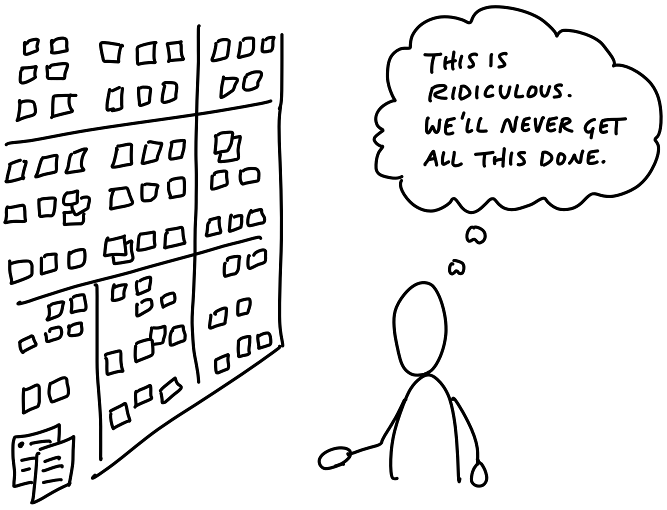


Bets, Not Backlogs

Now that we've written a pitch, where does it go? It doesn't go onto a backlog.

No backlogs

Backlogs are a big weight we don't need to carry. Dozens and eventually hundreds of tasks pile up that we all know we'll never have time for. The growing pile gives us a feeling like we're always behind even though we're not. Just because somebody thought some idea was important a quarter ago doesn't mean we need to keep looking at it again and again.

Backlogs are big time wasters too. The time spent constantly reviewing, grooming and organizing old ideas prevents everyone from moving forward on the timely projects that really matter right now.

A few potential bets

So what do we do instead? Before each six-week cycle, we hold a betting table where stakeholders decide what to do in the next cycle. At the betting table, they look at pitches from the last six weeks — or any pitches that somebody purposefully revived and lobbied for again.

Nothing else is on the table. There's no giant list of ideas to review. There's no time spent grooming a backlog of old ideas. There are just a few well-shaped, risk-reduced options to review. The pitches are potential bets.

With just a few options and a six-week long cycle, these meetings are infrequent, short, and intensely productive.

If we decide to bet on a pitch, it goes into the next cycle to build. If we don't, we let it go. There's nothing we need to track or hold on to.

What if the pitch was great, but the time just wasn't right? Anyone who wants to advocate for it again simply tracks it independently—their own way—and then lobbies for it six weeks later.

Decentralized lists

We don't have to choose between a burdensome backlog and not remembering anything from the past. Everyone can still track pitches, bugs, requests, or things they want to do independently without a central backlog.

Support can keep a list of requests or issues that come up more often than others. Product tracks ideas they hope to be able to shape in a future cycle. Programmers maintain a list of bugs they'd like to fix when they have some time. There's no one backlog or central list and none of these lists are direct inputs to the betting process.

Regular but infrequent one-on-ones between departments help to cross-pollinate ideas for what to do next. For example, Support can tell Product about top issues they are seeing, which Product can then track independently as potential projects to shape. Maybe Product picks off just one of those top issues to work on now. Then, in a future one-on-one, Support can lobby again for something that hasn't yet gotten attention.

This approach spreads out the responsibility for prioritizing and tracking what to do and makes it manageable. People from different departments can advocate for whatever they think is important and use whatever method works for them to track those things—or not.

This way the conversation is always fresh. Anything brought back is brought back with a context, by a person, with a purpose. Everything is relevant, timely, and of the moment.

Important ideas come back

It's easy to overvalue ideas. The truth is, ideas are cheap. They come up all the time and accumulate into big piles.

Really important ideas will come back to you. When's the last time you forgot a really great, inspiring idea? And if it's not that interesting—maybe a bug that customers are running into from time to time—it'll come back to your attention when a customer complains again or a new customer hits it. If you hear it once and never again, maybe it wasn't really a problem. And if you keep hearing about it, you'll be motivated to shape a solution and pitch betting time on it in the next cycle.

