example)
- ☒ Hook up manual Draft creation

- ☒ **Unscoped**
- ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ☐ Handle draft message timestamps after sending
- ☐ Hook up Send from Draft edit state
- ☐ Remember draft content when editing draft
- ☐ Remember addresses when editing draft
- ☐ Hook up re-saving from Draft edit state
- ☐ Reduce duplication in Draft forms
- ☐ Design index of existing Drafts
- ☐ Hook up Draft deletion from Draft edit state

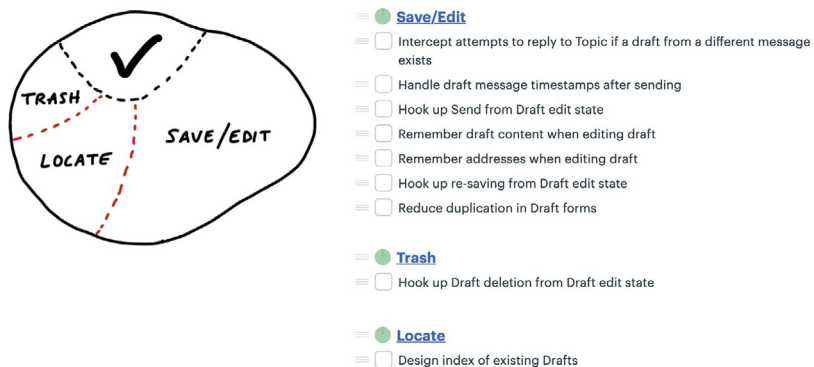
After finishing the one design task, the scope was complete.



- ☒ **Unscoped**
- ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ☐ Handle draft message timestamps after sending
- ☐ Hook up Send from Draft edit state
- ☐ Remember draft content when editing draft
- ☐ Remember addresses when editing draft
- ☐ Hook up re-saving from Draft edit state
- ☐ Reduce duplication in Draft forms
- ☐ Design index of existing Drafts
- ☐ Hook up Draft deletion from Draft edit state

The unscoped tasks that are left don't represent all the work that remains. More tasks are going to be discovered as they start working on each of those. Still, there is enough variety in the work to tease out more scopes. The team was motivated to break out the scopes already at this point because they knew they wanted their efforts to add up to another visible piece being finished before long.

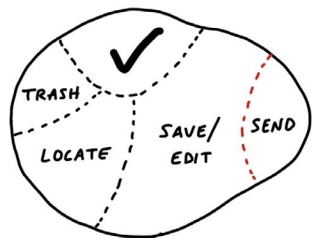
Taking a look at the tasks that were left, they decided to pull out tasks related to finding the drafts into a new scope called Locate and the task for deleting into a scope called Trash. The work that was left all seemed related to saving and editing the draft, so they called that Save/Edit.



Take a look at the Locate scope. There's only one task there right now. But surely there will be more work to do than just designing the index. When there are implementation tasks to do, that's where they'll go.

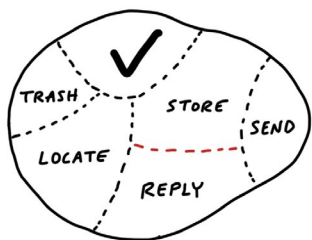
The designer started some work on Locate while the programmer focused on Save/Edit. As she dug into it, she noticed she could carve off a couple pieces to make more visible progress. There were really three scopes in it.

First she factored out the work related to sending the drafted message. She called that Send.



- **Send**
 - Hook up Send from Draft edit state
 - Handle draft message timestamps after sending
- **Save/Edit**
 - Intercept attempts to reply to Topic if a draft from a different message exists
 - Remember draft content when editing draft
 - Remember addresses when editing draft
 - Hook up re-saving from Draft edit state
 - Reduce duplication in Draft forms
- **Trash**
 - Hook up Draft deletion from Draft edit state
 - Design a way to trash drafts from the index of drafts
- **Locate**
 - Design index of existing Drafts
 - Design a way to navigate to Drafts via "Inbox..." menu

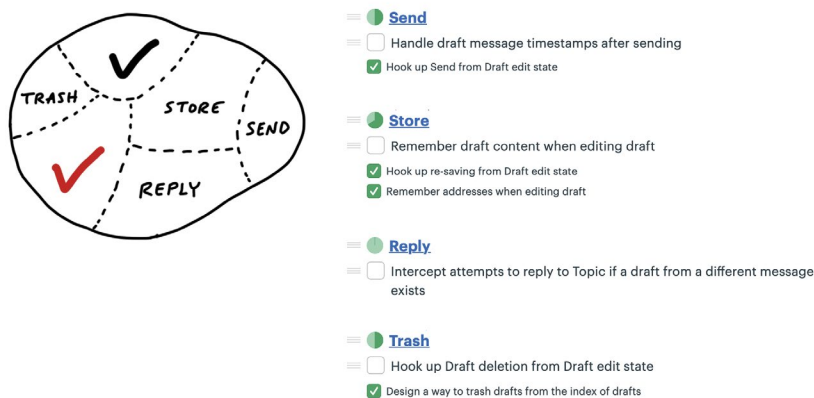
Finally, some of the remaining Save/Edit tasks were about storing information and one other was actually unrelated—it was a special case for handling drafts when replying to another message. She broke these out into two new scopes: Store and Reply.



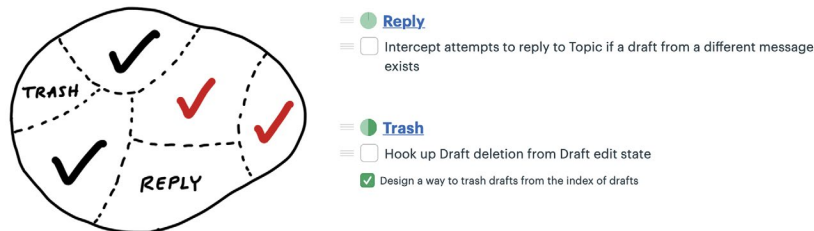
- **Send**
 - Hook up Send from Draft edit state
 - Handle draft message timestamps after sending
- **Store**
 - Remember draft content when editing draft
 - Remember addresses when editing draft
 - Hook up re-saving from Draft edit state
- **Reply**
 - Intercept attempts to reply to Topic if a draft from a different message exists
- **Trash**
 - Hook up Draft deletion from Draft edit state
 - Design a way to trash drafts from the index of drafts
- **Locate**
 - Design index of existing Drafts
 - Design a way to navigate to Drafts via "Inbox..." menu

At this point the team suddenly felt like they could see the whole of the project at a high level. All the major parts were visible at the macro level as scopes. None of them were so big that important or challenging tasks could hide inside of them unnoticed.

