Starting Successful Software Projects

A short guide to picking a direction, defining requirements and getting your development team productive

Najaf Ali, Founder at [Happy Bear Software]

Table of contents

Introduction

Most software projects fail. Following this guide will help you avoid the most common pitfalls. In this guide we're going to talk about:

- **Starting your project**. Establishing context for the project • Deciding what you're building • Defining your project's win conditions • Capturing this information in a *project charter*.
- **Assembling your team**. What skills to hire for • Whether to go freelance, with an agency, or hire a permanent developer • How to select a developer • Making your expectations clear.
- **Defining requirements**. Using wireframes to mock up your application • Defining user roles • Writing task definitions • Breaking tasks down into implementable pieces
- **Shipping early and often**. The first thing you should do when your project starts • Why to ship each feature as it's developerd • How to help your team ship early and often.
- **Iterating towards your win condition**. Defining iteration length • Keeping your backlog stocked with tasks • Prioritising tasks in terms of your win condition • Testing newly delivered functionality • How to give feedback on features/bugs • Iterating on your development process.

At Happy Bear Software, our expertise is on *execution*. Other firms will sell you the process, we sell you the product. This is the absolute minimum process required to get us (or any development team) on-board and productive in the shortest time.

Project start

You want to build a web application that solves a business problem. You've got some ideas in your head about how it could work. You want to translate these ideas into working software. The first step to doing this is to be explicit about your situation. You need to be clear about why you're building this software, what you want to build, and what success looks like for this project.

Once you've answered those questions, you'll have the information you need to make a plan to actually implement the project. You'll be able to start thinking about the expertise you need, the scope of your project and the sort of budget you should be considering.

Why are you here?

- Who exactly is going to use your software?
- What problem are you solving for them?
- How have they been solving that problem until now?
- What is bad about they way they're solving it now?
- Who are you and why are you building this software?
- All software projects are investments, so what are the investors in thisproject hoping to get out of it?
- Who are the key people motivated to get this project built?
- Why do they want it built?

What are you doing?

- How are you going to solve your users problem?
- What will the software allow your users to do?
- How will your solution be better than what users are already doing?
- How much can you invest in building the solution?
- In a few sentences, can you describe how a user will interact with your proposed solution?

What does success look like?

- If your project has an external time constraint, when does it have to go live for it to be a success?
- Are there any key metrics that can be used to measure it's success?
- What features must be shipped for it to successfully to solve the targeted problems?
- What goals need to be achieved for the project to be a success for investors?
- How will you confirm that those features in fact solve the targeted problems?
- What doesn't have to be completed for the project to be considered a success?

Writing a project charter

These questions are there to get you thinking in explicit terms about why you're working on this project, what exactly you're building and how you'll know when you've succeeded. Working through these questions will help you gain clarity and direction.
There are a lot of benefits to getting your answers to these questions on paper. A project charter captures this information.

Going through the motions of writing a project charter has a number of benefits:

- With a central, high-level document specifying the project's purpose, the project becomes easier to share, discuss and refine.
- New team members can gain an understanding of the high-level context and goals of the project at a glance.
- When there are disputes about the direction of the project, the team or other stakeholders can consult the charter to help make decisions.

The project should include, at the minimum:

- A code name. Software projects typically go through many name changes before a final name is settled on. A code name helps you easily.
- Answer the question of why you're building this software.
- Describe what you're building.
- Determine what success looks like.
- Provide any additional information, including a project timeline, details of key stakeholders, links to further reading and anything else that someone new to the project would find useful.

Here's an example of a project charter.

The charter should be short enough that your team members will actually read it. It needs to be kept up to date so that it's still useful for the team and anyone that finds themselves needing information on your project. If it gets too long or otherwise stops being useful, it should be edited down or deleted outright.

Once you've got this information in one place, the next step is to start to plan how you're going to get closer to the project's success criteria. That means thinking in detail about the desired software you want to build, about the team you're going to need to assemble to build it and the steps you're going to take to minimise project risk.

Assemble your software team

You know what web application you're going to build, why you're going to build it and what success looks like for this project. Now you have to find someone to build it.

