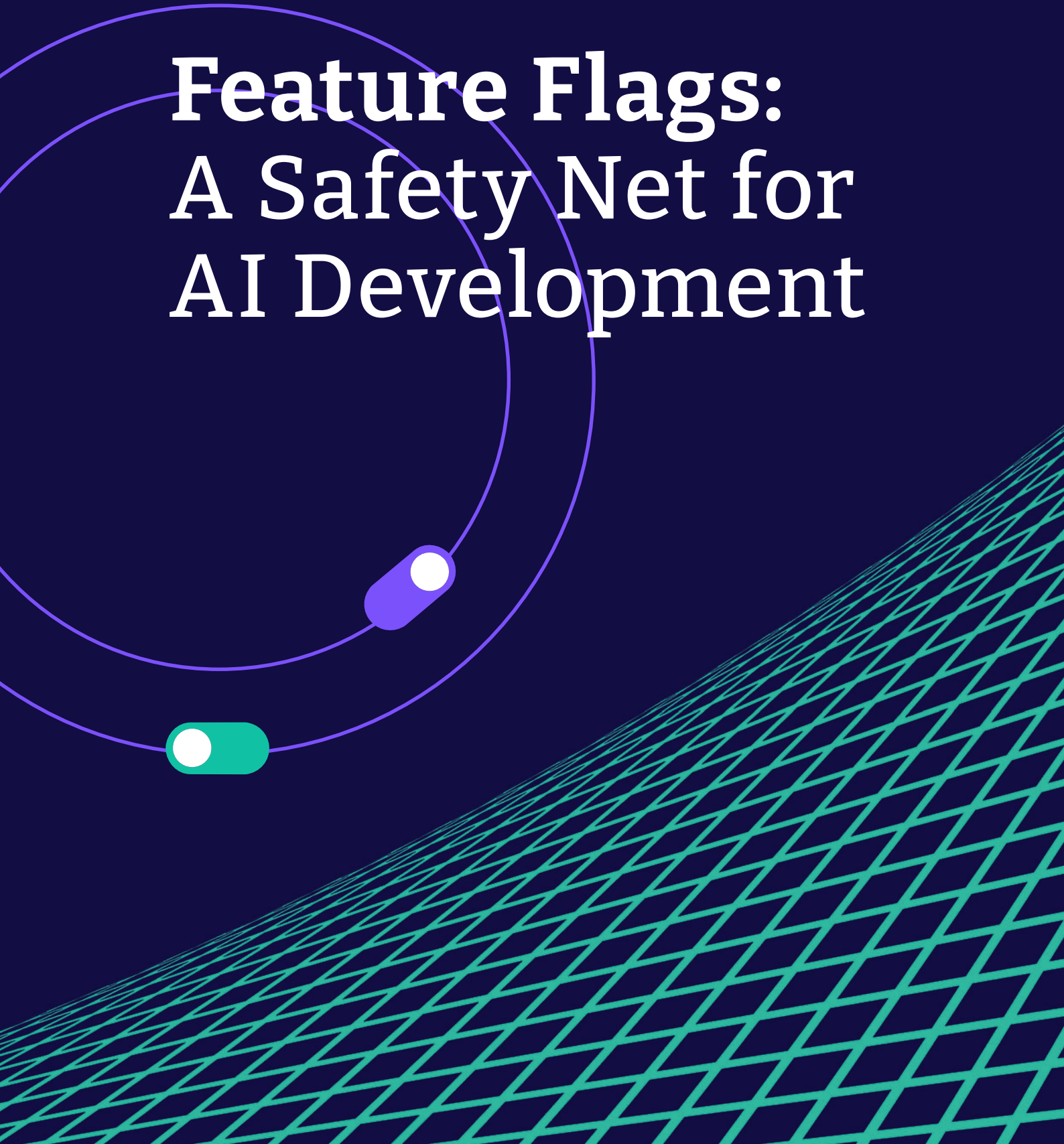


FLAGSMITH PLAYBOOK

The developer's guide to controlled AI rollouts

Feature Flags: A Safety Net for AI Development



Foreword

Generative AI exploded into our collective consciousness with the launch of ChatGPT in 2022. Initially, I was skeptical of practical use-cases for AI—it was a neat parlor trick, but not much more. This certainly seemed the case when it came to software development: early coding assistants' auto-complete suggestions were sometimes shockingly impressive, but overall, they got in the way more often than they helped.

However, this changed quickly. By 2024, the capabilities of AI models had grown significantly, while the introduction of agents delivered a dramatic shift in the capabilities of AI coding.

Now, as I write this in early 2026, it is clear that AI-assisted engineering represents a tectonic shift in our industry, fundamentally altering the economics of software. It's undeniable that writing code just got a lot cheaper.

However, building software involves a lot more than writing code. I spend a lot of time talking to engineering leaders, and one of the most prominent themes in these discussions is that AI has NOT dramatically reduced the overall cost of delivering features; it's just moved the bottleneck. Engineers are using AI to produce code at a faster rate, but those code changes are now hung up in code review, or testing, or deployment. Similar challenges exist earlier in the value stream—product managers and designers will struggle to keep AI-augmented engineers' backlogs stocked with thoughtfully prioritised, well-defined requirements.

This is the big challenge facing engineering leaders in the AI era: how do we optimise the full value stream from ideation, through implementation, and on to true customer value? These challenges are particularly pressing for larger organisations that maintain legacy codebases and operate in regulated environments. Succeeding with AI-assisted engineering in these contexts requires a very different approach than vibe-coding a greenfield app.

In this book, we'll tackle this problem head-on. We'll take the well-established principles of Progressive Delivery and apply them to an AI-powered SDLC. We'll also look at how AI and feature flags can work together to supercharge experimentation, radically accelerating our learning cycle and helping us build better features, faster!



Pete Hodgson

Fractional CTO & OpenFeature
Governance Board Member



Content

Chapter 1

Progressive Delivery for AI-Led Development

04

Chapter 2

AI-Accelerated SDLC

09

Chapter 3

AI-First Experimentation

19

Chapter 1

Progressive Delivery for AI-Led Development



AI is having a huge impact on how engineers build software. But when we look through the lens of the full software delivery lifecycle (SDLC), the productivity boost from AI has been mostly limited to that implementation phase.