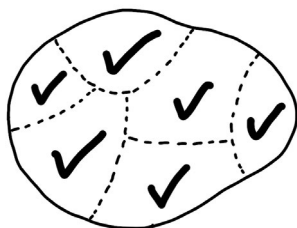
Meanwhile, the designer had made progress on Locate. After a little wiring, they were able to mark that done. Tasks were getting done on Send and Store as well.



Once Send and Store were finished, just a couple tasks remained for Trash and Reply.



And then the project was done.



Discovering scopes

Scope mapping isn't planning. You need to walk the territory before you can draw the map. Scopes properly drawn are not arbitrary groupings or categories for the sake of tidiness. They reflect the real ground truth of what can be done independently—the underlying interdependencies and relationships in the problem.

Scopes arise from interdependencies. The way parts depend on each other determines when you can say a given piece of the work is “done.” You don't know what the work and interdependencies actually are in advance. We talked earlier about imagined versus discovered tasks. The same principle applies to scopes. The scopes need to be discovered by doing the real work and seeing how things connect and don't connect.

That's why at the start of a project, we don't expect to see accurate scopes. We're more likely to see them at the end of week one or start of week two, after the team has had a chance to do some real work and find the natural dividing lines in the anatomy of the problem.

It's also normal to see some shuffling and instability in the scopes at first. The lines get redrawn or scopes renamed as the team feels out where the boundaries really are, like in the example above. The

team was focused on specific problems of saving and editing drafts, so it was easiest to identify that scope early. It wasn't until they got into the weeds that they noticed there were tasks specifically about sending the draft and made that a separate scope.

How to know if the scopes are right

Well-made scopes show the anatomy of the project. When you feel a pain in your body, you don't have to question whether it's in your arms or your legs or your head. You know the parts and their names so you can explain where the pain is. In the same way, every project has a natural anatomy that arises from the design you want, the system you're working within, and the interdependencies of the problems you have to solve.

Three signs indicate when the scopes are right:

1. You feel like you can see the whole project and nothing important that worries you is hidden down in the details.
2. Conversations about the project become more flowing because the scopes give you the right language.
3. When new tasks come up, you know where to put them. The scopes act like buckets that you can easily lob new tasks into.

On the other hand, these three signs indicate the scopes should be redrawn:

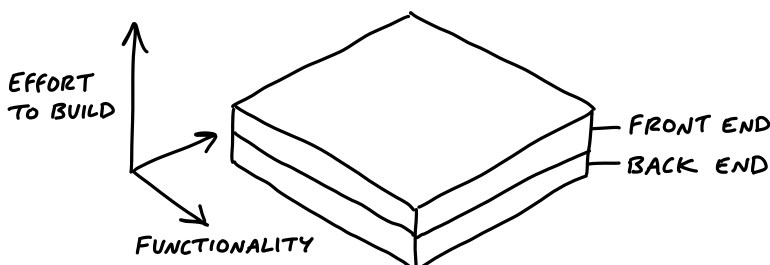
1. It's hard to say how "done" a scope is. This often happens when the tasks inside the scope are unrelated. If the problems inside the scope are unrelated, finishing one doesn't get you closer to finishing the other. It's good in this case to look for something you can factor out, like in the Drafts example.

2. The name isn't unique to the project, like "front-end" or "bugs." We call these "grab bags" and "junk drawers." This suggests you aren't integrating enough, so you'll never get to mark a scope "done" independent of the rest. For example, with bugs, it's better to file them under a specific scope so you can know whether, for example, "Send" is done or if you need to fix a couple bugs first before putting it out of mind.
3. It's too big to finish soon. If a scope gets too big, with too many tasks, it becomes like its own project with all the faults of a long master to-do list. Better to break it up into pieces that can be solved in less time, so there are victories along the way and boundaries between the problems to solve.

Let's close this chapter with a few tips for dealing with different kinds of tasks and scopes that will come up.

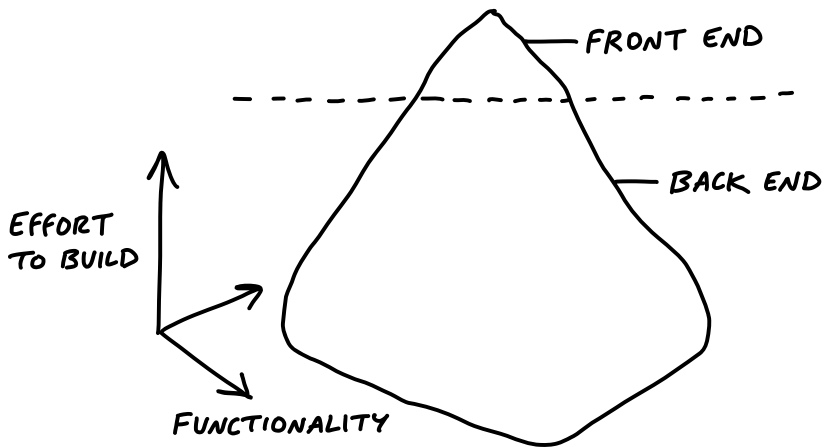
Layer cakes

Most software projects require some UI design and a thin layer of code below. Think of a database app where all you need to do is enter information, save it, and display it back. Work like this looks like a layer cake: You can judge the work by UI surface area because the back-end work is thin and evenly distributed. In these cases, you can integrate all design and programmer tasks together in the same scope. This is a good default for most "information system" type apps.



Icebergs

But sometimes there is significantly more back-end work than UI work or vice versa. For example, a new feature that only requires submitting a form could require very complex business logic to return the right answer. This kind of work is like an iceberg.



For icebergs, it can help to factor out the UI as a separate scope of work (assuming the UI isn't interdependent with the back-end complexity). If the back-end is complex enough, you can split it into separate concerns and then turn those into scopes as well. The goal in cases like this is to define some different things you can finish and integrate in stages, rather than waiting until the 11th hour with fingers crossed that it will all come together.

You also sometimes see upside-down icebergs, where there is a ton of UI complexity with less back-end complexity. For example, the data model for a calendar isn't complicated, but the interaction for rendering a multiple-day event and wrapping across grid cells could take a lot of time and problem-solving.

For both back-end and front-end icebergs, we always question them before accepting them as a fact. Is the complexity really necessary and irreducible? Do we need that fancy UI? Is there a different way to build that back-end process so it has fewer interdependencies with the rest of the system?