Start with a full-stack developer

Even if you have a large budget, hiring a large team at once may not be the best use of it. Starting with a smaller team and iterating to a larger one presents fewer challenges than starting with a large team from day one. It allows you to iterate quickly without paying large amounts of communication overhead that come with a bigger team.

A full-stack developer is one that works with both front-end and back-end web technologies. They can turn designs, wireframes, specifications, and anything else you use to define your application and turn them into working software that solves your business problems.

A full-stack developer should be the first person on your team. You might hire specialists to handle particular areas of expertise later, but no one else can actually ship the software you want to launch from beginning to end.

Engaging with a full-stack developer earlier in the process allows you start iterating on your software more quickly. This allows you to start giving feedback within days of project start rather than weeks or months, increasing the likelihood that the software that's delivered is the software you actually want.

Choose an appropriate legal structure

There are a number of ways to engage with a full-stack developer. Here are a few:

- Permanent employment - If you are sure that you will have years of work for a full-time developer then hiring a developer as a permanent employee may be a cost-effective option. Permanent employees take a lot of time and effort to find, qualify and hire. They're also difficult to fire. The main benefit in hiring permanent employees is that they provide a guaranteed ongoing development service and can gain a deep knowledge and understanding of your business.
- Freelance - Freelancers are a good option if you have experience managing development projects and you need "at-will" development expertise on a shorter term basis. It's expected that you'll hire/fire freelancers relatively often and so can experiment until you find ones that work well with you. They're typically more expensive than permanent employees, but can typically still be quite cost-effective.

- Agency - A development consultancy is a good option if you expect help in planning, defining requirements and managing your development process in addition to implementation. Agencies can provide continuity where freelancers can't. They also have ways to address your concerns and e.g. move different developers onto your project if your current one isn't working well with your team. For the reduced project risk that they provide, they typically cost more than freelancers.

Select a developer for maximum probability of success

However you structure the engagement, there are a few obvious steps you can take to maximise the chance of your project succeeding.

Here's what to look for:

- A recommendation from someone you trust. People with your interests at heart won't recommend incompetent professionals to you. If a developer has done a good job for someone you trust, then their recommendation gives you some guarantee that they might do a good job for you too.
- A track record of shipping software similar to yours. If you're building a web application that has a complex integration with a payment system, there will be less inherent risk if you hire a developer that has built a complex integration with a payment system before.
- Expertise in tools typically used to solve your problem. If you operate in the data science space, it's likely that you'll be using Python or R. If you're building a web application, it's likely that you'll be using PHP or Ruby on Rails. If you're building a JavaScript heavy application, it's likely that you will need to assess frameworks such as Angular, React and Ember. Choosing a developer that has expertise in technologies that you're likely to need will increase the project's chances of success.

Make your expectations clear

As part of your assessment process, you need to make clear exactly what it is you expect of the developer or agency you hire. In particular:

- How will we structure planning and requirements gathering?
- How will we determine whether a feature is "complete" or not?
- How exactly will the developer communicate the status of the project to you?

For example, you might specify that:

- We will hold weekly meetings on Monday afternoon at 13:00 to plan new features for the upcoming week.
- Features will be deemed "complete" when the acceptance criteria defined for each feature is met.

- We will provide all information required to start features before we expect developers to start them (wireframes, acceptance criteria, design, seed data, etc).
- Developers will make short (e.g. one-line) daily status updates about what they've been working on, what they're going to work on next and if they've face any obstacles that have hindered progress.
- Developers will also provide a weekly summary email that lists what was worked on and exactly what features were shipped to each environment.

The exact expectations don't matter as much as the fact that you specified any at all. By spelling out in detail exactly how you want developers to interact with you, you give them a personal set of success criteria. These success criteria should be in line with moving you incrementally closer to your project goal.

Not all developers will like your proposed way of working, so invite feedback from potentials and see if you can come to an agreeable workflow. Developers that are a bad fit for you will opt-out, and developers that resonate with your working style will be more likely to find you.