Of course, there's more to building product features than just writing code—there's all the downstream activities around reviewing, testing, and releasing said code, as well as all the upstream activities around prioritizing features, analyzing and defining requirements, designing user experiences, and so on.

If we imagine our SDLC as a production line in an industrial bakery, the arrival of AI-assisted engineering is akin to upgrading the bread oven in the middle of the factory. We can now bake dough into bread faster, but unless we improve other parts of the factory, we won't gain the improvements we might hope for from the system overall. If we're not producing dough fast enough, then that fancy

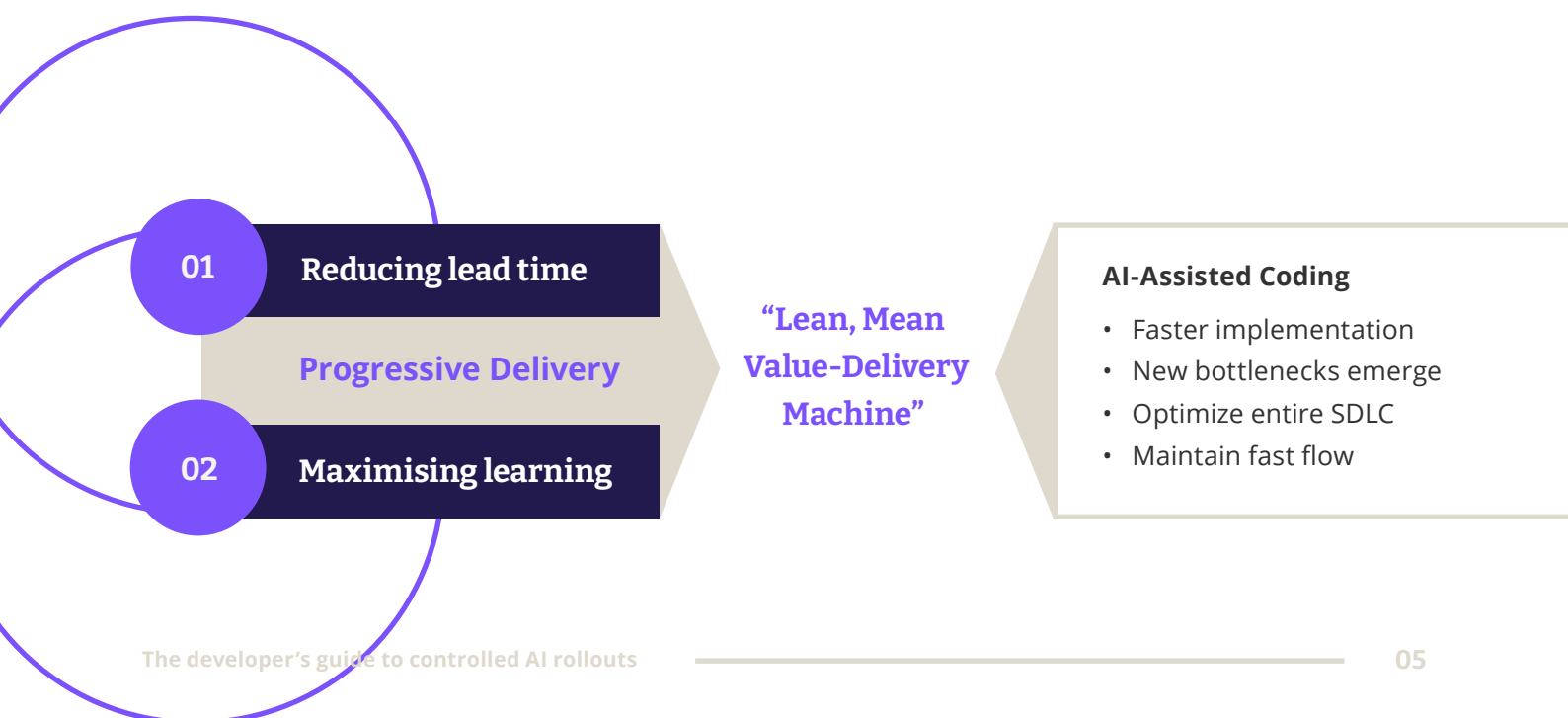
new bread oven may sit idle. If we can't package the baked bread fast enough, then the speed of the upgraded oven doesn't really help.

Solving these systemic optimization problems is a common challenge in the world of physical manufacturing, and for the last half a century, the standard toolkit has been an approach known as Lean Manufacturing.

In the world of software, pioneering organizations have taken those Lean ideas and applied them to improving the SDLC, leading to a set of practices known as Progressive Delivery.

Faster feedback, faster learning

Progressive Delivery is built around two core principles:



Why those two principles? First, by optimizing our SDLC to reduce lead time, we increase throughput and can deliver more features to our users. Second, by layering in techniques like A/B testing, we can be sure that the features we're delivering are actually valuable to those users. When combined, Progressive Delivery practices turn your engineering organization into a lean, mean value-delivery machine.

How does this apply to AI? The practices in Progressive Delivery that reduce lead time can help us to deal with the shifting bottleneck that AI-assisted coding creates—we can optimize the flow of work up- and downstream of the implementation phase.

Additionally, AI can supercharge our ability to learn and experiment—a topic we'll cover in detail in Chapter 3.

Increasing flow by reducing batch sizes

Central to Progressive Delivery is the idea of increasing flow by reducing batch sizes. Rather than shipping a big batch of changes once a month or once every two weeks, we aim for a steady stream of small changes flowing into production at a much more frequent pace—ideally daily, at least.

By doing so, we minimise wait states, where completed work sits waiting for the next scheduled deployment. Additionally, because our batches are smaller, they're easier to test, and if we find issues with a given deployment, we have a much smaller set of changes to inspect.

Safely achieving this rapid pace of deployment requires a set of technical practices, chief among them being feature flags.

In its simplest form, a feature flag is a dynamic if/else statement. It allows us to change control flow at runtime, turning a feature on or off without needing to deploy code, make configuration changes, or even restart any processes. This capability has a variety of different use cases, but when it comes to Progressive Delivery, the main one is in *separating deployment from release*.



