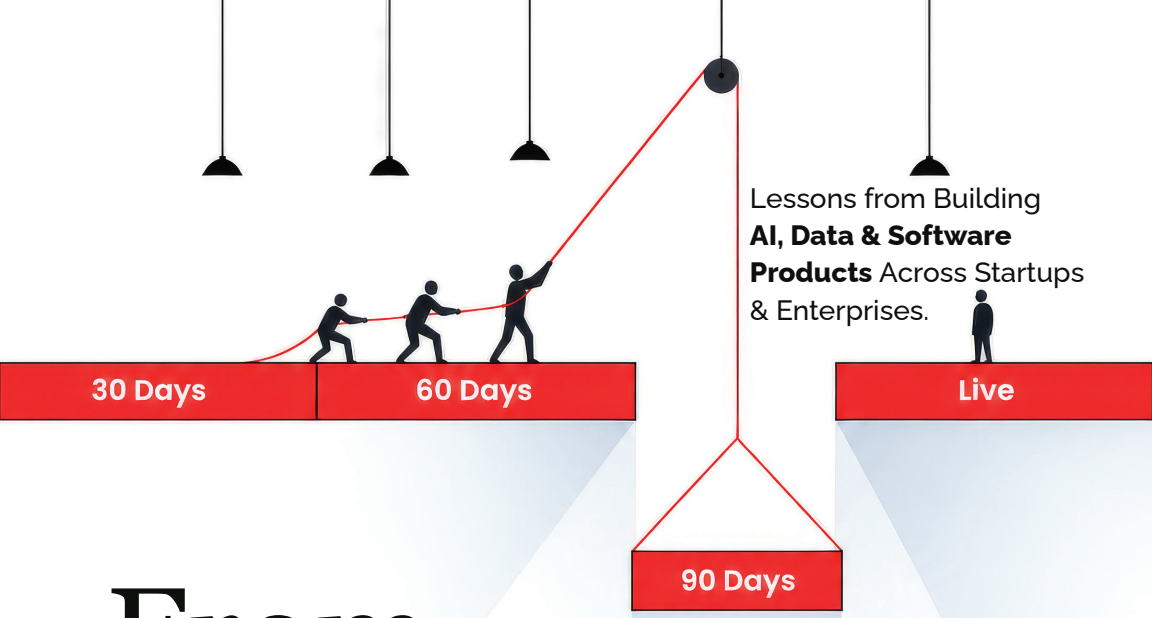


From Idea to Launch in 90 Days

A Practical Playbook for Building AI, Data & Digital Products That Deliver Real Business Value

