Menu

4 DECEMBER 2017 / #AGILE

How to effectively scope your software projects

by Angela Zhang

Since starting my career as a software engineer, I've learned that scoping is one of the hardest things to get right. Unfortunately, CS programs in universities don't really teach you how to scope projects. So here's my attempt at consolidating what I've learned on this topic.

Scoping isn't something that you can spend a day on during the project and never think about again. In fact, to scope a project accurately, you need to pay attention throughout the project during:

and its goals

2. The scoping phase: the time when most people think about scoping. This is where you try to list out the work that needs to be done given the project goals, and estimate how much time will be required to do them.
3. The execution phase: when you are actually implementing the project.

The planning phase

One of the most important things to do during this time is to **define very specific goals for the project**. For example, instead of "improve X to be more scalable and performant," a better and more specific goal might be to "improve X by adding unit tests, supporting 20K queries per second, and reducing capped mean of user latency to <= 200ms."

Having very specific goals allows you to ruthlessly cut anything that does not contribute to these goals, so that you don't suffer from feature creep. Along these lines, you might also consider explicitly defining **anti-goals**, and separating **must-haves** and **nice-to-haves**.

Minimize the batch size of the project by (1) setting up clear milestones and checkpoints for the project, and (2) making it easy to launch only part of the project. Not only does this help break down the project, but it will also allow the team to pause or cut the project early if another, higher priority task comes up.

De-risk the project as soon as possible. Two common ways of de-risking a project include (1) working on the riskiest parts upfront, and (2) prototyping the riskiest parts using dummy data and/or scaffolding. Whenever a new open-source system or external service is used, that usually represents a big risk.

Don't optimize for total amount of work done. Instead, **optimize for total amount of impact over time**. Once you've gotten the riskiest part out of the way, prioritize working on the part of the project that would result in the highest amount of impact immediately.

Here's one way to think about this: plot the impact of a project over time, where the Y axis is impact, and the X axis is time. At the start of the project, the impact is 0%, and at the end of the project, the impact is 100%. You want to maximize the area under the curve by doing high ROI tasks first.

Try to **avoid rewriting big systems from scratch as much as possible**. When doing a rewrite, we tend to (1) underestimate how much work it would be, (2) be tempted to add new features and improvements, and (3) build an overly complicated system because we are too focused on all the shortcomings of the first system.

Instead of doing a big rewrite, consider incrementally replacing subsystems. Have good abstraction layers that support swapping out one part of the old system at a time, so you don't need to wait for everything to be done to test the new system.

The scoping phase

- Only the **engineers writing the code should be the ones scoping** the tasks. Not their managers, not the PM, or the key stakeholders in the company.
- **Resist the temptation to under-scope**. Be honest about how long tasks will take, not how long you or someone else (such as your manager or the Go To Market team) wants them to take.

- **Divide the project into small tasks, each taking two days or less**. When you have tasks that are scoped to "**roughly 1**

enumerate all the subtasks that you might need to do.

- Define **measurable milestones** to get to the project goal. Schedule each with a specific calendar date representing when you expect this milestone to be reached. Then, establish some sort of external accountability on these milestones by, for example, committing them to your manager and setting up milestone checkins.
- **Think of project time estimates as probability distributions**, not best-case scenarios. Instead of telling someone that a feature will be finished in 6 weeks, tell them something like "there's a 50% likelihood of finishing the feature in 4 weeks, and a 90% chance we'd finish it in 8 weeks." This tends to force people to be more realistic.
- **Add buffer** to account for: (1) Dev time != calendar time, due to meetings, interviews, and holidays. I usually multiply the dev time by 1.5 to get to the calendar time. (2) Unexpected project tasks time, since there are always tasks that you didn't realize you need to do until much later, such as refactoring old code, debugging seemingly unexplainable behaviors, adding tests. The more experienced you are at scoping, the smaller this multiplier would get.
- **Use historical data.** Keep track of whether you've tended to overscope or underscope projects in the past (most people tend to underscope). Adjust your scoping accordingly.
- Keep in mind that **2X the number of people does not mean 1/2X the dev time**, as a result of communication overheads, ramp up time, etc.
- Consider **timeboxing open-ended parts of the project**. Instead of "find the best stream processing framework for this system," which can take months of research and prototyping, timebox it to "find a suitable streaming

to balance this against incurring long term operational overhead.

The execution phase

Re-scope regularly during the project execution. For each task, track how much time you estimated the task would take, then how long it actually took you to complete it. Do this for every measurable milestone. If your scoping is off by 50% for the first parts of the project, odds are your scoping will also be off by 50% for the rest of the project.

Use milestones to answer "how's the project going?" As engineers, it's tempting to answer "it'll be done by next week" or "until end of this month." But these types of vague answers tend to create projects that are *always* 2 weeks away from being finished. Instead, use the (re-scoped) milestones to give a concrete answer based on how much work is left.

If the project slips, make sure everyone understands why the project has slipped. Then, **develop a realistic and revised version of the project plan**. Dropping the project or cutting it short is a potential option that should be considered. Read more about The Sunk Cost Fallacy if you are curious why people tend to bias against cutting a project half way.

Giving credit where credit is due, a lot of information here comes from talking to engineers and managers like Spencer Chan and Nikhil Garg, reading books like The Effective Engineer by Edmond Lau, and personally scoping lots of projects with varying degrees of accuracy.

Lastly, if I'm being honest, I am by no means an expert on scoping and I still regularly make mistakes like not following some of the

so far so I can refer back to this in the future.

If you like this post, follow me on Twitter for more content on engineering, processes, and backend systems.

If this article was helpful, tweet it or share it.

Donate $5 and buy the world 250 hours of learning.

Countinue reading about

Agile

A radically simple approach to user stories

Project budgeting: an anti-pattern

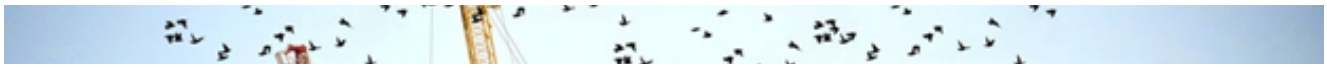
Use cases and organizational structure

See all 29 posts →

#DISTRIBUTED SYSTEMS

Building on Quicksand: A Summary

2 YEARS AGO



Menu

#PROGRAMMING

Don’t worry, be happy: How to build your future tech career in 5 simple steps.

2 YEARS AGO

Menu

Privacy Policy
Terms of Service
Copyright Policy

Git Tutorial
React Tutorial
Java Tutorial
Linux Tutorial
CSS Tutorial
jQuery Example
SQL Tutorial
CSS Example
React Example
Angular Tutorial
Bootstrap Example
How to Set Up SSH Keys
WordPress Tutorial
PHP Example