Summary

- The start of your software team is a full-stack developer.
- You can hire one as a permanent employee, as a freelancer or through an agency.
- A developer with good references, a track record of work on project like yours and expertise with tools that are used to solve your problem will result in a higher chance of success.
- Making your expectations about project structure and communication will result in developers more amenable to your style of working.

Once you've created a short-list of potential developers, it's time to begin drilling down into the details of the exact software you want built. You'll need an experienced developer at your side to do this effectively.

Defining Requirements

You've defined the project goals in broad strokes and are talking to someone that can build your web application. The next step is to plan the first stages of the project so that you can start shipping software.

The goal of defining requirements is to specify the desired behaviour of the software you want made in such a way that the developer working on your project can build and ship it. Everything else should fall by the wayside. Consultants selling the benefits of the "Agile" project management technique of the week will try to convince you otherwise. Ignore them and stay focused on shipping software.

Make continuous forward movement on the project. You don't need to plan the entire project up-front. You need just enough planning so that the developer can stay productive for the coming few weeks. You want to develop a shared understanding of the first few sets of requirements, and then continually plan a week or two ahead of current development.

Tools for requirements definition

One of the reasons that projects fall behind schedule is a mismatch in the understanding of requirements between the client and the developer. You're using the same words but your mental model of the exact requirements for delivery end up being different. This leads to a lot of back and forth that could have been avoided with more precise planning.

There are a number of tools at your disposal for defining tasks with more precision.

1. Wireframes

One of the first tools to reach for when defining requirements is wireframing. This is where you sketch low-definition outlines of the user interface you envision. You don't need to build high-definition sketches of what the UI looks like when building the basic functionality.
To create wireframes you can use pen and paper and then take pictures of the screens you sketch with your phone. At Happy Bear Software we use Balsamiq mockups, but there are plenty of other perfectly good wireframing tools available.

2. User roles

Your web application is likely going to have provide for users filling a number of different roles. For our purposes, a 'role' is simply a type of user for whom the functionality of the application is different from users of other roles.

Any web application available over the internet has the “anonymous user” role. This is a random user on the internet who hasn’t logged in, usually heavily restricted in the ways they can use your application. Web applications typically also have a “registered user” role that has access to the primary functionality of the application.

Roles can be defined in terms of the level of access a user has. “Admin” users might have access to more functionality or data than regular users. There may be multiple, hierarchical levels of access. The access roles might scoped to an “organisation” or “account” in the system. Users might also be segmented into roles based on their “plan”. SaaS web applications in particular offer tiered plans. Higher-cost plans usually have more features and higher limits on some form of resource consumption.

You can also segment users by the task they’re trying to achieve. A document editing application might have some users that are in charge of editing and others that need functionality for “signing off” documents for compliance purposes. The application might want to provide slightly different interface to each of these types of users.

Roles might also be defined by engagement metrics like how often the user uses the application, number of records they’ve created etc. The interface you provide to brand new users that have yet to learn the system might be different from the interface you show to experienced users.

Finally you could also segment users based on their profession, which would be data you collected at registration time. An appointment reminder application might present a different set of functionality to hair salons than it does to psychotherapists for example.

To summarise this list, users can be segmented by:

- Whether they are anonymous or registered.
- Access control and permissions
- SaaS plan tier
- The task they’re trying to complete
- Their level of engagement with your application
- Their profession and desired end-goals with the application
- Any number of other axes along which your users can be split

At this stage of requirements gathering, it’s worth defining your features in terms of the user roles that are important to you. This could be as simple as just “anonymous” and “registered” to start with, but choose whatever is relevant to your project aims.

3. Task definitions

For our purposes, task definitions are a way of describing a single desired change to how your software works. We use a task management tool called Trello to keep track of tasks, but there's no reason you have to do this. A google document, pen/paper, or whatever allows you to capture the requirements of a task in sufficient detail is fine.

Your task definitions should have three items of information on them:

Name

A very short name for the task. The purpose of this is so that when allupcoming tasks are arrayed in front of you, you'll be able to quickly recall what this task refers to.