No long-lived feature branches

Traditionally, when coding up a feature, we work on a feature branch, keeping our changes separate from our main branch until they are complete and ready for testing. With feature flags, this isn't necessary, as our changes can be hidden behind a flag. We can merge and deploy our changes as we go, rather than landing all of them as one big merge.



Increased deploy tempo

When a new feature is protected behind a feature flag, we no longer need to perform full validation of every code change in a given deployment. We reduce the cost and risk associated with each deployment, unlocking a faster deployment tempo.



Continuous testing

Because we're deploying more frequently and we're able to enable a feature flag whenever we want, we can start testing a feature as soon as it's ready, rather than on a fixed "sprint cycle" cadence. Moving from batches to single-item flow is one of the best ways to reduce lead time.



Testers in production

Feature flags allow us to turn on features for specific users. This means that we can turn a feature on in production, but only for testers. By testing in production, we get more confidence that we're testing the exact set of software and data that our customers will experience.

Feature flags allow us to ship “latent code” to production—changes that are deployed, but hidden behind a disabled flag. This brings a whole set of benefits.

We also reduce the need for pre-production testing environments, which can have a major positive impact on lead time—we have the option of removing an entire set of deploy and test activities from our SDLC!

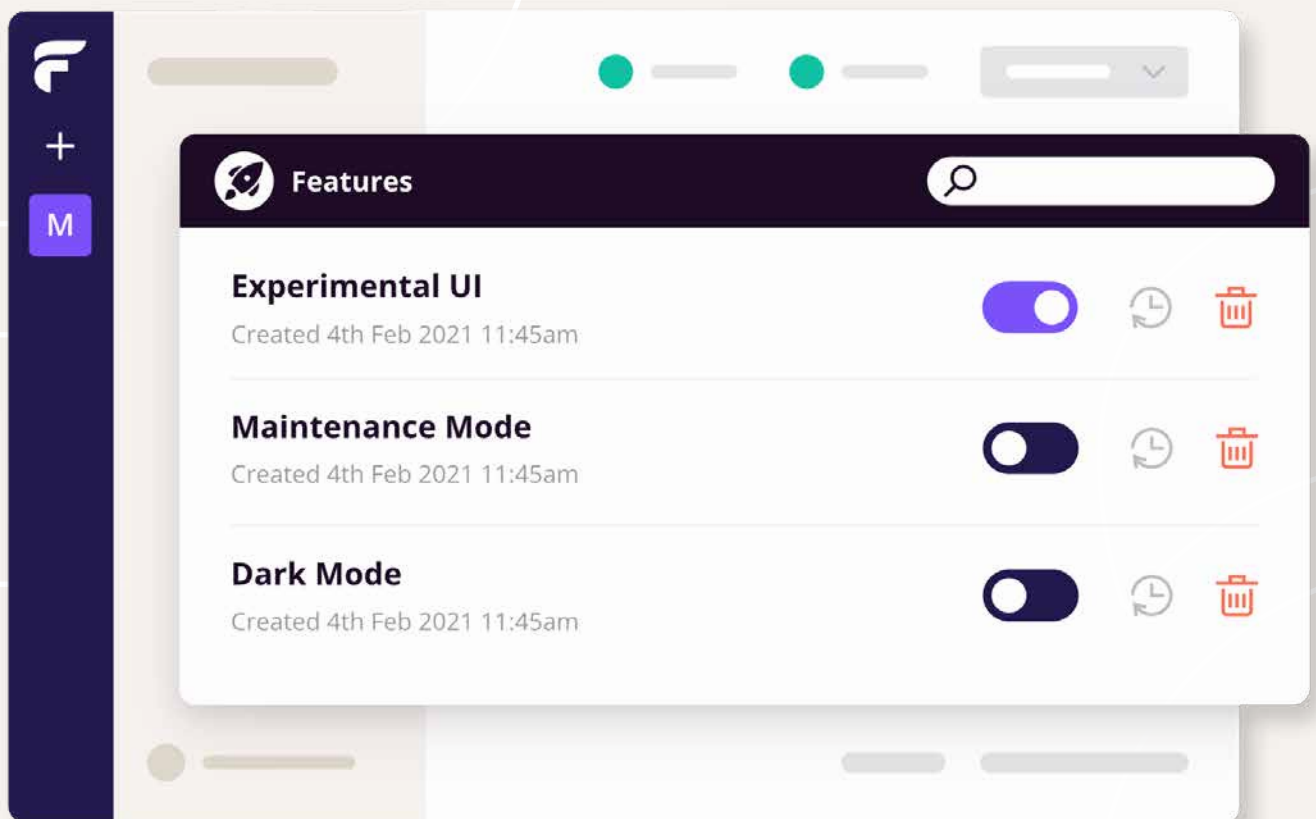
All of these benefits reduce lead times and/or enable a faster release cadence.

Unlocking experimentation at scale

Feature flags are also a necessary foundation for product experimentation. To test the impact of a new feature, we can place it behind an A/B test—a feature flag that is on for some users and off for others—and then measure what impact the feature has on those different user populations.

A product with a mature feature flagging practice can both iterate rapidly and experiment cheaply. This is the promise of Progressive Delivery: a lean,

efficient SDLC that can drive incremental product improvement via data-driven decisions.



Chapter 2

AI-Accelerated SDLC

SDLC



In the last chapter, we talked about how Progressive Delivery practices can improve the efficiency of our software delivery lifecycle by reducing lead times. In this chapter, we're going to look at another complementary approach to optimising software delivery: reducing friction and toil using AI.

Automating away toil



“Toil”, as defined by Google engineers in the Site Reliability Engineering book, is work that is “...manual, repetitive, automatable, tactical, devoid of enduring value...”

In the context of software delivery, it's the rote manual tasks you go through to move a feature through the machinery of your delivery process. The rituals involved with starting a new ticket, merging a PR, creating a feature flag, marking a feature as ready for testing, etc.

Toil is work that doesn't really require human skill or judgement, but where the details of the tasks are variable enough that it's not trivial to automate. This is also the type of work that's a prime candidate for handing off to a coding agent.