Mahesh Lalwani
Antriksha Somani

Contents

Introduction 01

Part I

01

Discover & Architect

Days 1-30

CHAPTER 1

Start With the Problem, Not the Technology 03

1.1 Five Questions to Ask Before Choosing a Technology 7

CHAPTER 2

Why Most Software Projects Fail Before Development Begins 08

2.1 Define the Outcome Before You Define the Scope 9

2.2 Bridge the "Mismatched Mental Models" 10

2.3 Be Realistic About Your Data 10

2.4 Look Below the Surface - The Dependency Iceberg 12

2.5 Treat the MVP as an Experiment, Not a Smaller Final Product 13

2.6 Give the Project a Complexity Budget 13

2.7 One Product Needs One Owner 14

2.8 Five Questions to Ask Before Development Begins 16

CHAPTER 3

Choose the Right AI Architecture Without Overengineering It 17

3.1 Start With the Workload, Not the Model 18

3.2 Buy the Intelligence, Build It, or Blend the Two? 19

3.3 Don't Fall in Love With Your First API 20

3.4 Keep Probabilistic AI Away From Deterministic Business Rules 21

3.5 Retrieve Before You Train 24

3.6 Modernize at the Seams 26

3.7 Design for Failure 27

3.8 Make Cost, Performance and Quality Visible From Day 1 28

3.9 Five Questions to Ask Before Choosing Your AI Architecture 31

Part II

02

Targeted Execution & Build

Days 31-60

CHAPTER 4

Build Lean Teams That Move Fast 34

4.1 Build Around Outcomes, Not Job Titles 35

4.2 Give Engineers the Problem, Not Just the Ticket 36

4.3 The Architect Should Be a Player Coach 38

4.4 Shorten the Distance Between the Builder and the User 39

4.5 Every Important Decision Needs One Owner 41

4.6 Protect the Team From Organisational Gravity 41

4.7 AI Changes the Economics of a Lean Team 42

4.8 Measure Flow, Not Activity 43

4.9 Five Questions to Ask About Your Execution Team 46

Contents

CHAPTER 5
AI Doesn't Replace Great Engineering 47

5.1 Vibe Coding Is Great - Until It Isn't 48
5.2 Reuse Before You Rebuild 48
5.3 Architecture Matters More When Code Becomes Cheap 49
5.4 Build Security and IP Protection in From Day 1 50
5.5 AI Changes the Definition of "Done" 51
5.6 Don't Ask for Perfect - Define "Good Enough" 52
5.7 Know When to Keep a Human in the Loop 53
5.8 Five Questions to Ask Before Putting AI Into Production 56

Part III
03
Launch, Guard, & Scale
Days 61–90 & Beyond

CHAPTER 6
Launch Is When the Real Learning Begins 58

6.1 Build for Adoption Before You Launch 59
6.2 Instrument the Product Before You Need the Answers 60
6.3 Listen to Behavior But Don't Stop Talking to Users 61
6.4 Measure Value, Not Just Usage 62
6.5 Treat Corrections as Learning Data 63
6.6 Don't Let Feedback Become the Roadmap 64
6.7 Sometimes the Best Improvement Is Subtraction 65
6.8 Create Short, Disciplined Learning Loops 66
6.9 Five Questions to Ask After Launch 71

CHAPTER 7
Build AI That Makes Economic Sense 72

7.1 The Prototype Bill Isn't the Production Bill 73
7.2 Know What One Successful Outcome Actually Costs 74
7.3 Use the Least Expensive Intelligence That Reliably Does the Job 75
7.4 The Cheapest AI Call Is the One You Don't Make 76
7.5 Give Every Agent a Budget 77
7.6 Make the Gateway an Economic Control Point 78
7.7 Connect Cost to Business Value 79
7.8 Optimize After You Understand 80
7.9 Five Questions to Ask About Your AI Economics 82

CHAPTER 8
From Project to Product 83

8.1 Production AI Doesn't Stand Still 84
8.2 Make Every Change Safe to Reverse 84
8.3 A Product Needs an Owner After the Project Ends 85
8.4 Keep the Operating Scorecard Simple 86
8.5 Know the Difference Between Learning and an Incident 87
8.6 Five Questions to Ask on Day 91 89

Introduction

Towards the end in ***Harry Potter and the Chamber of Secrets,*** Harry is experiencing a profound identity crisis. He notices terrifying similarities between himself and ***Tom Riddle*** (Voldemort's teenage self) - both are half-bloods, orphans, can speak to snakes, and feel a sense of belonging at Hogwarts. And at that point, ***Albus Dumbledore*** tells him,

"It is our choices, Harry, that show what we truly are, far more than our abilities."

While their ***abilities*** are very similar, their ***choices*** set them apart. Harry is trying to make his world a better place, while Voldemort is out to ruin it for power and control.

I feel the same principle applies to technology and software engineering today. Many organizations have access to similar technologies, AI tools, cloud platforms and engineering talent. What increasingly separates them is not simply their capabilities, but the ***choices they make with those capabilities*** - what problems they choose to solve, what they build, what they decide not to build, how quickly they learn from customers, and how effectively they turn an idea into something that creates real value.

For years, the traditional software consulting playbook was built around large teams, long development cycles and hourly billing. I believe that model is becoming obsolete. AI has fundamentally changed the economics and speed of software engineering, but it has not replaced the need for good engineering. If anything, architecture, product judgment, data quality, security and experienced technical leadership matter even more when everything else can move faster.