WON'T THAT TAKE LONGER THAN SIX WEEKS?
NOT IF WE FOCUS ON THIS ONE USE CASE...

The Betting Table

Now that we have some good potential bets in the form of pitches, it's time to make decisions about which projects to schedule.

Six-week cycles

Committing time and people is difficult if we can't easily determine who's available and for how long. When people are available at different times due to overlapping projects, project planning turns into a frustrating game of Calendar Tetris. Working in cycles drastically simplifies this problem. A cycle gives us a standard project size both for shaping and scheduling.

Some companies use two-week cycles (aka "sprints"). We learned that two weeks is too short to get anything meaningful done. Worse than that, two-week cycles are extremely costly due to the planning overhead. The amount of work you get out of two weeks isn't worth the collective hours around the table to "sprint plan" or the opportunity cost of breaking everyone's momentum to re-group.

This led us to try longer cycles. We wanted a cycle that would be long enough to finish a whole project, start to end. At the same time, cycles need to be short enough to see the end from the beginning. People need to feel the deadline looming in order to make trade-offs. If the deadline is too distant and abstract at the start, teams will naturally wander and use time inefficiently until the deadline starts to get closer and feel real.

After years of experimentation we arrived at six weeks. Six weeks is long enough to finish something meaningful and still short enough to see the end from the beginning.

Cool-down

If we were to run six-week cycles back to back, there wouldn't be any time to breathe and think about what's next. The end of a cycle is the worst time to meet and plan because everybody is too busy finishing projects and making last-minute decisions in order to ship on time.

Therefore, after each six-week cycle, we schedule two weeks for cool-down. This is a period with no scheduled work where we can breathe, meet as needed, and consider what to do next.

During cool-down, programmers and designers on project teams are free to work on whatever they want. After working hard to ship their six-week projects, they enjoy having time that's under their control. They use it to fix bugs, explore new ideas, or try out new technical possibilities.

Team and project sizes

In addition to standardizing the length of our cycles, we also roughly standardize the types of projects and teams that we bet on.

Our project teams consist of either one designer and two programmers or one designer and one programmer. They're joined by a QA person who does integration testing later in the cycle.

These teams will either spend the entire cycle working on one project, or they'll work on multiple smaller projects during the cycle. We call the team that spends the cycle doing one project the big batch team and the team working on a set of smaller projects the small batch team. Small batch projects usually run one or two weeks each. Small batch projects aren't scheduled individually. It's up to the small batch team to figure out how to juggle the work so they all ship before the end of the cycle.

Now that we have a standard way to think about capacity, we can talk about how we decide what to schedule.

The betting table

The betting table is a meeting held during cool-down where stakeholders decide what to do in the next cycle. The potential bets to consider are either new pitches shaped during the last six weeks, or possibly one or two older pitches that someone specifically chose to revive. As we said last chapter, there's no "grooming" or backlog to organize. Just a few good options to consider.

Our betting table at Basecamp consists of the CEO (who in our case is the last word on product), CTO, a senior programmer, and a product strategist (myself).

C-level time is only available in small slices, so there's an atmosphere of "waste no time" and the call rarely goes longer than an hour or two. Everyone has had a chance to study the pitches on their own time beforehand. Ad-hoc one-on-one conversations in the weeks before usually establish some context too. Once the call starts, it's all about looking at the options that made it to the table and making decisions.

The output of the call is a cycle plan. Between everyone present, there's knowledge of who's available, what the business priorities are, and what kind of work we've been doing lately. All of this feeds into the decision-making process about what to do and who to schedule (more on this below).

The highest people in the company are there. There's no "step two" to validate the plan or get approval. And nobody else can jump in afterward to interfere or interrupt the scheduled work.

