Implementation Planning vs. Developing: A Mental Model I Keep Coming Back To

A practical reflection on how implementation planning can reduce development effort - without pretending the chart is a scientific model.

en-US - June 11, 2026 - Software Development / Planning / Engineering

This is not a study. It is not a benchmark. It is definitely not me pretending that a chart can explain every engineering decision.

It is just a mental model I keep coming back to after working on features, refactors, integrations, migrations, and larger changes where the real challenge was not only writing the code, but understanding what the code was about to touch.

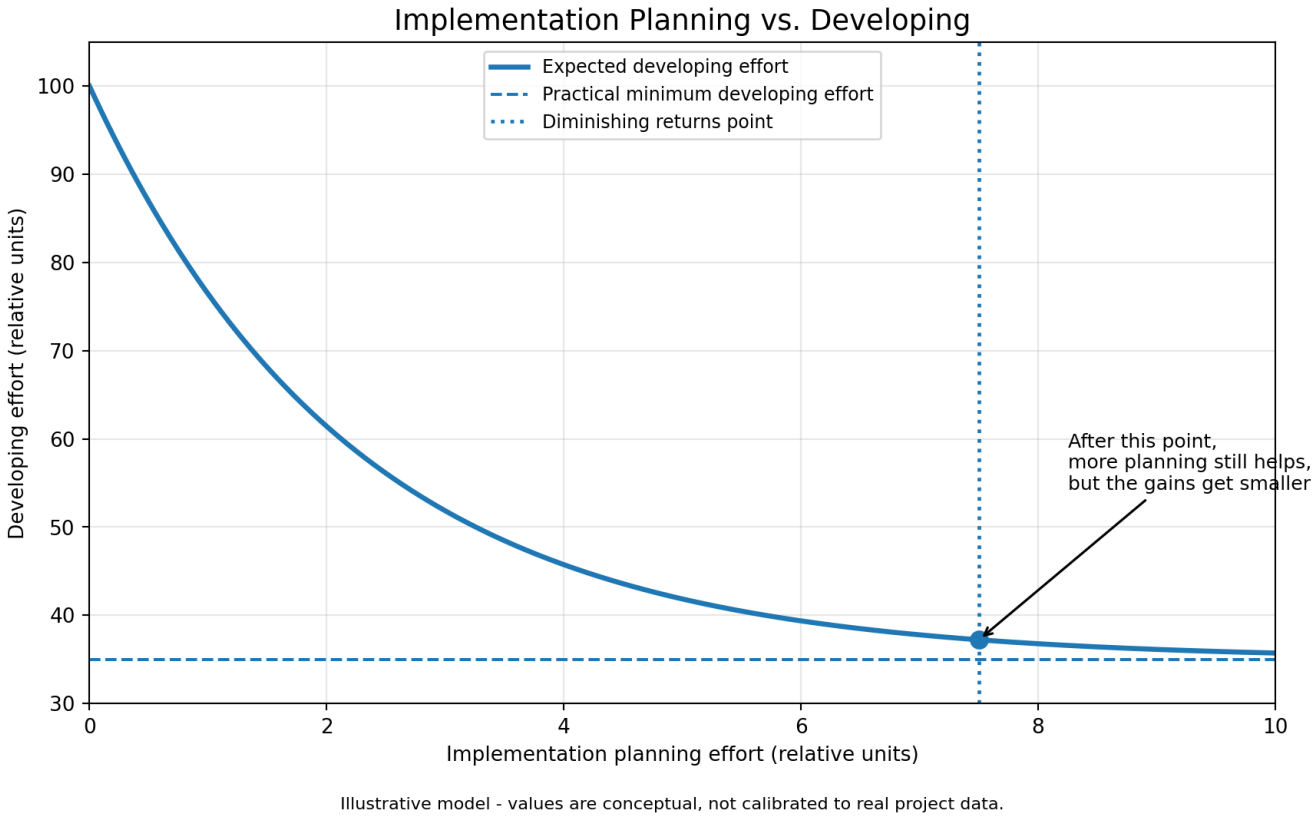
The idea became clearer to me after a change in the way our team started organizing some work. On the project I work on, we began creating tasks specifically for **implementation plans** before starting the actual development tasks. In some cases, the development task would only be created, refined, or properly broken down after the implementation planning task was finished.

At first, that separation felt a bit unusual to me. Part of my brain wanted to jump straight into the code, because that is where things feel concrete. But after seeing this process happen a few times, I started to understand the value of treating the plan as its own piece of work. Not as bureaucracy. Not as a ceremony. More like a pause to understand the terrain before committing to a route.

When I say **Implementation Planning**, I mean the work that happens before the actual hands-on building: understanding the impact, checking system boundaries, thinking through the structure, reading existing code, identifying patterns, listing test cases, and exposing the boring-but-important manual steps that can surprise the team later.

When I say **Developing**, I mean the hands-on part: writing code, adjusting tests, reviewing, integrating, debugging, deploying, and dealing with the things that only become visible once the change starts to exist.

The way I picture the relationship looks something like this:



Illustrative model - values are conceptual, not calibrated to real project data.

The numbers are not the point. They are relative, illustrative units. The shape is the useful part.

The simple idea

The more useful implementation planning we do, the less unnecessary developing effort we usually create later.

Not less developing effort in the sense that coding becomes magically easy. It does not.

But less effort wasted on avoidable confusion.

Less rework caused by a hidden business rule. Less back-and-forth because the ownership of a flow was unclear. Less late discovery of an integration detail. Less "oh, this also affects that other service." Less guessing while already inside the implementation.

That is where planning helps a lot.

A good implementation plan is not a contract with the future. It is more like a map. It does not remove every surprise from the road, but it helps the team avoid walking in circles.