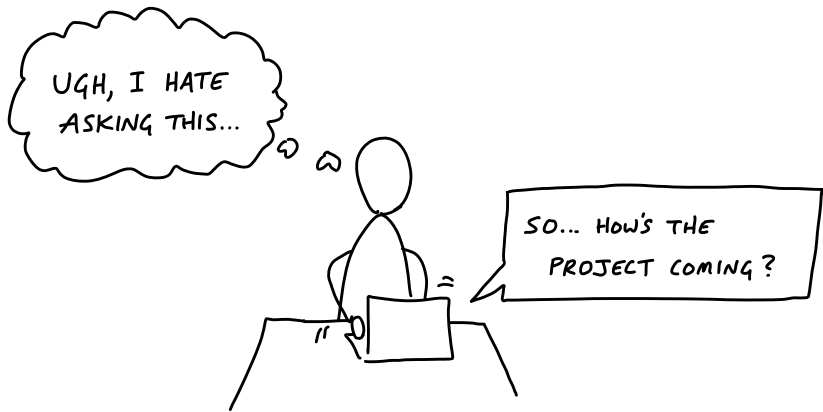
Chowder

There are almost always a couple things that don't fit into a scope. We allow ourselves a "Chowder" list for loose tasks that don't fit anywhere. But we always keep a skeptical eye on it. If it gets longer than three to five items, something is fishy and there's probably a scope to be drawn somewhere.

Mark nice-to-haves with ~

New tasks constantly come up as you get deeper into a problem. You'll find code that could be cleaned up, edge cases to address, and improvements to existing functionality. A good way to deal with all those improvements is to record them as tasks on the scope but mark them with a ~ in front. This allows everyone on the team to constantly sort out the must-haves from the nice-to-haves.

In a world with no deadlines, we could improve everything forever. But in a fixed time box, we need a machete in our hands to cut down the constantly growing scope. The ~ at the start of an item, or even a whole scope, is our best tool for that. We'll come back to this technique when we talk about making cuts to scope in Chapter 14, *Decide When to Stop*.



Show Progress

Good-hearted managers don't like asking for status. It's awkward, feels like nagging, and gets even worse when they have to ask follow-up questions to get sufficiently clear about what's going on.

Managers would rather be able to see the status themselves whenever they need to. We saw in the last chapter how organizing to-dos into scopes helps the team to stay on top of the work. But this doesn't help the manager directly. There are a couple problems with to-dos that make them insufficient for judging status.

The tasks that aren't there

Consider a list with a few completed items and no incomplete items left. This could mean that all the work is done. But it could also mean that the team knows there's more work but hasn't defined tasks yet.

Sometimes a team will define a scope early in the project without populating it with tasks. It marks that some work needs to be done but that actual tasks haven't been discovered yet.

Or think about doing some QA at the end of a scope. All the tasks are done. There's nothing else to do. Then the act of testing populates the scope with new tasks for the issues found.

This goes back to the notion of imagined versus discovered tasks. In our naive notion of a list that's planned up-front, somebody populates it with items that are gradually checked off. In real life, issues are discovered by getting involved in the problem. That means *to-do lists actually grow as the team makes progress.*



If we tried to judge at t_2 how far along the project is, we'd be misled. From an outsider's perspective, there's no way to know whether the number of outstanding tasks will go down or up. To know that, you'd need more context on the work inside the scope to understand what it means that those particular tasks are done and whether others might still be coming.

Estimates don't show uncertainty

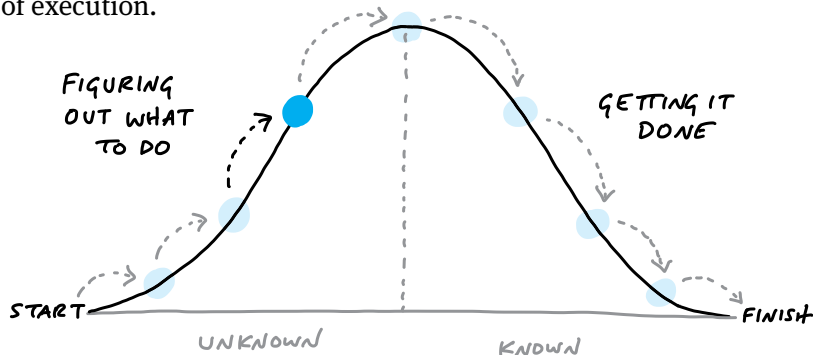
Some teams try to attach estimates to their tasks or scopes to report status. The problem with estimates is they have a very different meaning depending on the nature of the work being estimated.

Say you have two tasks, both estimated to take four hours. If one task is something the team has done ten times in the past, you can be confident in the estimate. Suppose the other task is something the team has never done before, or it has unclear interdependencies. It could take the four hours if all goes perfectly, but due to the unknowns in it, it could stretch out to two to three days. It's not meaningful to write "4 hours, or maybe 3 days" as the estimate.

Recognizing this, we came up with a way to see the status of the project without counting tasks and without numerical estimates. We do that by shifting the focus from what's done or not done to what's unknown and what's solved. To enable this shift, we use the metaphor of the hill.

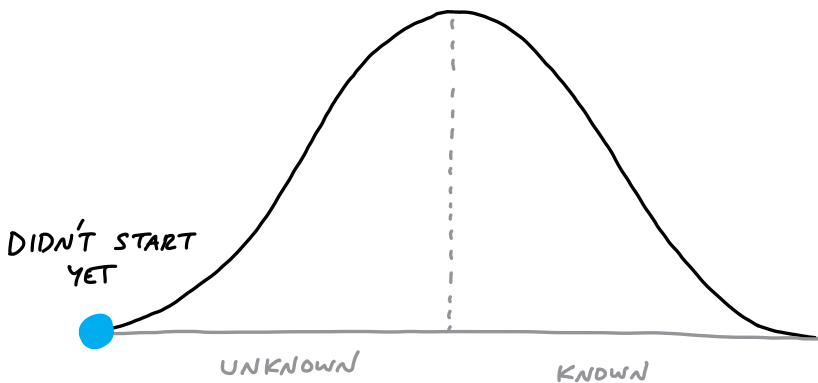
Work is like a hill

Every piece of work has two phases. First there's the uphill phase of figuring out what our approach is and what we're going to do. Then, once we can see all the work involved, there's the downhill phase of execution.