This buy-in from the very top is essential to making the cycles turn properly. The meeting is short, the options well-shaped, and the headcount low. When these criteria are met, the betting table becomes a place to exercise control over the direction of the product instead of a battle for resources or a plea for prioritization. With cycles long enough to make meaningful progress and shaped work that will realistically ship, the betting table gives the C-suite a "hands on the wheel" feeling they haven't had since the early days.

The meaning of a bet

We talk about "betting" instead of planning because it sets different expectations.

First, bets have a payout. We're not just filling a time box with tasks until it's full. We're not throwing two weeks toward a feature and hoping for incremental progress. We intentionally shape work into a six-week box so there's something meaningful finished at the end. The pitch defines a specific payout that makes the bet worth making.

Second, bets are commitments. If we bet six weeks, then we commit to giving the team the entire six weeks to work exclusively on that thing with no interruptions. We're not trying to optimize every hour of a programmer's time. We're looking at the bigger movement of progress on the whole product after the six weeks.

Third, a smart bet has a cap on the downside. If we bet six weeks on something, the most we can lose is six weeks. We don't allow ourselves to get into a situation where we're spending multiples of the original estimate for something that isn't worth that price.

Let's look at these last two points more closely.

Uninterrupted time

It's not really a bet if we say we're dedicating six weeks but then allow a team to get pulled away to work on something else.

When you make a bet, you honor it. We do not allow the team to be interrupted or pulled away to do other things. If people interrupt the team with requests, that breaks our commitment. We'd no longer be giving the team a whole six weeks to do work that was shaped for six weeks of time.

When people ask for "just a few hours" or "just one day," don't be fooled. Momentum and progress are second-order things, like growth or acceleration. You can't describe them with one point. You need an uninterrupted curve of points. When you pull someone away for one day to fix a bug or help a different team, you don't just lose a day. You lose the momentum they built up and the time it will take to gain it back. Losing the wrong hour can kill a day. Losing a day can kill a week.

What if something comes up during that six weeks? We still don't interrupt the team and break the commitment. The maximum time we'd have to wait is six weeks before being able to act on the new problem or idea. If the cycle passes and that thing is still the most important thing to do, we can bet on it for that cycle. This is why it's so important to only bet one cycle ahead. This keeps our options open to respond to these new issues. And of course, if it's a real crisis, we can always hit the brakes. But true crises are very rare.

The circuit breaker

We combine this uninterrupted time with a tough but extremely powerful policy. Teams have to ship the work within the amount of time that we bet. If they don't finish, by default the project doesn't get an extension. We intentionally create a risk that the project—as pitched—won't happen. This sounds severe but it's extremely helpful for everyone involved.

First, it eliminates the risk of runaway projects. We defined our appetite at the start when the project was shaped and pitched. If the project was only worth six weeks, it would be foolish to spend two, three or ten times that. Very few projects are of the "at all costs" type and absolutely must happen now. We think of this like a circuit breaker that ensures one project doesn't overload the system. One project that's taking too long will never freeze us or get in the way of new projects that could be more important.

Second, if a project doesn't finish in the six weeks, it means we did something wrong in the shaping. Instead of investing more time in a bad approach, the circuit breaker pushes us to reframe the problem. We can use the shaping track on the next six weeks to come up with a new or better solution that avoids whatever rabbit hole we fell into on the first try. Then we'll review the new pitch at the betting table to see if it really changes our odds of success before dedicating another six weeks to it.

Finally, the circuit breaker motivates teams to take more ownership over their projects. As we'll see in the next chapter, teams are given full responsibility for executing projects. That includes making trade-offs about implementation details and choosing where to cut scope. You can't ship without making hard decisions about where to stop, what to compromise, and what to leave out. A hard deadline and the chance of not shipping motivates the team to regularly question how their design and implementation decisions are affecting the scope.

What about bugs?

If the teams aren't interrupted in the six week cycle, how do we handle bugs that come up?

First we should step back and question our assumptions about bugs.