Why a separate planning task can help

One thing I like about having a dedicated implementation planning task is that it gives the team permission to slow down for the right reason.

Without that explicit step, planning often happens in fragments: a quick comment here, a Slack thread there, a small investigation during development, and a few important details discovered only after the work has already started. Sometimes that is fine. But for larger changes, those scattered discoveries can quietly become expensive.

A separate planning task creates a small container for the unknowns. It gives the engineer time to read the code, ask questions, check assumptions, and turn a vague development task into something the team can actually reason about.

Sometimes the output is not a huge document. It might be a short plan, a list of affected areas, a proposed implementation sequence, or a few risks that need alignment. The value is not in producing a beautiful plan. The value is in making the next task less blind.

In that sense, the development task that comes after the plan is usually better shaped. It has clearer boundaries, better test expectations, fewer hidden manual steps, and a more honest understanding of what might go wrong.

What planning is actually trying to uncover

For me, the most valuable part of planning is not creating a long document. It is forcing the right questions to appear before the cost of changing direction gets too high.

A feature or a large technical change usually carries more context than the ticket shows. There are system boundaries, business rules, legacy decisions, code patterns, historical compromises, rollout details, and sometimes a few "please do not touch this unless you know why" areas.

Implementation planning is the moment where I try to pull those details closer to the surface.

Things like:

- What systems, modules, jobs, events, or APIs will be affected?
- What is the proposed implementation structure?
- Which existing code patterns should guide the solution?
- Which patterns should probably be avoided?
- What business rules or legacy behaviors can change the direction of the work?
- What test cases need to be covered?
- What edge cases are easy to miss?
- What manual steps are involved?
- Are there scripts, migrations, backfills, feature flags, config changes, or rollout steps?
- What needs to be checked after the release?
- What can be automated, and what still needs human verification?

The more complex the system, the more these details matter.

In a clean and isolated codebase, maybe the plan can be very small. But in a real product with history, dependencies, legacy logic, business-specific rules, and previous technical decisions, implementation planning becomes less about writing a plan and more about reducing the number of expensive surprises.

Why the first part of the curve drops fast

At the beginning, planning tends to have a high return.

Even a short conversation, a small design note, or a quick exploration of the code can remove a surprising amount of uncertainty.

Sometimes one hour of planning saves many hours of developing because it prevents the team from starting in the wrong direction.

That is the steep part of the curve.

The team discovers that a flow already exists. Or that a field has a hidden meaning. Or that a dependency behaves differently in production. Or that a similar change caused problems before. Or that the implementation can be much simpler if it follows an existing pattern.

This is the kind of planning that feels valuable immediately. It turns "let's start coding and see what happens" into "we know the main path, the main risks, and the first few decisions."

Why the curve eventually flattens

But planning has limits.

At some point, more planning still helps, but the gain becomes smaller. That is the flattening part of the curve.

There are a few reasons for that.

First, developing can never go to zero. Even with a great plan, someone still needs to write the code, test it, review it, integrate it, deploy it, and fix the things that only show up when the system is exercised.

Second, some uncertainty is not removable upfront. Some questions only become real when the code meets the existing system. You can read, inspect, and discuss a lot, but sometimes the honest answer is: we need to build a small part and validate it.

Third, too much planning can start giving the team a false sense of certainty. The document gets bigger, but the risk is not necessarily getting smaller at the same pace.

That is why I do not see planning as something to maximize.

I see it as something to calibrate.

A small equation for the mental model

I like this equation as a simple way to represent the idea:

```
Developing(P) = D_min + (D0 - D_min) * e^(-kP)
```

Where:

- `P` is the implementation planning effort
- `Developing(P)` is the expected developing effort
- `D0` is the developing effort when planning is close to zero
- `D_min` is the practical minimum developing effort
- `k` is how strongly planning reduces uncertainty and rework

This is not a formula I would use to estimate a real sprint.

I would not put these numbers in a spreadsheet and pretend they are precise.

The equation is useful only because it captures the shape of the intuition: planning reduces developing effort quickly at first, then the return gets smaller over time.

The two traps

The first trap is under-planning.

This is when the team starts developing before understanding enough of the impact, dependencies, structure, tests, rollout, and system-specific constraints. The missing planning does not disappear. It moves into the development phase, where it becomes interruptions, rework, and last-minute discoveries.

The second trap is over-planning.

This is when the team tries to answer every possible question before building anything. The plan gets longer, but the team is not always getting proportionally safer. Sometimes the next useful learning step is not another meeting or another diagram. It is a small implementation spike, a test, or validating an assumption in the code.

Both traps are understandable.

Under-planning often comes from pressure to move fast. Over-planning often comes from a desire to avoid mistakes.

I have done both. Most teams probably have.

The hard part is finding the middle.

What enough planning feels like

For me, enough planning usually feels like this:

The team understands the main path. The risky areas are visible. The implementation structure is not a mystery. The code patterns are known. The test strategy is clear enough. The manual steps are listed. The rollout does not depend on hope. The unknowns that remain are acceptable and intentional.

That last part matters: acceptable and intentional.

A good plan does not eliminate every unknown. It makes the remaining unknowns explicit enough that the team can decide to carry them into developing.

That is a very different feeling from discovering them by accident later.

My current takeaway

Implementation planning is a way to move uncertainty earlier, when it is usually cheaper to discuss, challenge, and adjust.

It helps developing become more focused, but it has diminishing returns. The goal is not to create the most complete plan possible. The goal is to create enough clarity to build with confidence.

For me, the sweet spot is somewhere between chaos and theater.

Not "let's just code and figure it out."

Not "let's plan until nothing feels uncertain."

More like:

"Let's understand enough to make the next decisions responsibly, and then let the implementation teach us the rest."

That is the balance I am still trying to get better at.