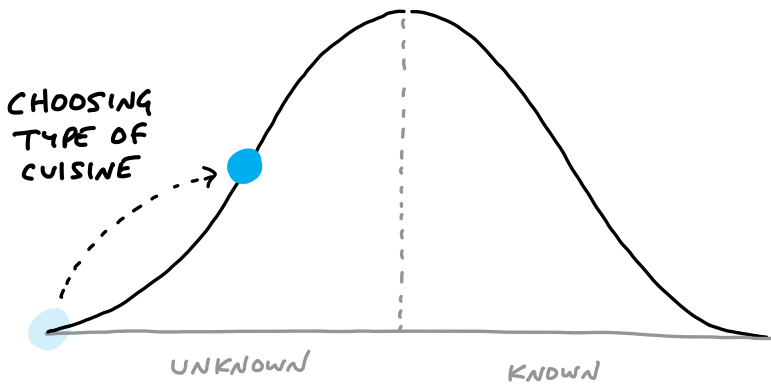
Let's use an everyday example to get the feeling of the hill.

Suppose you're planning to host a dinner party. You've set the date, but it's still a couple weeks ahead and you haven't thought about what to cook yet. You have no idea what type of cuisine the meal will be or what dish to make. That would place you at the start of the hill on the bottom-left.



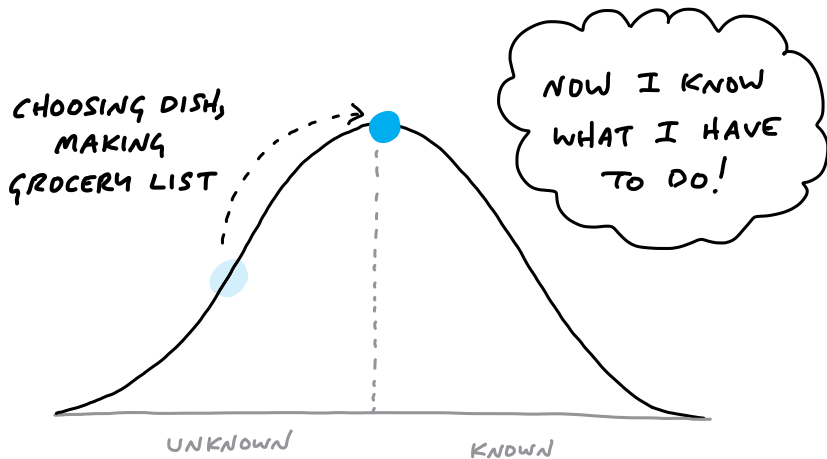
Next you think about who's attending and note that a couple people are vegetarian. That eliminates some options (like grilling out) but still leaves a lot of options open. You consider both Italian and Indian. You think Indian might be more fun to cook, with more interesting vegetarian options. So you decide to look for Indian recipes.

At this point, the question "What percent complete is the project?" doesn't even make sense. And if someone asked you to estimate how long the shopping and prep will take, you couldn't answer that either because you haven't chosen a dish yet. The answer would be: "I've done some work to figure out what kind of cuisine, but I haven't narrowed it down to a specific dish yet." We can represent that by putting you halfway up the "figuring it out" side of the hill.



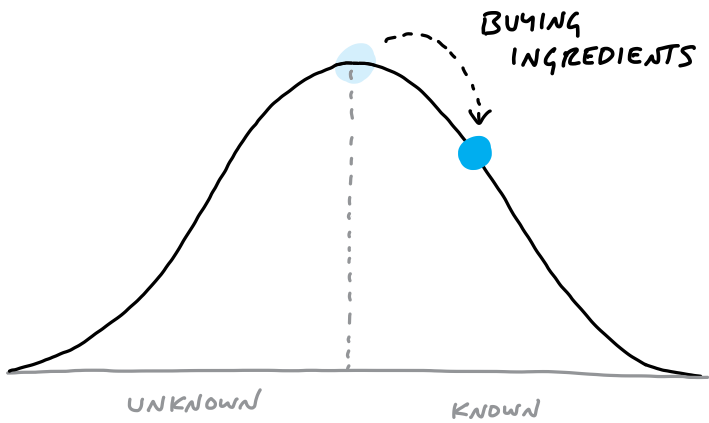
Next you do some searching online and look through your recipe books. You want to find a recipe that will be interesting but doesn't require ingredients that will be too hard to find. You settle on a recipe and prepare a shopping list.

Now you are in a very different position than before. The feeling changes from "I'm still not sure what I'm doing" to "Now I know what to do." You're at the top of the hill.

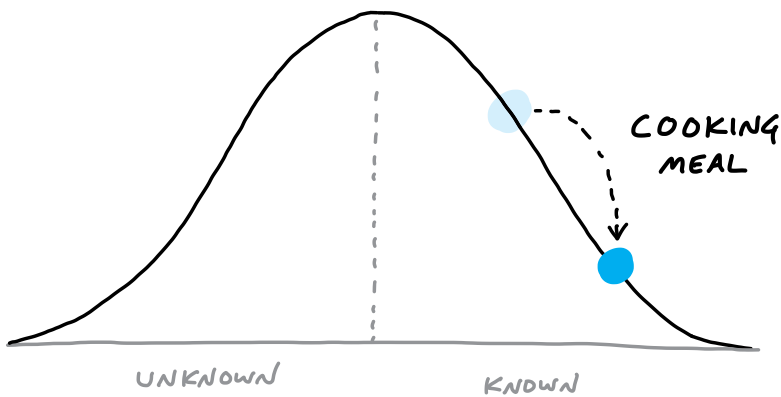


From this vantage point, you can see all of the steps that are left. It's even fair to estimate how long all the work will take ("Let's see...an hour to grocery shop, 30 minutes of prep, cook for 45 minutes...").

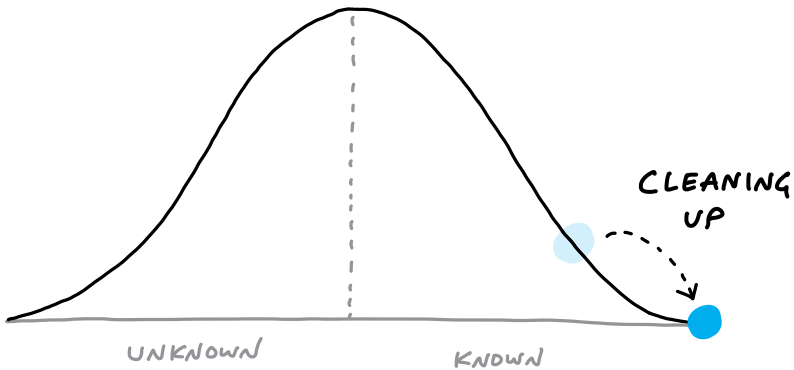
The day before the dinner party, you go to the grocery store and buy the ingredients. This moves you downhill. You're closer to finishing the task.



Next comes the work of prepping and cooking the meal.



After the meal is over, there's just a little work left: the clean-up.