It's become clear that coding agents can radically reduce the cost of writing code, especially when the implementation work is straightforward and doesn't need a lot of context or design judgement. What a lot of software teams are starting to discover is that these agentic systems are also very capable of accomplishing other rote tasks when given clear direction and the appropriate tools.

Let's look at a simple example of toil that an AI can automate: the steps an engineer might go through when picking up a new ticket from the backlog

Step 1

Move the ticket into “In Development” in the team's project management system

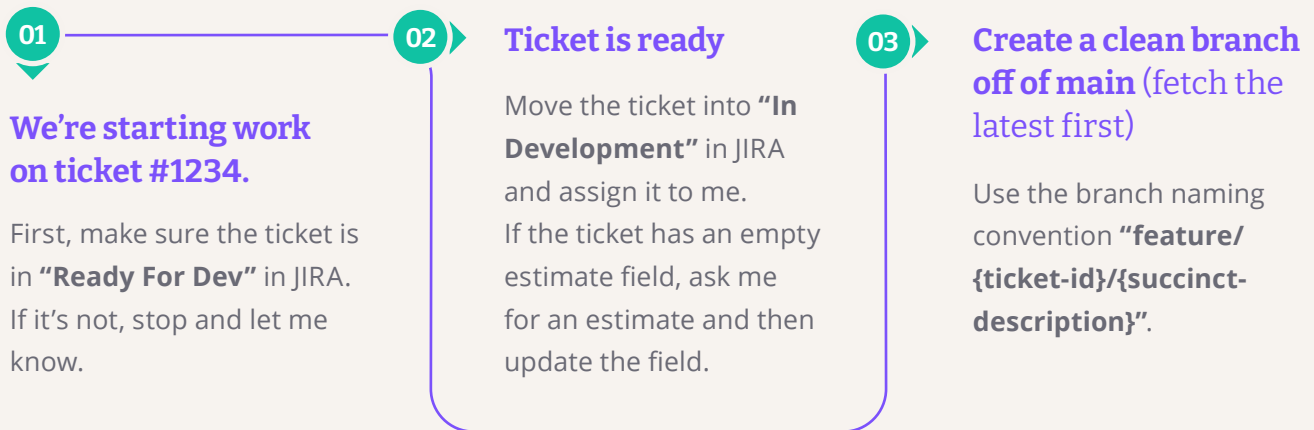
Step 2

Add a rough estimate, if it hasn't already been specified

Step 3

Check that the local repo's working tree is clean, and create a new branch using the team's naming conventions

The process isn't complicated, but it's tedious. It's something a coding agent can do on your behalf. A prompt like this would get you most of the way there:



This prompt offloads all of the toil for this process to an AI, with the engineer only doing the thinking part: coming up with an estimate of effort for the feature.

Coding agents all have mechanisms that let you save these sorts of repeated prompts as a "skill" or similar, which can then be shared within a team. There's a one-time cost to create the prompt and configure the coding agent with a tool that gives the agent access to the project management system.

This example is very simple, but it illustrates a broader opportunity: most tedious work that you'd be tempted to offload to a new intern can likely be offloaded to an agent.



Reducing cost opens up new options

The economics of using a coding agent to attack toil are very different to what you might see from previous approaches to increasing engineering productivity.

Within large engineering organisations, it's very common to see internal teams with names like "platform engineering" or "developer productivity" that are focused on making product delivery teams more effective. These teams are essentially building internal products that automate away toil, but building and maintaining this custom software is very expensive, so it only makes sense at certain economies of scale.

An engineer working at Meta or LinkedIn can take advantage of highly optimised internal platforms for things like release management, experimentation, code review, and so on. But for most organisations, it's just not economically feasible to build these sophisticated internal tools, so engineers at those companies experience more toil.

AI-accelerated Progressive Delivery

For the rest of this chapter, we'll take a deeper dive into how a team can use AI to reduce friction and toil at various points of the SDLC, using feature flag management as an illustrative use case.

Silicon Valley companies like Meta, LinkedIn, and Google have long been pioneers of Progressive Delivery practices. Most have invested in powerful internal platforms to minimise the friction and toil of managing feature flags and analysing experiments. This reduced friction is understood to be a competitive advantage—it leads to more experiments, faster learning, and thus better products.



The arrival of agentic AI is leveling this playing field. It's now much, much cheaper to automate away toil using AI, allowing engineering teams across the industry to work within the same sort of streamlined SDLC processes as big tech companies.

In contrast, more traditional enterprises don't invest as heavily in this area. Experiments are expensive to set up, and managing the rollout of a feature flag requires a lot of toil. These organisations naturally tend to experiment less and iterate slower.

Let's look at how agentic AI changes that dynamic and allows enterprise software teams to perform more like a startup. We'll walk through all the stages of progressive delivery for a simple feature, using an example. Specifically, we're a web experience team at a large bank, and we want to experiment with a "go paperless" widget on the

homepage that makes it easy for customers to opt into paperless banking statements.

Beyond actually implementing this feature, there'll be a number of small chores related to feature flagging and A/B testing this feature. Let's walk through each of those to see how AI can reduce the toil.

01. Adding the feature flag

Early in the feature's development, an engineer will need to place the feature behind a feature flag. This step involves adding the flag decision point in the code (an if-else statement), creating the flag in the flag management platform, and noting that this work is feature-flagged in the project management system.

Before

An engineer spends 5 minutes coding up the decision point, 10 minutes creating the flag via the flag management UI (including making sure it's configured "off" in all environments), and then 5 minutes adding a "feature-flag-name" field to the ticket in JIRA.

Toil time: 15 min



After

The engineer prompts their coding agent: *"Hide this new widget feature behind a feature flag called paperless-homepage-widget. Create the flag in Flagsmith too, turned off in all environments. Fill out the 'feature-flag-name' attribute in Jira (the ticket number is in the branch name)".*

Toil time: 1 min



Note that this assumes that the coding agent has the correct tools available. Typically, this would mean an [MCP server](#) for your flag management system (e.g., [Flagsmith's MCP server](#)) and for your ticketing platform.

Also note that for this sort of common workflow, this prompt would likely be extracted into a reusable [agent skill](#), meaning that a more realistic prompt would be "Use the feature-flagging skill to hide this new widget feature. Flag is paperless-homepage-widget."