There is nothing special about bugs that makes them automatically more important than everything else. The mere fact that something is a bug does not give us an excuse to interrupt ourselves or other people. All software has bugs. The question is: how severe are they? If we're in a real crisis—data is being lost, the app is grinding to a halt, or a huge swath of customers are seeing the wrong thing—then we'll drop everything to fix it. But *crises are rare*. The vast majority of bugs can wait six weeks or longer, and many don't even need to be fixed. If we tried to eliminate every bug, we'd never be done. You can't ship anything new if you have to fix the whole world first.

That said, nobody likes bugs. We still want ways to deal with them. Three strategies have worked for us.

1. **Use cool-down**. Ask any programmer if there are things they wish they could go back and fix and they'll have a list to show you. The cool-down period between cycles gives them time to do exactly that. Six weeks is not long to wait for the majority of bugs, and two weeks every six weeks actually adds up to a lot of time for fixing them.

2. **Bring it to the betting table**. If a bug is too big to fix during cool-down, it can compete for resources at the betting table. Suppose a back-end process is slowing the app down and a programmer wants to change it from a synchronous step to an asynchronous job. The programmer can make the case for fixing it and shape the solution in a pitch. Then instead of interrupting other work, the people at the betting table can make a deliberate decision. Time should always be used strategically. There's a huge difference between delaying other work to fix a bug versus deciding up front that the bug is worth the time to fix.

3. **Schedule a bug smash**. Once a year—usually around the holidays—we'll dedicate a whole cycle to fixing bugs. We call it a "bug smash." The holidays are a good time for this because it's hard to get a normal project done when people are traveling or taking time off. The team can self-organize to pick off the most important bugs and solve long-standing issues in the front-end or back-end.

Keep the slate clean

The key to managing capacity is giving ourselves a clean slate with every cycle. That means only betting one cycle at a time and never carrying scraps of old work over without first shaping and considering them as a new potential bet.

It is crucial to maximize our options in the future. We don't know what will happen in the next six weeks. We don't know what brilliant idea will emerge or what urgent request might appear.

Even if we have some kind of road map in our heads at the time scale above cycles, we keep it in our heads and in our side-channel discussions. Each six weeks we learn what's working and what isn't, what's important and what's not. There's no downside to keeping the option open and massive upside from being available to act on the unexpected.

What about projects that just can't be done in one cycle? In that case we still only bet six weeks at a time. Suppose we envision a feature that takes two cycles to ship. We reduce our risk dramatically by shaping a specific six week target, with something fully built and working at the end of that six weeks. If that goes as expected, we'll feel good about betting the next six weeks the way we envisioned in our heads. But if it doesn't, we could define a very different project. Or we could put the multi-cycle thing on pause and do something urgent that came up. The important thing is that we always shape what the end looks like for that cycle and that we keep our options open to change course.

Place Your Bets

Look where you are

Depending on whether we're improving an existing product or building a new product, we're going to set different expectations about what happens during the six-week cycle.

This invites us to reflect on where we are in the arc of our product's development and bet accordingly.

Existing products

When we add features to an existing product, we follow the standard Shape Up process: shape the work, bet on it, and give it to a team to build. We expect the team to finish and ship some version of the shaped work by the end of the cycle.

On an existing product, all of the existing code and design that *isn't* going to change defines a kind of empty space that the new feature will fit into. Shaping and building is like crafting a piece of furniture for a house that is already built.

New products

New products are different. Whereas adding to an existing product is like buying a couch for a room with fixed dimensions, new product development is like figuring out where the walls and the foundation should go so the building will stand.

We've noticed three phases of work when we build a new product from scratch. In each phase, the way that we shape and our expectations for how the team will work together during the cycle are different. These phases unfold over the course of multiple cycles, but we still only bet one cycle at a time.

R&D mode

At the very earliest stages of a new product, our idea is just a theory or a glimmer. We don't know if the bundle of features we imagine will hold together in reality, and the technical decisions about how to model them in code are even less clear.