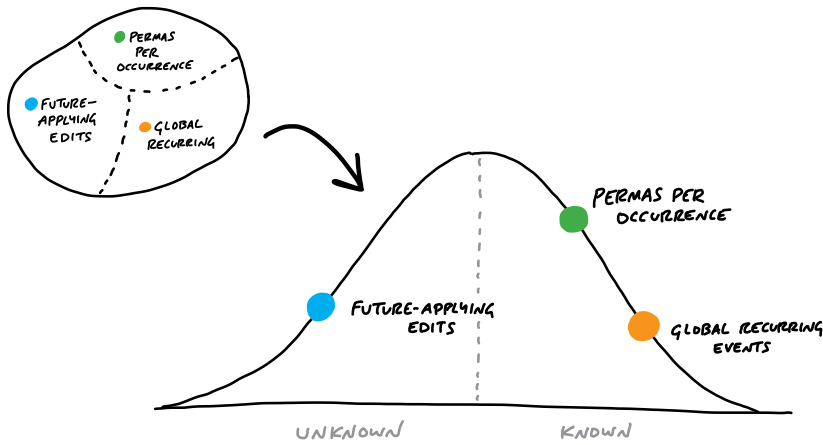
Note how the hill shows how the work *feels* at different stages. The uphill phase is full of uncertainty, unknowns, and problem solving. The downhill phase is marked by certainty, confidence, seeing everything, and knowing what to do.

Scopes on the hill

We can combine the hill with the concept of scopes from the last chapter. The scopes give us the language for the project (“Locate,” “Reply”) and the hill describes the status of each scope (“uphill,” “downhill”).

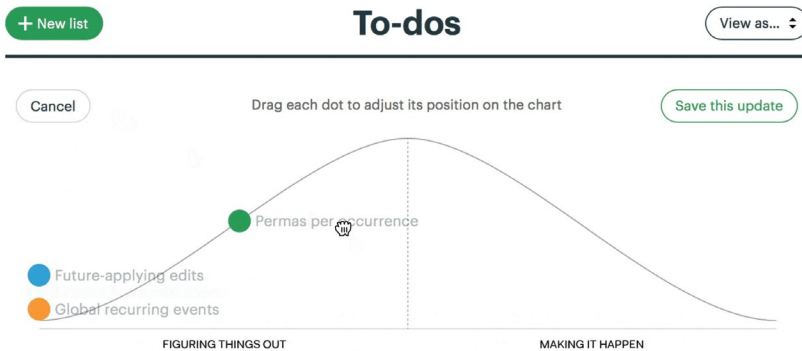
To see the status of the scopes, we can plot each one as a different color on the hill.

This is a snapshot from a project to implement recurring events in Basecamp. Here “Future-applying edits” is a scope that is still being worked out, with significant unknowns to solve. The other two scopes have no meaningful unknowns left, and “Global recurring events” is closer to finished.

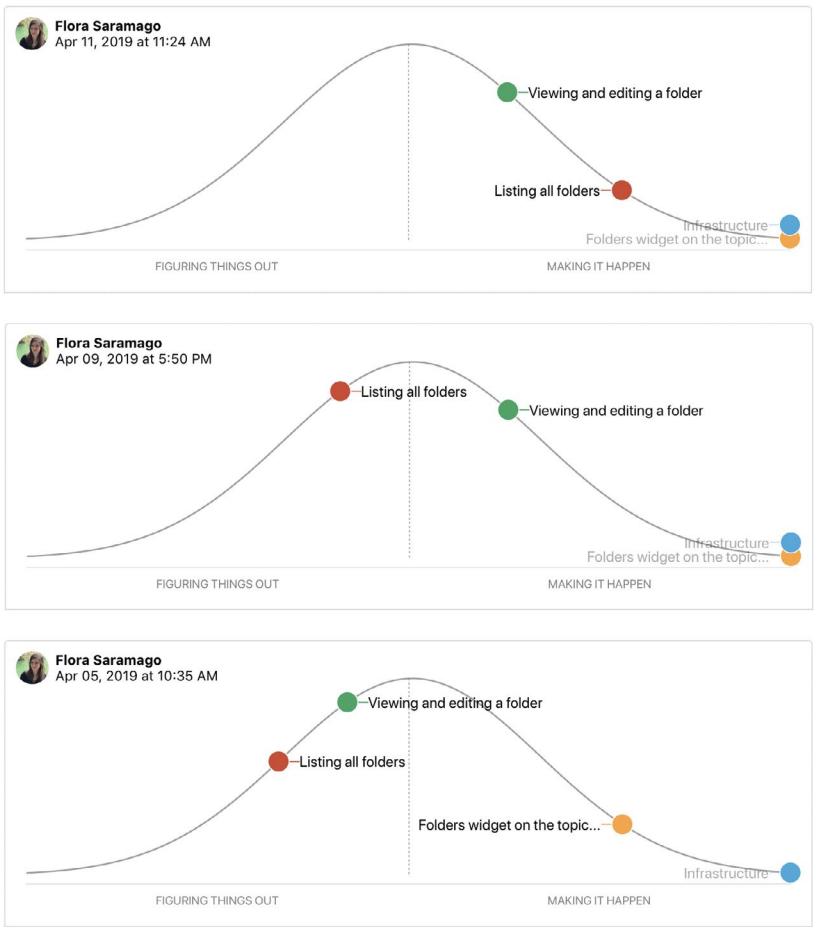


Status without asking

We built a feature exclusive to Basecamp for creating hill charts and updating them with a few clicks. The team members, who have the full context of where the work stands, intuitively drag the scopes into position, and save a new update that's logged on the project (see [How to Implement Shape Up in Basecamp](#)).



For managers, the ability to compare past states is the killer feature. It shows not only where the work stands but how the work is *moving*.



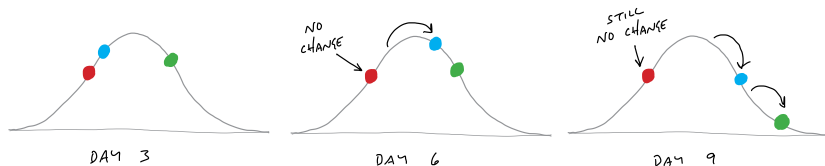
With this second-order view, managers can judge what's in motion and what's stuck. They can see which problems the team chose to solve and how much time they spent at each stage from unknown to known to done.

This report becomes the manager's first destination when they feel anxious about a project. Since it's self-serve, there's no need to interrupt the team with the awkward status question. And in cases where something doesn't look right, the manager can jump directly into a conversation about the relevant piece of work. "Looks like 'Autosave' has been uphill for a while. What's the unknown that's holding it back?" The manager can workshop this specific piece of the project without having to first untangle it from all the other things that are moving along as expected.