02. Internal QA

Time passes, and the feature is ready for internal testing.

Before

A tester would start testing the feature. Hopefully, they're already aware that it's protected by a feature flag and are able to find the flag name in the feature ticket. They open up their flag management UI and add a rule that turns the flag on just for their test user. They test the feature, and once they're done, remove the custom rule from the flagging platform.

Toil time: 30 min



After

A tester prompts their coding agent:
"I'm testing ticket #1234. If the feature isn't generally available in staging, then please turn it on, but just for my shane@example.com test user". Once they're done testing, they tell the agent, "OK, I'm done testing. Remove that custom rule, please, and move the ticket into Ready For Launch."

Toil time: 1 min



03. Canary release

The product manager for our new “go paperless” feature feels that it’s ready to go live, but we want to do that in a controlled way. She’d like to start with a canary release to 5% of users, just to make sure there are no unexpected issues in production.

Before

The product manager looks up the feature ticket and the associated feature flag name. They then look up the feature flag in their flag management UI and add a canary release rule that turns the feature on for 5% of users. They then add a “canarying” tag to the feature ticket.

Toil time: 15 min



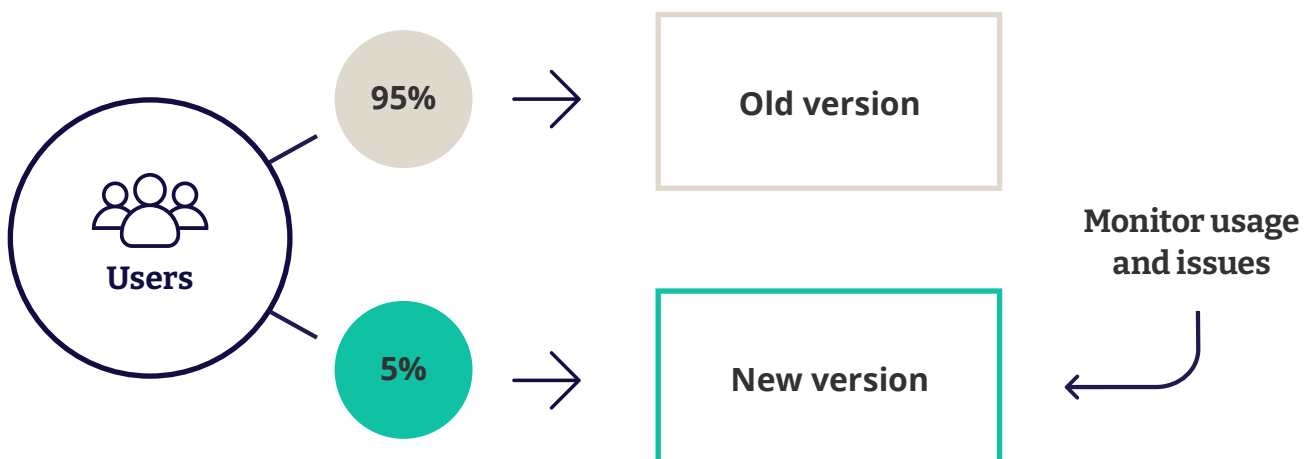
After

The PM prompts their coding agent:

“I’m ready to release the go paperless widget. Please confirm its feature flag is currently turned off in prod, and then turn it up to 5%. Then add a ‘canarying’ tag to the associated ticket.”

“Before you make ANY changes, please double-confirm with me that you have the correct feature-flag!”

Toil time: 1 min



04. A/B test



The canary release proceeded without incident, and our PM is now ready to roll the widget out as an A/B test so that she can measure the business impact of the change.

Before

The product manager looks up the feature flag in their flag management UI, removes the 5% canary rule, and replaces it with an A/B test.

"They also update the feature ticket and replace the "canarying" tag with an "experimenting" tag."

A week later, they look up the A/B test in their experimentation platform to see if it's reached statistical significance. It has, and the widget has succeeded in increasing the opt-in rate by 12%, and does not appear to impact any other KPIs.

The PM is ready to conclude the A/B test. They update the rules in the flag management UI and remove the "experimenting" tag from the ticket.

Toil time: 1 hour



After

The PM prompts their agent: *"I want to move the go-paperless widget feature from canarying to an A/B test. Update the feature flag configuration to a 50/50 test. Also, update the labels in the associated ticket from 'canarying' to 'experimenting'."*

Check with me that you have the right feature flag before you make any changes!

A week later, they ask the agent: *"Is the go-paperless experiment at statistical significance yet?"*

The agent uses various tools (including an MCP server for the experimentation platform) to look up the correct feature flag name and check on the associated experiment. It reports back that the experiment is complete, with a deep link to the experimentation platform so the PM can check out the details.

The PM is now ready to conclude the experiment. They prompt their agent: *"Please move this feature flag to GA, and remove the 'experimenting' tag from the ticket."*

Toil time: 10 min



05. Cleanup



A month later, the engineering team is doing its regular cleanup of obsolete feature flags. This involves two operations: checking for stale flags—flags that have been fully released but whose decision points have not been removed from the product—and checking for dead flags—flags that have been fully released and are no longer being actively evaluated because their decision points have already been removed.

Before

An engineer starts by looking for stale flags. They log into the flag management platform and pull a list of flags that have been fully released for more than two weeks but are still actively being evaluated.

For each flag, they:

- Search through the source code for any remaining flag decision points
- Create a PR that removes those decision points (and refactor stale code as necessary)

Next, the engineer cleans up dead flags by pulling a list of flags that are fully released and have not been evaluated for over two weeks. They do a quick manual scan of the flag names, then remove each flag from the flag management system.

Toil time: 4 hour



After

The team's engineering manager uses their coding agent to clean up stale flags. They prompt: *"Show me all flags in production that haven't been modified in 90 days, have no code references, and are still receiving traffic."*

The agent returns a list of 7 flags. The EM responds with: *"For each flag, spawn a subagent and use the flag-cleanup skill to remove the flag."*

The coding agent then works in parallel to find and clean up decision points for each flag, running local verification and creating a separate PR for each one. An engineer reviews and merges each PR.