This means there is a lot of scrapwork. We might decide half-way to standing up a feature that it's not what we want and try another approach instead.

In other words, we can't reliably shape what we want in advance and say: "This is what we want. We expect to ship it after six weeks." We have to learn what we want by building it.

We call this stage R&D mode and adjust for it in three ways.

1. Instead of betting on a well-shaped pitch, we mainly bet the *time* on spiking some key pieces of the new product idea. The shaping is much fuzzier because we expect to learn by building.

2. Rather than delegating to a separate build team, our senior people make up the team. David (CTO) takes the programming role and works with Jason (CEO and designer) or a senior designer with Jason's guidance. This is necessary for two reasons. First, you can't delegate to other people when you don't know what you want yourself. Second, the architectural decisions will determine what's possible in the product's future — they define the "holes" that future features fit into. At this phase the team needs to hold the vision of the product and be able to judge the long-term effects of design decisions.

3. Lastly, we don't expect to ship anything at the end of an R&D cycle. The aim is to spike, not to ship. In the best case we'll have some UI and code committed to serve as the foundation for subsequent work. The goal is to learn what works so we can commit to some load-bearing structure: the main code and UI decisions that will define the form of the product going forward.

We can't ship anything to customers with just a single cycle of R&D work. But we still don't commit to more than one cycle at a time. We may learn from the first cycle that we aren't ready to tackle the product yet. Or we may discover that our intuition rang true and the product is coming together. Depending on how it goes, we'll decide cycle-by-cycle whether to continue spending informal time in R&D mode.

Production mode

If we continue to get warmer after some R&D cycles, we'll eventually reach a point where the most important architectural decisions are settled. The product does those few essential things that define it, and the foundation is laid for the dozens of other things we'll have to do before we can ship to customers.

With this structure in place, the senior team can bring in other people to contribute. This is the flip to production mode, where we work in formal cycles with clear-cut shaping, betting, and building phases. Production mode is like working on an existing product: the precedent set by the R&D work enables new contributors to identify where new functionality belongs and how it fits into the whole.

In production mode:

1. Shaping is deliberate again. The shaped work describes

what we expect to see at the end of the cycle.

2. The team that builds the projects is no longer limited to the senior group. It becomes possible to bet multiple teams in parallel (if you have them) and cover more ground.

3. Shipping is the goal, not spiking. But because the product isn't publicly available to customers yet, we define 'ship-ping' differently. Shipping means merging into the main codebase and expecting not to touch it again.

Since we aren't shipping to customers at the end of each cycle, we maintain the option to remove features from the final cut before launch. This means we can still be experimental. We can bet six weeks on a feature without knowing if we'll want it in the final product. That's not a problem as long as we set expectations to the build team: we can't predict what we'll want in the final cut, and we're willing to risk this cycle to take our best swing at the idea.

Cleanup mode

In the final phase before launching the new product, we throw all structure out the window. We call this cleanup mode. It's a free-for-all. We've built enough new products to learn that there are always things we forget, things we miss, details that aren't right, and bugs that creep in over the course of the R&D and production mode cycles.

There's something about putting your finger near the launch button that makes your hair stand up. Everything suddenly gets "real." Things we dismissed before pop out at us with new importance.

That's why we reserve some capacity at the end for the unexpected. In cleanup mode:

1. There's no shaping. The cycle is closer in spirit to the "bug smash" mentioned in the previous chapter. Leadership stands at the helm throughout the cycle, calling attention to what's important and cutting away distractions.

2. There aren't clear team boundaries. Everyone jumps in to help however they can.

3. Work is "shipped" (merged to the main codebase) continuously in as small bites as possible.

Discipline is still important. We have to check ourselves to make sure these are must-haves we're working on, not just our cold feet begging us to delay launch. Cleanup shouldn't last longer than two cycles.