Nobody says "I don't know"

Nobody wants to raise their hand to management and say "I don't know how to solve this problem." This causes teams to hide uncertainty and accumulate risk. The moments when somebody is stuck or going in circles are where the biggest risks and opportunities lie. If we catch those moments early, we can address them with help from someone senior or by reworking the concept. If we don't catch them, the unsolved problems could linger so far into the cycle that they endanger the project.

The hill chart allows everybody to see that somebody might be stuck without them actually saying it. A dot that doesn't move is effectively a raised hand: "Something might be wrong here."

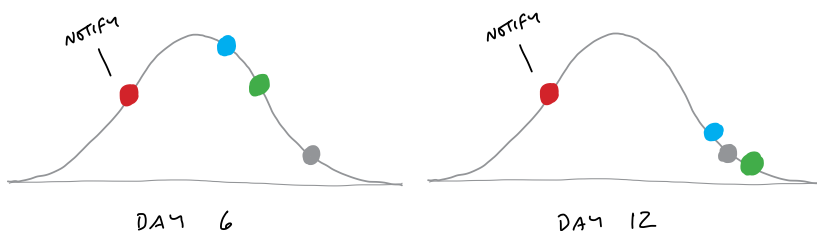


Once it's been spotted, the language of uphill/downhill facilitates the conversation. It's less about the person (Looks like you're stuck!) and more about the work. The question is: What can we solve to get that over the hill?

Prompts to refactor the scopes

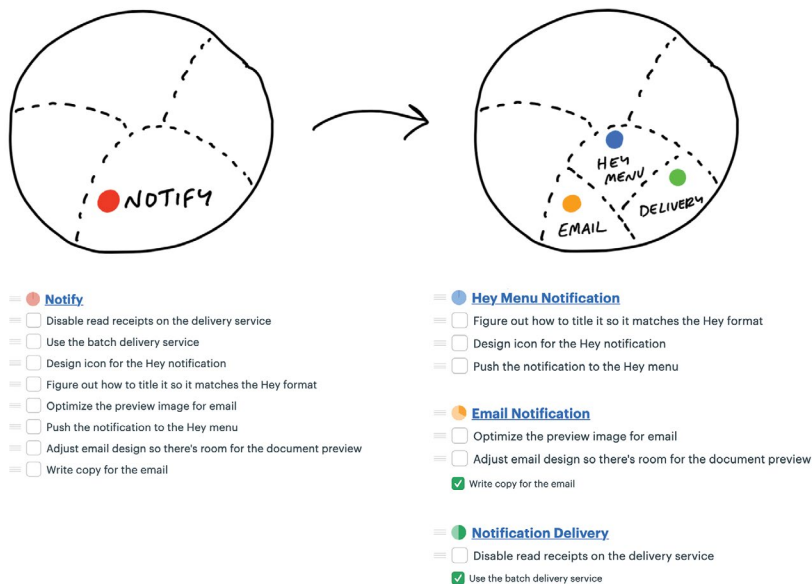
Sometimes probing into a stuck scope reveals that it isn't stuck at all. The problem is in how the lines of the scope were drawn.

Here's a case where the "Notify" scope was stuck on the hill for too long.

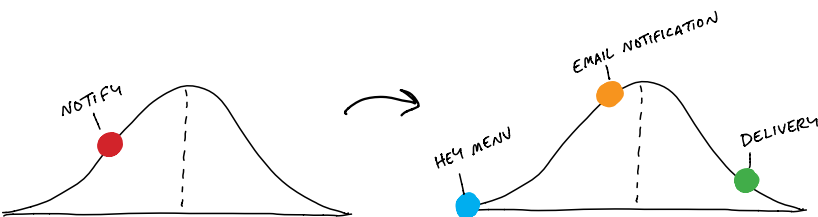


When we checked in with the team, it turned out the work was moving along just fine. The problem was that "Notify" wasn't a single thing. It had three different parts: designing an email, delivering the email in the back-end, and displaying the notification in an in-app menu. The team mostly finished the code for delivering the email. The design of the email was nearly figured out. But they hadn't started on the in-app display. It wasn't possible to say whether "Notify" as a whole was over the hill or not because parts of it were and parts of it weren't.

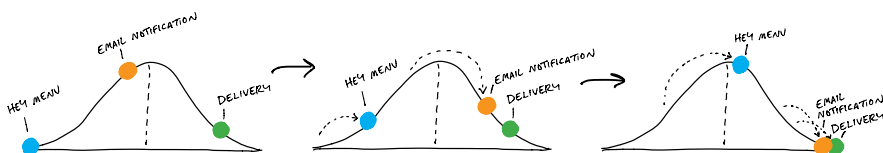
The solution in a case like this is to break the scope apart into smaller scopes that can move independently.



Now the team can move each dot to accurately show where the work stands.



The benefit comes at the second order. With the scopes separated out, they can move independently over time. Now the team can show more progress more frequently than before.



Build your way uphill

Some teams struggle with backsliding when they first try the hill chart. They consider a scope solved, move it the top of the hill, and later have to slide it back when they uncover an unexpected unknown.

When this happens, it's often because somebody did the uphill work with their head instead of their hands. Coming up with an approach in your head is just the first step uphill. We often have a theory of how we'll solve something—"I'll just use that API"—and then the reality turns out to be more complicated. It's good to think of the first third uphill as "I've thought about this," the second third as "I've validated my approach," and the final third to the top as "I'm far enough with what I've built that I don't believe there are other unknowns."

Solve in the right sequence

In addition to seeing where the work stands, we can use the hill chart to sequence the work—which problems to solve in which order.

Some scopes are riskier than others. Imagine two scopes: One involves geocoding data—something the team has never done before. The other is designing and implementing an email notification. Both have unknowns. Both start at the bottom of the hill. This is where the team asks themselves: If we were out of time at the end of the cycle, which of these could we easily whip together—despite the unknowns—and which might prove to be harder than we think?

That motivates the team to push the scariest work uphill first. Once they get uphill, they'll leave it there and look for anything critically important before finishing the downhill work to completion. It's better to get a few critical scopes over the top early in the project and leave the screw-tightening for later.