The name is what you write when you're quickly jotting out a list of tasks that you will discuss in more detail later. There isn't enough information about the task for a developer to start work on it. All you've done so far is capture that the task exists so that you know you need to think about it later.

Description

A description is there to provide a broad overview of what the task entails and why you're doing it. Some recomend that the description takes the following format:

As a [user role], so that I can [achieve some goal], I want [some functionality that let's me achieve that goal].

This helps to quickly define which user roles this feature is applicable to, why they need the feature and what in particular they want.

There's no reason to stick to this formula religiously. If there's a task for upgrading the Ruby version used by the application to the latest stable release, the description can just be: Upgrade the application's Ruby version to the latest stable release.

Once you've written a description, there still isn't enough information to start work on the task. The description serves as a starting point for the conversation you and a developer will need to have before work on the task can begin.

Acceptance criteria

Acceptance criteria are a list of criteria that form your shared understanding of what it means for a task to be "complete". A developer will likely use the acceptance criteria as a todo-list for building the feature in question. They'll then check against the acceptance criteria to confirm

that they've completed. The acceptance criteria can then act as a guide for you to give them feedback.

The acceptance criteria are the result of a conversation that you and your developer have about the task at hand. This conversation can start with you writing a list of acceptance criteria and the developer giving you feedback on them. You can ask the developer to write the acceptance criteria for a task and give them feedback instead if you prefer. Developers can help you to discover edge-cases and implications you haven't thought of when writing acceptance critiera, so it's highly inadvisable to write them on your own.

Here are some things your acceptance criteria might define:

- How the task will change the behaviour of the software for each defined user role.
- Whether the desired feature should match a given wireframe.
- What cases the resulting functionality doesn't need to address at this stage.

Bringing task definitions together

Here's an example of a task definition with all of the above-mentioned parts:

Comments on blog posts

As an anonymous user, so that I can express my opinion on a blog post, I want the ability to leave a comment.

Acceptance criteria:

- *Anonymous users should be able to add comments at the bottom of each post (see attached wireframe).*
- *Comments should display in reverse chronological order under each post*
- *Comments cannot be empty.*
- *Comments cannot contain more than 1000 words.*
- *Registered users should be able to disable comments on their blog.*
- *Registerd users should be able to disable comments on a particular post.*

The end result of your requirements gathering should be a set of task definitions in roughly the format described above.

However you decide to define your requirements, here are the most important things you need to do:

- Note down the various features and tasks that you intend to have developed.
- Discuss them with a developer, preferably the one who's going to develop these features.

- Capture the details of the specification that you agree on with the developer as acceptance criteria.

Task Decomposition

Remember that your goal is swift forward movement on the project. That means getting as much of your software shipped as quickly as possible. You want your developers to deliver the functionality as they work on it, rather than in one big release at the end of each week or month. To do this, we need to take the requirements we documented as task definition and break them down into the smallest possible pieces of functionality that can be delivered individually.

Let's consider our example of adding comments to blog posts. We can start by grouping the related acceptance criteria together, and then separate the tasks out into their own definitions.

---

1 • Comments on posts

As an anonymous user, so that I can express my opinion on a blog post, I want to be able to leave a comment.

Acceptance criteria:

- *Anonymous users should be able to add comments at the bottom of each post (see attached wireframe).*
- *Comments should display in reverse chronological order under each post*

---

2 • Comment validation

As an anonymous user, so that my comments on posts are well formed, I want any comments I enter to be validated

Acceptance criteria:

- *Comments cannot be empty.*
- *Comments cannot contain more than 1000 words.*

---

3 • Comment configuration

As a registered user user, so that I can decide whether or not to allow comments on my blog, I want to be able to turn comments on/off across my blog or on a per-article basis.

Acceptance criteria:

- *Users should be able to disable comments on their blog.*
- *Users should be able to disable comments on a particular post.*

These were all part of one task to begin with and are now split into three separate units of work.

Smaller items of work are easier to schedule, easier to reason about and can be delivered independently of each other.