Cleanup is also the phase where leadership makes those "final cut" decisions. A smaller surface area on a V1 means fewer questions to answer, less to support, and less we're committing to maintain indefinitely. Sometimes we need to see all the features working as a whole to judge what we can live without and what might require deeper consideration before shipping it to customers.

Examples

The Dot Grid Calendar

We built the Dot Grid Calendar (see Chapter 2) for Basecamp, an existing product. We shaped the project, bet six weeks on it, a team built it, and then we shipped it straight to customers.

A new product: HEY

In 2020, after two years of development, we launched a new email app and service called HEY. HEY was in R&D mode for the first year of its development. A team of three, Jason (CEO), David (CTO), and Jonas (senior designer) explored a wide variety of ideas before settling on the core. Nearly a year of production mode cycles followed, where all of Basecamp's teams fleshed out HEY's feature set. We ended with two cycles of cleanup and significantly cut back the feature set to launch in July 2020.

To be precise, there was some overlap between R&D and production mode after that first year. Basecamp was big enough as a company that the senior team could shape and delegate production-mode projects around parts of the app that were settled while continuing to explore new territory in R&D mode themselves.

Every bet on HEY was placed one at a time. The betting table didn't know they would be working on HEY for two years during those first few R&D cycles. Gradually they gained confidence in the idea and grew a big-picture appetite for how many cycles they were willing to spend on HEY. But they made no specific commitments about what would go into those cycles. And flipping attention back to Basecamp, our existing product, was always on the table.

An experimental feature: Hill Charts

A third example shows some grey area. When we built the Hill Charts feature in Basecamp (see Chapter 13), we had no idea if it was going to work out or not. Basecamp was an existing product, and it felt too risky to bet on releasing this experimental feature to customers. So we framed the project more like a production mode bet on a new product. We shaped a first version that was just functional enough to use ourselves. We didn't expect to ship it to customers without doing an additional cycle on it. This was a risk: we bet one cycle, not two. If it didn't work out, we'd scrap it. If something more important came up, we might not ever schedule the second cycle. But we ended up feeling confident after the first cycle. We shaped a project to round it out, decided to bet another cycle, and then shipped it to customers.

Questions to ask

Here are some common questions you might hear when people at the betting table are debating which bets to place.

Does the problem matter?

Just like in pitch write-ups, we always take care to separate problem and solution. The solution doesn't matter if the problem isn't worth solving.

Of course, any problem that affects customers matters. But we have to make choices because there will always be more problems than time to solve them. So we weigh problems against each other. Is *this* problem more important than *that* problem right now?

How the people at the table judge problems depends on their perspective, role, and knowledge. For example, a problem might impact a small segment of customers but put a disproportionate burden on support. Depending on your exposure to support and which aspect of the business you're focused on, you may weigh that differently.

Sometimes a solution that is too complicated or too sweeping may invite questions about the problem. Do we really need to make so many changes across the app? Have we understood the problem specifically enough? Maybe there's a way to narrow it down so that we get 80% of the benefit from 20% of the change.

Is the appetite right?

It's good when we have a solution shaped to a reasonable time frame, like two or six weeks. But we might still debate whether it's worth the time. Suppose a stakeholder says they aren't interested in spending six weeks on a given pitch. The negotiation could go a couple directions from there:

1. Maybe the problem wasn't articulated well enough, and there's knowledge that the shaper can add to the conversation right now to swing opinion. For example, "Yeah it doesn't happen often, but when it does people are so vocal about it that it really tarnishes perception of us." Or "Maybe it sounds trivial, but support has to go through 11 time-consuming steps to get to resolution."

2. Sometimes saying "no" to the time commitment is really saying no to something else. Maybe there's something about the solution or the technical implementation they don't like. Asking "How would you feel if we could do it in two weeks?" can uncover that it's not so much about the time. The CTO might answer, "I don't want to introduce another dependency into that area of the app."

3. The shaper might just let the idea go if interest is too low.

4. The shaper might go back to the drawing table and either work on a smaller version (for a shorter appetite) or do more research if they believe the problem is compelling but they weren't armed well enough to present it.