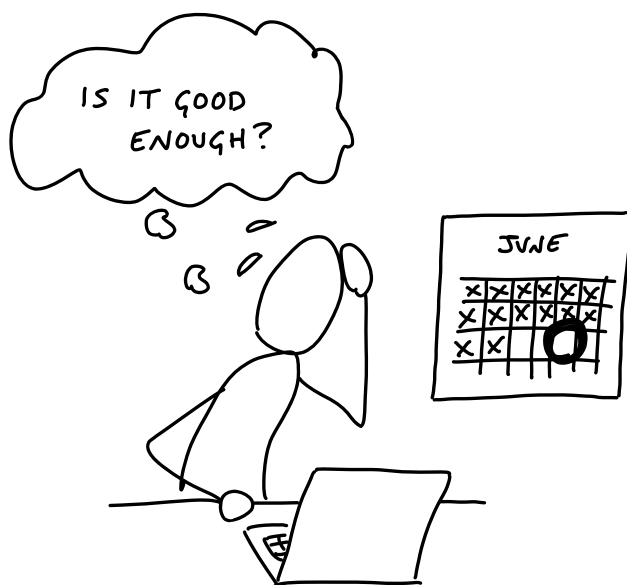
Work expands to fill the time available. If the team starts with the email template first, they could easily spend weeks iterating on copy or creating the ultimate best-ever email design. But they don't *need* to do that. There's some version of an email template that could be worked out in a day during the final week and it would be sufficient. The geocoder, on the other hand, might present novel problems that the team could struggle with for weeks. They don't want that surprise to come at the end of the cycle.

Journalists have a concept called the “inverted pyramid.” The idea is their articles start with the most essential information at the top, then they add details and background information in decreasing order of importance. This allows print newspaper designers to get the crucial part of the story on the front page and cut the end as needed without losing anything essential.

Effective teams sequence their problem solving in the same way. They choose the most important problems first with the most

unknowns, get them to the top of the hill, and leave the things that are the most routine or least worrisome for last.

As the end of the cycle approaches, teams should have finished the important things and left a variety of “nice to haves” and “maybes” lingering around. That brings us to the next chapter, on deciding when to stop.



Decide When to Stop

When the end of the cycle approaches, the techniques we covered so far will put the team in a good position to finish and ship. The shaped work gave them guard rails to prevent them from wandering. They integrated one scope at a time so there isn't half-finished work lying around. And all the most important problems have been solved because they prioritized those unknowns first when they sequenced the work.

Still, there's always more work than time. Shipping on time means shipping something imperfect. There's always some queasiness in the stomach as you look at your work and ask yourself: Is it good enough? Is this ready to release?

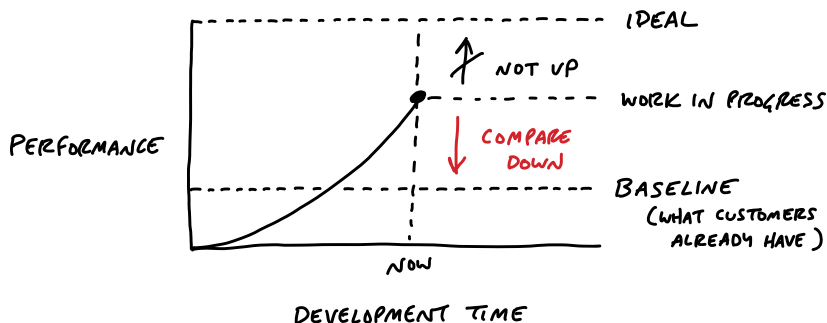
Compare to baseline

Designers and programmers always want to do their best work. It doesn't matter if the button is on the center of the landing page or two pages down a settings screen, the designer will give it their best attention. And the best programmers want the code base to feel like a cohesive whole, completely logically consistent with every edge case covered.

Pride in the work is important for quality and morale, but we need to direct it at the right target. If we aim for an ideal perfect design, we'll never get there. At the same time, we don't want to lower our standards. How do we make the call to say what we have is good enough and move on?

It helps to shift the point of comparison. Instead of comparing up against the ideal, compare down to baseline—the current reality for customers. How do customers solve this problem today, without this feature? What's the frustrating workaround that this feature eliminates? How much longer should customers put up with something that doesn't work or wait for a solution because we aren't sure if design A might be better than design B?

Seeing that our work so far is better than the current alternatives makes us feel better about the progress we've made. This motivates us to make calls on the things that are slowing us down. It's less about us and more about value for the customer. It's the difference between “never good enough” and “better than what they have now.” We can say “Okay, this isn't perfect, but it definitely works and customers will feel like this is a big improvement for them.”



*Make scope cuts by comparing down to baseline
instead of up to some perfect ideal*

Limits motivate trade-offs

Recall that the six-week bet has a circuit breaker—if the work doesn’t get done, the project doesn’t happen.

This forces the team to make trade-offs. When somebody says “wouldn’t it be better if...” or finds another edge case, they should first ask themselves: Is there time for this? Without a deadline, they could easily delay the project for changes that don’t actually deserve the extra time.

We expect our teams to actively make trade-offs and question the scope instead of cramming and pushing to finish tasks. We create our own work for ourselves. We should question any new work that comes up before we accept it as necessary.

Scope grows like grass

Scope grows naturally. Scope creep isn’t the fault of bad clients, bad managers, or bad programmers. Projects are opaque at the macro scale. You can’t see all the little micro-details of a project until you get down into the work. Then you discover not only

complexities you didn't anticipate, but all kinds of things that could be fixed or made better than they are.

Every project is full of scope we don't need. Every part of a product doesn't need to be equally prominent, equally fast, and equally polished. Every use case isn't equally common, equally critical, or equally aligned with the market we're trying to sell to.

This is how it is. Rather than trying to stop scope from growing, give teams the tools, authority, and responsibility to constantly cut it down.

Cutting scope isn't lowering quality

Picking and choosing which things to execute and how far to execute on them doesn't leave holes in the product. Making choices makes the product better. It makes the product better *at some things* instead of others. Being picky about scope *differentiates* the product. Differentiating what is core from what is peripheral moves us in competitive space, making us more alike or more different than other products that made different choices.

Variable scope is not about sacrificing quality. We are extremely picky about the quality of our code, our visual design, the copy in our interfaces, and the performance of our interactions. The trick is asking ourselves which things actually matter, which things move the needle, and which things make a difference for the core use cases we're trying to solve.

Scope hammering

People often talk about “cutting” scope. We use an even stronger word—hammering—to reflect the power and force it takes to repeatedly bang the scope so it fits in the time box.