While the agent is working on the stale flags, the EM spins up a second agent session to work on cleaning up dead flags. They prompt: *"Show me a list of all flags that are fully released and have not been evaluated for over two weeks."*

They review the list of flags that the agent returns, then respond with: *"Please remove all of these flags."*

Toil time: 10 min



Lowering the barrier for progressive delivery

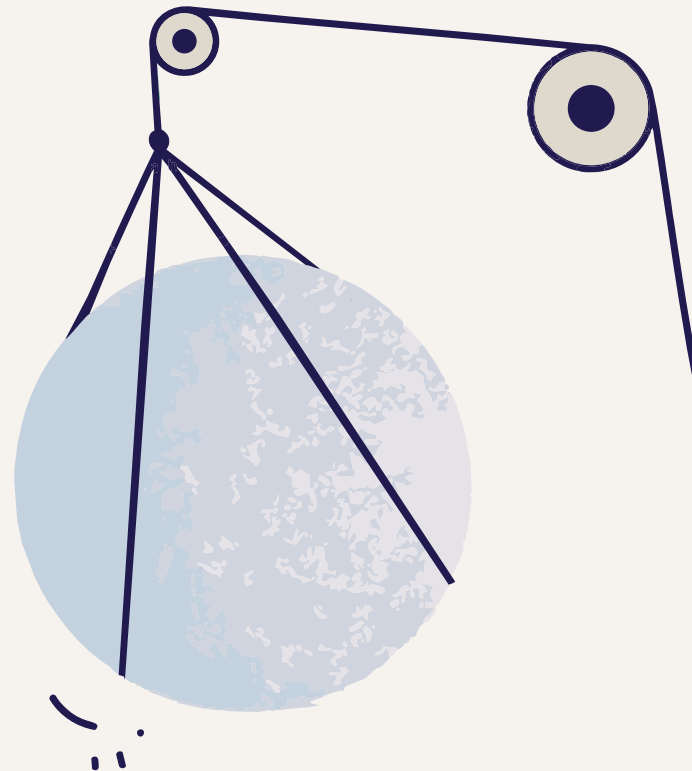
Looking across all the steps involved in this example, we reduced the overall toil involved from **6 hrs** to **23 min!**

This is a **huge reduction in overhead**, shifting the cost/benefit calculation of when to apply progressive delivery practices. When the cost of putting a change behind a feature flag becomes radically cheaper, it starts to become the default for most features.

Toil is everywhere

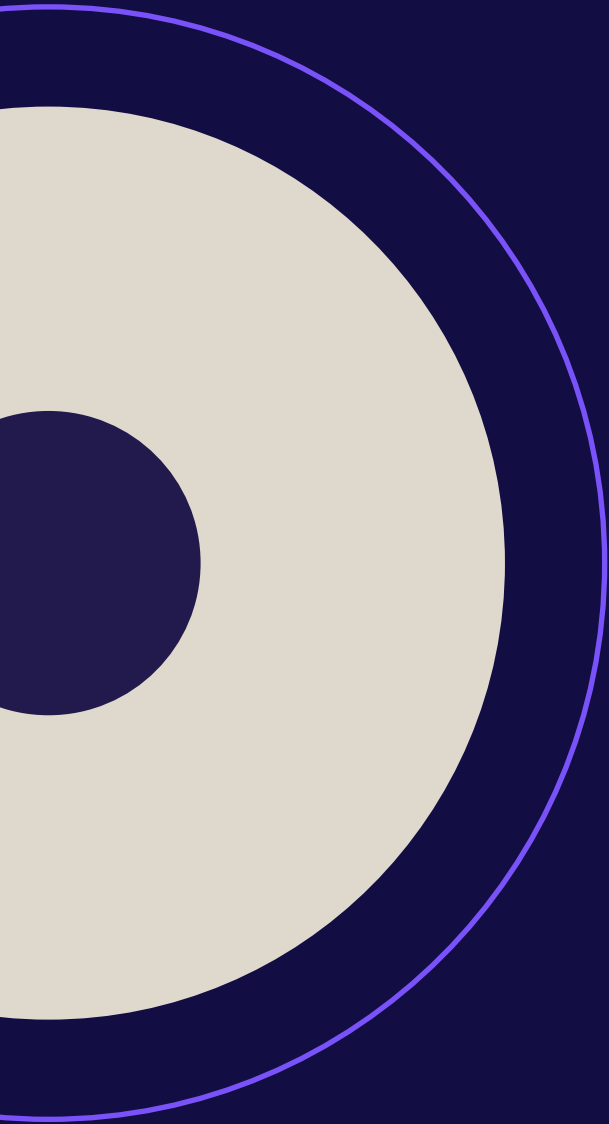
The use case we just looked at is central to product delivery, but it's just one example of how AI can reduce toil. As teams become familiar with the capabilities of agentic AI, they will discover more and more tasks like this across the SDLC.

At big tech companies, the economies of scale—plus an understanding of the value in optimizing the SDLC—drove these companies to build sophisticated developer platforms to streamline this stuff. Agentic agents with MCP-powered tools now bring a flavor of that to every organization.



Chapter 3

AI-First Experimentation



We've looked at how AI can reduce the toil and streamline the SDLC. Now let's look at how we can use AI to rapidly experiment, building rapid prototypes that let us learn quickly without compromising the engineering quality of our production systems.

AI has now evolved to the point that someone without programming experience can bring working software into existence just by describing it. You can prompt a coding agent with a description of the functionality you want, then watch as the AI builds an app from scratch. Once the app is built, you can continue to chat with the agent, refining functionality and adding new features.

This technique is known as “vibe-coding”—placing your full trust in the coding agent and not even reviewing the code it's creating. Systems like v0, Lovable, and Replit let you get a sense of this experience very easily. If you haven't tried out a system like this, I highly encourage spending some time getting a sense of what these systems are capable of.

The first time you try it out, vibe coding can be a magical, jaw-dropping experience. However,

software created this way has real limitations. While coding agents continue to improve, they're not yet capable of making all the software design decisions required to create secure, reliable, maintainable systems. Social media abounds with stories of vibe-coded apps that leaked user data, or non-technical users who watched in horror as a coding agent deleted their production databases (and then apologized profusely).