Is the solution attractive?

The problem may be important and the appetite fair, but there can be differences about the solution.

For example, adding interface elements to the screen carries an invisible cost: giving up the real estate. A button in the corner of the home page might perfectly solve the problem. But that real estate is valuable. If we give it up now, we won't be able to use it in the future.

Are we selling it too cheaply to solve this particular problem?

If someone offers an immediate design solution, like “how about we move that button to an action menu instead,” we might discuss it. But generally we’ll avoid doing design work or discussing technical solutions for longer than a few moments at the betting table. If we catch ourselves spending too much time in the weeds we’ll remind ourselves “okay, we’re not doing design here” and move back up to the high level.

Is this the right time?

The kind of project we want to do next can depend on which projects we’ve done recently. Maybe it’s been too long since we’ve made a splash of news with a new feature. Or perhaps we’ve been building too many new features and feel overdue to fix some long-standing customer requests. Or if the teams spent the last couple cycles in the same area of the app, their morale may dip if we plan yet another project doing the same kind of work.

Those are all reasons that we might pass on a project even though it’s perfectly well shaped and valuable. The project’s great; it’s just not the right time.

Are the right people available?

As part of the betting process we choose who specifically will play which role on each team. That is, we’ll pair a project with a specific small team of a designer and one or two programmers. We have a “Core Product” team of designers and programmers and we select from that pool when planning teams for each cycle. The team will work with each other for the whole cycle and then the next cycle can be a different combination of people.

Different projects require different expertise. Maybe we need some more front-end programming on this one. Or this other one is going to invite a lot of scope creep so we need someone who's good with the scope hammer.

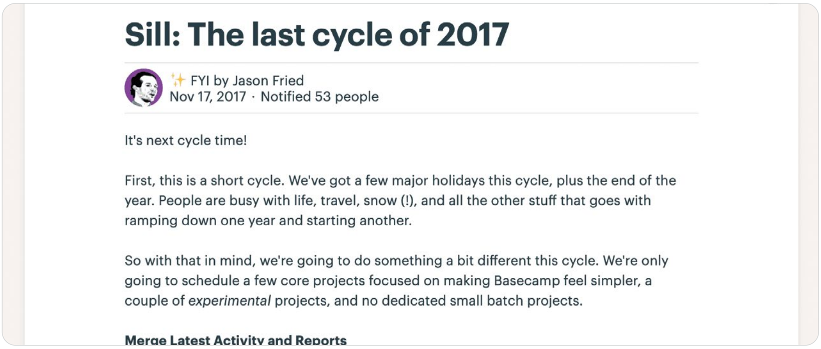
The type of work each person has been doing is another factor. Someone who's done a long string of small batch projects might prefer to take on a big batch, or vice versa.

And lastly there's always a little Calendar Tetris with peoples' availability. Vacations or sabbaticals affect which projects we can schedule in the coming cycle.

We've seen some other companies use a different model where instead of assigning the projects to people, they let the team members choose which projects they want to work on. Culturally, we are too meeting-averse for this extra step. But we've heard it can work well for some teams because the project teams have a little more buy-in.

Post the kick-off message

After the bets are made, someone from the betting table will write a message that tells everyone which projects we're betting on for the next cycle and who will be working on them.



Sill: The last cycle of 2017

✨ FYI by Jason Fried
Nov 17, 2017 · Notified 53 people

It's next cycle time!

First, this is a short cycle. We've got a few major holidays this cycle, plus the end of the year. People are busy with life, travel, snow (!), and all the other stuff that goes with ramping down one year and starting another.

So with that in mind, we're going to do something a bit different this cycle. We're only going to schedule a few core projects focused on making Basecamp feel simpler, a couple of *experimental* projects, and no dedicated small batch projects.

Merge Latest Activity and Reports

Jason announces the bets for the next cycle with a Basecamp message