Summary

Once you've got a broad understanding of the goals of your project and a developer in line to start building it, it's time to specify the first few features of the project in detail. Tools for doing this include:

- Wireframes
- User roles
- Task definitions, including names, descriptions and acceptance criteria.

Once you have your initial requirements specified as task definitions, you can break those definitions down into smaller tasks to make the work easier to swiftly deliver.

At the end of your initial planning and requirements gathering activity, you should have a list of tasks that both you and your developer agree are ready to start. With clear requirements set, your developer can now focus on delivering the best implementation possible.

Ship early, ship often

You have a developer and a clear set of requirements to get your project started. Now you need to begin shipping software.

By “shipping” we mean going through the process of implementing your requirements and deploying it to somewhere you can interact with it.

The following do not count as shipping:

- Having the code for it complete but not deployed yet
- Having it work on your developer’s computer
- The software being mostly done but with a few loose ends to tie up

For a web application, shipped typically means that there’s a URL that you visit with your browser that will serve your application and allow you to interact with it.

The first thing you should do is ship

Creating a deployment of your software should be the very first thing your developer works on. They can’t deliver any software to you before this step. This should result in an application hosted at a domain name or IP address that doesn’t do very much to begin with.

Setting up a deployment environment for a web application is easy when it doesn’t do anything at all. It becomes more complex with the more features you add to your application. Rather than figure all of the requirements out right at the end of the project, typically a few days before a big launch date, shipping first allows you to solve any deployment problems as you go. You pay this cost up front when you have ample time to spend on them rather than later when you’re feeling greater schedule pressure.

Ship each feature as it’s developed

You should ship each and every feature as it’s developed, rather than in one big periodic release.

Shipping means fewer errors

Each change you ask to make to the system requires special consideration when deploying. Does deploying this change require special actions (like running database schema migrations or making configuration changes)? Will it react to production data differently than to development data? Are there any other problems it will have in production that it doesn’t have in development?

Skilled developers will try to foresee these problems to the extent possible but experienced developers know that they'll eventually miss something. While they will try to prevent problems, they will also prepare for things to go wrong.

If you ship a large feature all in one go, then you have to deal with all the emergent production only problems in one go. This can result in complex, difficult to diagnose bugs that hamper development speed and otherwise hold your project back.

By instead breaking each task down into the smallest tasks that make sense to deliver and deploy them as they're developed, you can deal with one set of problems at a time. The parts of the task that deploy without issue will work just fine. The parts of the task that surface problems on deployment can then be easily identified. This makes your development process less error-prone. It also means that when thereare errors in production, they're easily diagnosed and fixed.

Shipping gets you closer to the software you want, faster

One major source of risks in software projects is the gap between the software that you want built and the software that your developer believes you want build. You can take a number of steps (like providing designs/wireframes and clear acceptance criteria) to try and close this gap. However clients often get a clearer understanding of their requirements once they see a first version of their deployed software.

Rather than trying to endlessly perfect the requirements gathering process, we can try to create a tight feedback loop. That means that instead of you giving feedback on the delivered software after a big release, you give it as soon as it's completed and deployed. If the acceptance criteria has been met then you can immediately create a new task definition based on your feedback and schedule it into the next iteration.

By deploying and giving feedback as each feature is developed, you help to make the desired and delivered software converge in smaller steps. You can make finer-grained course corrections on the development process than if you had to wait for a big release to interact with the software. This is especially important on projects where the major requirements change (which is, in our experience, all projects).

Shipping shows progress

The basic content of client communication on software development projects at the implementation is status updates on progress. The more simply this is communicated the better. Communicating the status of progress on large, complex tasks is strictly more difficult than that of small, simple ones.

From a non-technical client's point of view, a developer saying that they're "still working" on a large task is a black box. They have no visibility into the details of the task, so have no way of

knowing the real status of progress. Even if given a step-by-step break down of the current progress, they have no way of verifying and giving feedback on the completed steps. The y end up having to rely on the trust they have for the developer, which can be eroded if the developer incorrectly estimates the remaining time required to complete the task.