Additionally, vibe-coding only really works in greenfields—when an agent is creating an app from scratch, or making changes to a small codebase which can be easily understood in its entirety by a coding agent. Anything more complex than that needs a more hands-on, human-in-the-loop approach where an engineer helps to guide and constrain the agent.

Disposable software for rapid exploration

Despite these limitations, vibe coding should not be dismissed as something only suitable for “toy” apps.

This new way of creating software opens up some exciting opportunities for the non-engineers on a team to experiment via code. Product managers can use coding agents to quickly build fully functional prototypes as a way to rapidly iterate on a user experience, communicate a product idea, or even test out a “clickable prototype” with real users.



Taken to its logical conclusion, there are even circumstances where a team might experiment with a product feature by adding a vibe-coded change to a production app—protected by a feature flag and removed from the app once the experiment is complete.

This approach might make some engineers a little queasy, as the quality and maintainability of a vibe-coded implementation will not be great, but that's fine as long as everyone holds a shared understanding that this is “throw-away code”.

It's disposable software, created cheaply using AI with the sole intention of exploring an idea and getting feedback. The value is in the exploration and the learning, and the code itself is thrown away after the experiment.

Blazing a trail, then laying the rail

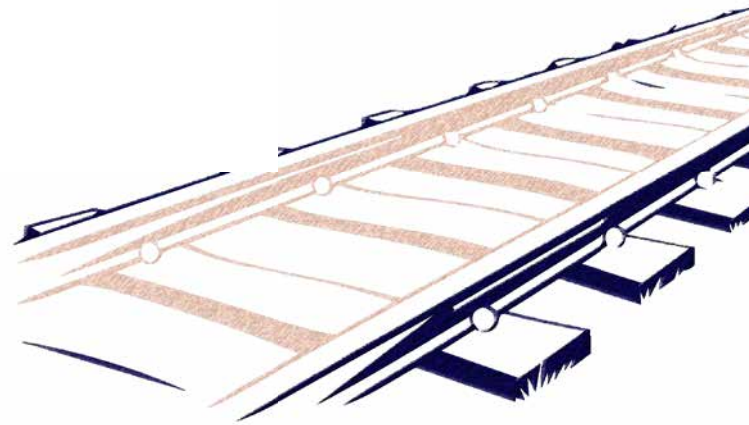
One way to visualize this throw-away-then-rebuild strategy is as *blazing a trail then laying the rail*.

If you're trying to build a railroad through unexplored jungle, it doesn't make sense to just start laying down track. The first phase is to send out some scouts to bushwhack through the undergrowth and identify the best route. That's your product managers, exploring rapidly with vibe-coded experiments.

Then, once you've identified the best path through the terrain, you can send in your engineers to replace that bush-whacked trail with a structurally sound railroad—one that can be trusted to carry a heavy load for years to come.

Product ideas that have been validated by these disposable experiments would be rebuilt using standard “professional-grade” software development practices—which may still, of course, involve a good amount of agentic coding.

In fact, a good starting point for that “professional-grade” rebuild may well be for an engineer to take that vibe-coded implementation and work with a coding agent to build a specification for a thoughtfully engineered implementation.



In this two-phased approach, we're able to do cheap, fast exploration early on. Anyone who's worked on a brand-new codebase knows just how much faster you can move when you don't have to support all the load-bearing features and architecture of a mature product.

Then, with the path identified, we can re-implement the functionality in a robust and maintainable way, confident that the time investment is worthwhile because we've already explored and validated the concept.

Trailblazing with safety

Engineering organizations need to do some work to enable this initial trailblazing phase. It requires a way for non-engineers to be able to make changes, potentially in production systems, safely.

The type of support needed depends on how tightly integrated these vibe-coded experiments are into your actual product code. Here are some different levels of integration to consider.

Interactive mockups for internal use



What it is

Rather than designing a user experience in a tool like Figma, a designer or product manager instead uses a tool like v0 to create a working interactive prototype, which can then be tested and showcased internally.



What it needs

For this use case, the implementation is fully disconnected from actual product code or production systems—no infrastructure or platform work is needed, beyond access to agentic coding tools.

Interactive customer-facing mockups



What it is

You can, of course, also show a vibe-coded interactive prototype to external stakeholders.



What it needs

If you're showcasing software, you'll need to deploy it somewhere. That means either using a vibe-coding tool that also provides hosting or setting up internal infrastructure streamlined enough for a non-technical user to deploy an app. For a strong first impression, it's worth going the extra mile on things like DNS records so you get clean URLs like "your-prototype.your-company.com".

Experimental MVP



What it is

A product manager or other non-technical stakeholder believes that a new feature set might bring value to users and wants to validate that theory by launching a very cheap MVP and tracking how it's used.

For example, a product manager for a travel booking app might want to test out the concept of including weather forecasting information during a trip planning workflow. They could do this by vibe-coding a standalone app and then having product engineers add some links to it from the booking flow and from some transactional emails.



What it needs

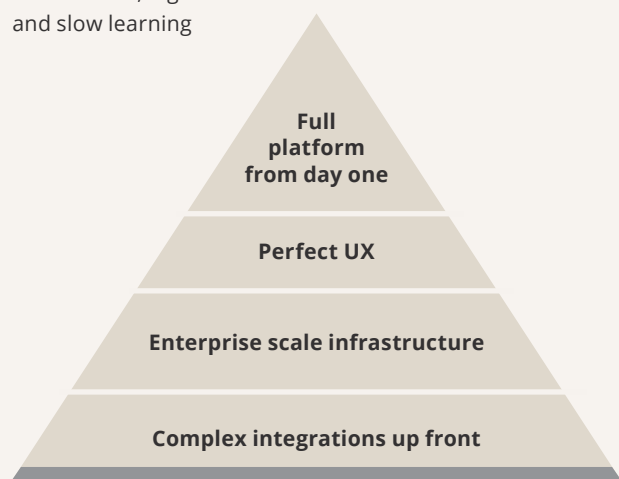
We're now entering the realm of deployed production software. This vibe-coded app needs to be deployed into company-managed infrastructure, possibly connected to authentication systems, and possibly needing secure access to internal APIs. Beyond hosting infrastructure, this MVP will need the basics of a delivery pipeline—source control, some sort of basic CI/CD, etc.