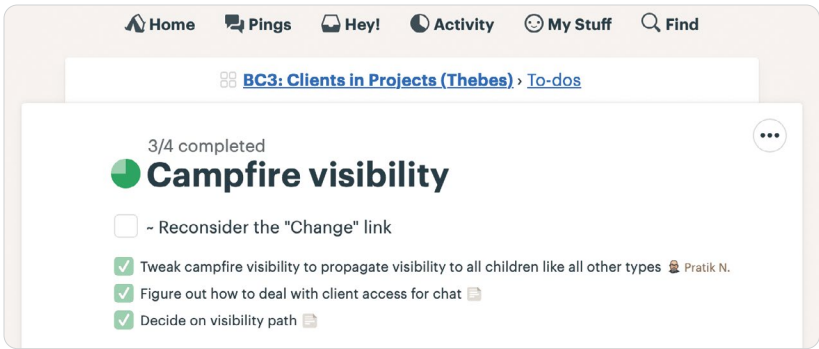
As we come up with things to fix, add, improve, or redesign during a project, we ask ourselves:

- Is this a “must-have” for the new feature?
- Could we ship without this?
- What happens if we don’t do this?
- Is this a new problem or a pre-existing one that customers already live with?
- How likely is this case or condition to occur?
- When this case occurs, which customers see it? Is it core—used by everyone—or more of an edge case?
- What’s the actual impact of this case or condition in the event it does happen?
- When something doesn’t work well for a particular use case, how aligned is that use case with our intended audience?

The fixed deadline motivates us to ask these questions. Variable scope enables us to act on them. By chiseling and hammering the scope down, we stay focused on just the things we need to do to ship something effective that we can be proud of at the end of the time box.

Throughout the cycle, you’ll hear our teams talking about must-haves and nice-to-haves as they discover work. The must-haves are captured as tasks on the scope. The scope isn’t considered “done” until those tasks are finished. Nice-to-haves can linger on a scope after it’s considered done. They’re marked with a tilde (~) in front. Those tasks are things to do if the team has extra time at the end

and things to cut if they don't. Usually they never get built. The act of marking them as a nice-to-have is the scope hammering.



A finished scope with one nice-to-have (marked with a “~”) that was never completed

QA is for the edges

At Basecamp's current size (millions of users and about a dozen people on the product team), we have one QA person. They come in toward the end of the cycle and hunt for edge cases outside the core functionality.

QA can limit their attention to edge cases because the designers and programmers take responsibility for the basic quality of their work. Programmers write their own tests, and the team works together to ensure the project does what it should according to what was shaped. This follows from giving the team responsibility for the whole project instead of assigning them individual tasks (see Chapter 9, Hand Over Responsibility).

For years we didn't have a QA role. Then after our user base grew to a certain size, we saw that small edge cases began to impact hundreds or thousands of users in absolute numbers. Adding the

extra QA step helped us improve the experience for those users and reduce the disproportional burden they would create for support.

Therefore we think of QA as a level-up, not a gate or a check-point that all work must go through. We're much better off with QA than without it. But we don't depend on QA to ship quality features that work as they should.

QA generates discovered tasks that are all nice-to-haves by default. The designer-programmer team triages them and, depending on severity and available time, elevates some of them to must-haves. The most rigorous way to do this is to collect incoming QA issues on a separate to-do list. Then, if the team decides an issue is a must-have, they drag it to the list for the relevant scope it affects. This helps the team see that the scope isn't done until the issue is addressed.

We treat code review the same way. The team can ship without waiting for a code review. There's no formal check-point. But code review makes things better, so if there's time and it makes sense, someone senior may look at the code and give feedback. It's more about taking advantage of a teaching opportunity than creating a step in our process that must happen every time.

When to extend a project

In very rare cases, we'll extend a project that runs past its deadline by a couple weeks. How do we decide when to extend a project and when to let the circuit breaker do its thing?

First, the outstanding tasks must be true must-haves that withstood every attempt to scope hammer them.

Second, the outstanding work must be all downhill. No unsolved

problems; no open questions. Any uphill work at the end of the cycle points to an oversight in the shaping or a hole in the concept. Unknowns are too risky to bet on. If the work is uphill, it's better to do something else in the next cycle and put the troubled project back in the shaping phase. If you find a viable way to patch the hole, then you can consider betting more time on it again in the future.

Even if the conditions are met to consider extending the project, we still prefer to be disciplined and enforce the appetite for most projects. The two-week cool-down usually provides enough slack for a team with a few too many must-haves to ship before the next cycle starts. But this shouldn't become a habit. Running into cool-down either points back to a problem in the shaping process or a performance problem with the team.

Move On

Let the storm pass

Shipping can actually generate new work if you're not careful.

Feature releases beget feature requests. Customers say "Okay, that's great, but what about that other thing we've been asking for?"

Bugs pop up. Suggestions for improvements come in. Everyone is focused on the new thing and reacting to it.

The feedback can be especially intense if the feature you shipped changes existing workflows. Even purely visual changes sometimes spur intense pushback. A small minority of customers might over-react and say things like "You ruined it! Change it back!"

It's important to stay cool and avoid knee-jerk reactions. Give it a few days and allow it to die down. Be firm and remember why you made the change in the first place and who the change is helping.

Stay debt-free

It can be tempting to commit to making changes in response to feedback, but then you no longer have a clean slate for the next cycle. Remember: these are just raw ideas coming in. The way to handle them is with a gentle "no." Saying "no" doesn't prevent you from continuing to contemplate them and maybe shape them up into future projects. Saying "yes," on the other hand, takes away your freedom in the future. It's like taking on debt.

Remember, the thing you just shipped was a six-week bet. If this part of the product needs more time, then it requires a new bet. Let the requests or bugs that just came up compete with everything else at the next betting table to be sure they're strategically important.

Feedback needs to be shaped

Here we come full circle. The raw ideas that just came in from customer feedback aren't actionable yet. They need to be shaped. They are the raw inputs that we talked about in step one of the shaping process: Set Boundaries.

If a request is truly important, you can make it your top priority on the shaping track of the next cycle. Bet on something else for the teams to build and use that time to properly shape the new idea. Then, when the six weeks are over, you can make the case at the betting table and schedule the shaped version of the project for the greatest chance of success.

Conclusion

Key concepts

The Shape Up method presented in this book is tightly interwoven. It may take some thought and experimentation to pull out the right pieces and adapt them to your team.

Whether your team can adopt the method at once or not, I hope that the language and concepts in this book gave you some things to take home immediately:

- Shaped versus unshaped work
- Setting appetites instead of estimates
- Designing at the right level of abstraction
- Concepting with breadboards and fat marker sketches
- Making bets with a capped downside (the circuit breaker) and honoring them with uninterrupted time
- Choosing the right cycle length (six weeks)
- A cool-down period between cycles
- Breaking projects apart into scopes
- Downhill versus uphill work and communicating about unknowns
- Scope hammering to separate must-haves from nice-to-haves

Get in touch

We'd love to hear what you think so we can make the Shape Up method easier to adopt. What did we miss? What still isn't clear? What do you wish we had talked about that we didn't? We'd also love to hear about your successes and challenges as you try to apply it to your teams and projects.

Send us an email at shapeup@basecamp.com.