From a developer's point of view, giving the client feedback on progress of a large, complex task presents a number of challenges. There may be implementation details that are difficult to explain or seemingly irrelevant to the client that are nonetheless holding the task up. Giving constant status updates may in fact be preventing the developer from concentrating on the task at hand in sufficient depth.

When a lender seeks payment from a borrower on a loan, the best possible status update they can receive is payment. The best possible status update that a client can receive in a software project is shipped software.

By breaking tasks down into the smallest deliverable pieces, the client no longer needs to rely on trust and the developer no longer needs to give awkward status updates. The developer can deliver the software, tell the client how to test it and that is the status update. That's what shipping early and often buys you.

Help your team ship often

You want your team to ship often. Here's what you need to do to make that happen:

- Ensure that the developers that will actually implement the software are consulted when writing your acceptance criteria on task definitions. If your developers aren't given clear requirements, they can't ship anything.
- Break tasks down into the smallest conceivable pieces that can be shipped individually. You will probably need your developer's help to do this.
- Give developers time to set up a deployment process that makes shipping software extremely easy to do. They should be able to do it in two or three steps, and it shouldn't take longer than ten minutes to complete, beginning to end.
- Tell your developers explicitly that they're to deploy the software as it's developed. A feature is not "done" until it's deployed. A feature should be deployed successfully and QA'd before they continue to the next feature in their to-do list.

Summary

- Shipping early and often makes your software project more successful.
- The first thing you should do on your project is set up a production environment that your developers can deploy software to.
- Shipping as features are developed results in a number of benefits:
    - It results in less error-prone and easier to fix software.

    - It allows you to give more fine-grained feedback on delivered features.
    - It gives you more visibility into the progress your developers are making.
- You can help your team ship more often by consulting them during requirements gathering, allowing them to set up a smooth deployment process, and making it clear that they're to deploy as they go.

Once you become used to seeing consistent (usually daily) forward movement on your project, you become much more confident in yours and your team's ability to deliver on the requirements you set out to meet. You insulate yourself from much of the risk inherent in software projects and so greatly increase your chance of success.

Iterate towards your win condition

You've defined requirements, found a developer and you're now shipping software. You need to keep doing this and focus on achieving your win condition. Working in iterations will help you do this.

Work in iterations

In iteration is a fixed period of work on your project that results in software that you can put in front of users. Iterations usually last one or two weeks. Each iteration should move your software closer to your win condition.

For an iteration to be successful, a number of things have to happen.

First, decide how long an iteration is going to be. The shorter the iteration, the more meeting overhead it requires. One week can be good for projects getting started as it allows for quicker changes in direction. Later on when your win conditions/requirements are understood in greater detail, you might find productivity gains in increasing the iteration length to two weeks.

Before any iterations begin, you need to define in as much detail as possible the tasks that you want to accomplish. Once you've consulted with developers and defined these tasks in sufficient detail to implement them they become your "backlog". This is a set of task definitions that is ready for you to prioritise. Your backlog will need to be refined, pruned and added to so that you have a ready list of tasks that your developers can work on.

You should prioritise iterations based on the "launchability" of the delivered software. Instead of working up to large complex features, assume for a moment that this is iteration will be the only work done on this project. Choose features that if successfully developed by the end of the iteration, you could deliver to your users. If you have large, complex features to deliver, start with a small, simple version that gives user 10% of the desired functionality. Prioritising like this means that you're thinking in terms of your product launch from day one.

Before each iteration, after prioritising your tasks decide with a developer what tasks they estimate they can reasonably deliver by the end of the iteration. The goal here is not pin-point accuracy, just a shared understanding of what you can reasonably expect to have delivered by the end of the iteration. The smaller the tasks, the easier it is for the developer to reason about the time it takes to complete them. There's no guarantee that what you agree upon will actually be shipped by the end of the iteration, but the act of agreeing will focus both you and the developer on a clear goal for the time box. Having a fixed set of tasks to work on for the iteration also allows your developer to focus more fully on the task at hand.