Some small changes will also be needed in the main production applications to point the user towards this MVP via links or redirects. These changes would be made using a feature flag to enable controlled rollout of the MVP and allow the option of A/B testing.

It's quite possible that a non-engineer would be able to work with a coding agent to make these small changes to the main product, but a product engineer would still need to review them.

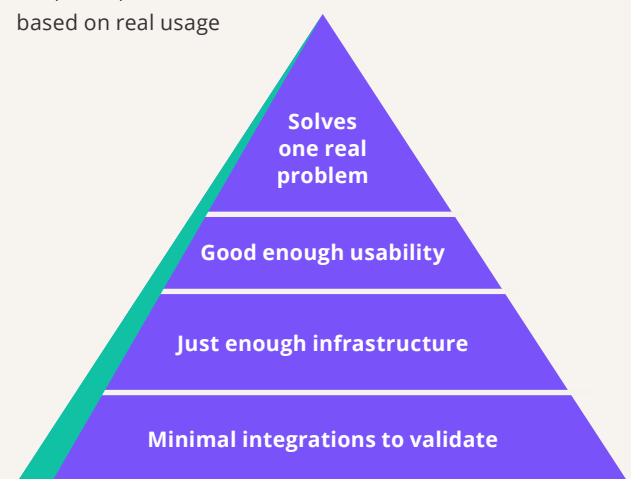
Wrong way to MVP

Overbuilding leads to wasted time, high cost and slow learning



Right way to MVP

Start small, validate fast, learn, and iterate based on real usage



Throw-away features



What it is

A product manager or designer wants to experiment directly in the product, trying out a new UX for an existing feature or adding new functionality embedded within the core product. Taking the same travel booking example from earlier, perhaps a product manager wants to include a weather forecast widget directly within the core booking flow.

There must be a clear understanding that these vibe-coded changes are throwaway experiments: blazing a trail and validating an idea, but with an expectation of being ripped out and replaced if the experiment bears fruit.

A non-engineer needs to be able to work with a coding agent in the core codebase, building and testing a change without help.

This requires an “agent-friendly” codebase, where an AI can easily:

- Learn the design and conventions of the codebase
- Run validation loops (unit tests, linters, and type-checkers)
- Run and test the app locally (e.g., via Playwright MCP)

It also requires a very mature CI/CD pipeline—the type of system where you’d be confident that a brand-new employee can propose a code change and get accurate feedback on its quality.



What it needs

Enabling this type of in-app experimentation requires systems and workflows that allow non-engineers to make code changes to production systems with minimal disruption to engineers, while still maintaining safety.

Finally, you’d need a mature progressive delivery practice, where a feature-flagged code change can deploy to a production environment and then get released in a controlled way, with guardrails that check for regression and roll back if appropriate.

Deliver value safely and quickly with confidence



Succeeding with this model

There are two key elements to succeeding with this dual-track model of experimentation:

- Firstly, there must be a clear distinction between throw-away prototypes and MVPs vs. production-grade software.
- Secondly, systems need robust infrastructure and strong guardrails in place to allow non-engineers to safely deploy prototypes and/or make changes to production codebases.

Cheap experiments vs. robust engineering

There must be a shared understanding of what software is being created as a disposable prototype.

You shouldn't be surprised that a non-technical stakeholder might see a fully functional prototype and naturally think "great, it's done, right!?" It's important to set expectations up front that these disposable prototypes are purely intended as a mechanism for rapid experimentation and are NOT the final product. You can remind stakeholders that "laying the rail"—re-implementing the functionality in a robust, well-tested way—will go a lot quicker now that the prototype has blazed the trail.

Likewise, since this is working software, sometimes even living within core codebases, there'll be a strong temptation to apply all the engineering rigour that's needed for long-lived code. Clearly flagging these trail-blazing changes as temporary experiments, with a commitment that they'll be removed or rebuilt, will help reviewers to understand where they can relax policy around maintainability and internal quality.

Guardrails for safe experimentations

Teams can only yield the full potential of this dual-track model once non-engineers are able to make changes to production codebases, which in turn requires a large platform investment in progressive delivery infrastructure.

The bar you should be aiming for is a system where a new engineer could confidently make a production change on their first day, protected by strong feature-flagging infrastructure. To be clear, this type of maturity is quite rare today, only found within high-performing engineering organizations, but it's certainly achievable with the appropriate investments.

The advent of AI-assisted engineering makes that investment in progressive delivery a lot more compelling: it now enables an entire new methodology of product experimentation, democratising access to the means of production and allowing teams to learn and experiment at a faster rate than ever before.

Wrapping up

It's clear that AI dramatically increases our capacity to produce code. It's also clear that this doesn't necessarily improve an organization's ability to deliver value. Bottlenecks that were once hidden by the expense of writing code are now exposed: review queues, testing delays, and release friction are becoming the new constraints. To reap the full benefits of agentic coding, organisations must now invest in systemically improving flow across the entire SDLC.

Feature flags, and the broader set of Progressive Delivery practices they enable, are a critical part of that system-level optimization. By decoupling deployment from release, shrinking batch sizes, and enabling continuous experimentation, they turn software delivery into a highly iterative, customer-centric process. In an AI-accelerated world, where changes are cheaper and more frequent, these practices are no longer aspirational, they're table-stakes expectations.

What's more, AI is not just making coders more productive, it's democratising who gets to write the code. With the help of coding agents, more people across the organization can contribute directly to building and shaping product experiences. This shift is disruptive, and these new cohorts of code contributors can only be brought on board safely if the underlying system is resilient. Feature flags are part of that safety net, allowing teams to rapidly prototype and iterate while containing risk.

Ultimately, succeeding with AI-assisted engineering isn't about choosing between speed and quality, or between vibe-coding and endless code review. It's about designing a resilient system that can accommodate a faster pace and a broader demographic of code contributors.

“

Progressive Delivery practices, along with an intentional approach to experimentation, provide that system. Together, they enable a mode of building software that is faster, safer, and more aligned with real user value.



Organizations that embrace this new form of product development will reap the full rewards of this new AI era.

Learn more at
flagsmith.com

Contact us at
sales@flagsmith.com