Soon you'll see your task definitions being delivered as working software. Once your developer confirms them as done, the best thing you can do is manually test the new functionality yourself as soon as possible. If the delivered result doesn't match the previously agreed acceptance criteria for the task, then push back and make the developer amend the work.

The delivered software may comply with a reasonable reading of the acceptance criteria but still not be what you intended for this iteration. In this case, create a new task definition and use the differences between what was delivered and what you intended as acceptance criteria. Schedule this task into the next iteration rather than the current one.

During testing you might find what appear to be bugs in the delivered software. In order to get these fixed as swiftly as possible, create a task definition for the bug that describes it in detail. The definition should provide details of the expected/observed behaviour, steps to reproduce and the specific acceptance criteria that the behaviour violates.

While the developer is working through the implementation, you need to be preparing your backlog for the next iteration. This means observing progress, meeting with other stakeholders, re-evaluating the priority of existing tasks and writing up new task definitions with clear acceptance criteria. At some point during the iteration, you will need to consult with your developer about the acceptance criteria on your task definitions, in particular about tasks that you intend to schedule in the next few iterations.

If the developer manages to complete all tasks in this iteration before the end, then you can pull in smaller tasks from the backlog that are ready to start. If one or two tasks were not completed by the end of the iteration, they can be continued in the next iteration.

Towards the end of the iteration, you'll want to start the process again. Decide with your developer a set of tasks that can reasonably be expected to be completed by the end of the next iteration and schedule them in.

At the end of each iteration, remember the win condition that you defined at the beginning of your project. Ask yourself if you're moving closer to it. What development tasks do you need to schedule in to get closer to that condition?

There will be problems with any conceivable development process and you will need to adapt it to the situation on the ground. You should meet regularly with your developer to ask if there's anything you can change about your development process to make it better. Your aim is to remove obstacles to getting work done.

Launch is the beginning

Software is never finished. Think of any successful web software company. They have software products with product-market fit. Why do they keep expensive engineering teams on staff? If

their products are successful, they could surely just keep a skeleton crew on staff for maintenance and stop all work on development.

This is because the win conditions on software projects change. If you meet the goal of doing a product launch, the next win condition might be to get it to one thousand users. After that your win condition might change to some metric of user engagement.

The scope of your software project will change. The team may expand or contract. The project may in fact be put into maintenance mode or instead become the main focus of your business. It's very difficult to predict how software projects change over time.

An iterative development process helps you respond to that change. It helps you get maximum productivity and communication from your development team. It means that you can re-prioritise tasks at the end of each iteration rather than at large project milestones. It allows you to demo a working version of your product almost as soon as you start development on it.

While you may be focused on your product launch now, it is just the beginning. Once you have a first version in front of users, the requests for features will pour in. Both leading up to the product launch and after it, you will need a process that helps you respond to change. An iterative development process will do that.

Kickstart your development project with Origin

A web application roadmapping, planning, and requirements gathering workshop

Does all of that sound like a lot of work? Do you have a high priority software project but aren't sure where to start?

At Happy Bear Software we run a planning workshop called *Origin* that walks you through these steps. Origin breaks down into three steps:

1. **Research.** We meet with you, conduct interviews, examine all existing documentation and do our own research in order to gather as much information about your project as possible.
2. **Definition.** We help you decide what you're going to build, why you're going to build it, and what your project's win conditions are. We present you with alternatives and decide on a path ahead to meet your success criteria given the project constraints. We write all of this down in a *project charter.*
3. **Plan.** We define the features your project requires at a high level. We then use wireframes and user stories to break down the most important features into tasks small enough to begin implementation. We produce a *product backlog* that documents all of these features and tasks.

At the end of *Origin* **you may decide that your project is not in fact feasible** given the budget/time constraints at hand. It's much cheaper to realise this after a short engagement with us than it is three months into a development project.

Once you've completed *Origin* **there's no obligation to hire us for development**. Once we deliver your project charter and backlog you can use it to shop around for development teams to actually build your software for you.

If you'd like to discuss Origin in more detail, get in touch at contact@happybearsoftware.com and we'll set up a meeting. We look forward to hearing from you.