At Ccube, we have chosen a different approach. We focus on outcomes rather than activity. Lean, engaged teams rather than large ones. And speed, culture and innovation rather than lengthy development cycles. Our 30-60-90-day approach is designed to help clients validate ideas quickly, build what matters and get solutions into the hands of real users creating a competitive advantage through execution.

Over the years, we have learned that complex problems do not always require complex solutions. They require clarity, good engineering choices, the right people and the discipline to keep moving forward.

That is what this book is about.

This isn't another software development book. It is a practical collection of methodologies, lessons and stories from the field - things we have seen work, things we have seen fail, and things we have learned while helping companies turn ideas into products across ***AI, Data and Applications.***

I hope some of these ideas resonate with you, and, more importantly, help you make better choices as you take your own ideas from concept to launch.

Part I

Discover & Architect

Days 1–30

CHAPTER 1

Start With the Problem, Not the Technology

There is an old saying: when all you have is a hammer, everything looks like a nail. The same can sometimes be true with technology.

When technologists learn a new language, tool, framework or AI model, there is a natural tendency to look for opportunities to use it. We get excited about what the technology can do and start framing problems around the tools we know. But the newest or most powerful technology may not be the right solution for the problem at hand.

At **Ccube**, we have tried to build a culture where we **don't talk about tools until we understand the problem.**

We start by understanding the cost of the problem from both the business and technical users' perspectives. What is it costing the business today - in lost revenue, wasted hours, customer friction, operational risk or missed opportunities?

Who is experiencing the problem? What has already been tried, and why didn't it work?

We engage with stakeholders to understand not only what they want changed, but also the impact that a new solution will have on their people, processes and customers.

I believe this exercise of validating the problem and demand before generating a single line of code is one of the most important parts of product engineering. It helps everyone participate, have a say, establish priorities and focus sprint cycles on solving the problems that will actually move the business forward.

And sometimes the fastest way to launch is to spend a little more time upfront making sure you are solving the right problem.

Build Problem Solvers, Not Technology Silos

Another important part of our approach is how we think about engineering teams.

Of course, we need people with expertise in AI/ML, front-end, back-end, data engineering and other disciplines. But we don't want people to define themselves only by those labels. **First and foremost, we want them to be creative problem solvers.**

As AI makes writing and generating code faster, I believe architecture, engineering fundamentals, systems thinking, APIs and microservices, curiosity and adaptability become even more important, not less.

Knowing how to use a technology is valuable. Knowing **when to use it, when not to use it, and what to use instead is engineering judgment.**

A Lesson I Learned Years Ago

Mahesh Lalwani
Founder & CEO, Ccube

A few years ago, at one of my former logistics accounts in Canada, we were called in as observers to listen to a technical proposal from an established Silicon Valley database company. They were favored to win a multi million dollar project.

Once the sales part of the presentation was over, I asked if I could speak with their technical lead. He joined us from Palo Alto over the conference room's speakerphone.

His answer was simple: "In cases where data comes in bursts."

That happened to be exactly how the logistics company's data behaved.

That one answer changed the direction of the evaluation. Their technology may have been excellent, but it wasn't the right technology for this particular problem. Our team was ultimately selected to design and implement the solution.

I have remembered that conversation ever since.

The lesson wasn't that one database was better than another. It was that ***technology should be selected for the problem, not the problem shaped to fit the technology.***

1.1 Five Questions to Ask About Your Execution Team

Before selecting a platform, architecture, AI model or development tool, I recommend asking five simple questions:

01 What problem are we really trying to solve?

02 What is this problem costing us today?

03 Who experiences the problem, and what would success look like to them?

04 What have we tried before, and what did we learn from it?

05 Under what circumstances would the technology we are considering not be the right choice?

I particularly like the last question.

> ***Whether you are talking to your own technical team, a software vendor or a consulting company, ask them when they would not recommend their proposed solution.***

You may learn more from that answer than from the entire sales presentation.

CHAPTER 2

Why Most Software Projects Fail Before Development Begins

We often look for the reasons a software project failed after development has started.

Was the architecture wrong? Did we choose the wrong technology? Was the team not strong enough? Did requirements keep changing? Did development take too long?

In my experience, many projects are already in trouble **before the first line of code is written.**

That is why, instead of waiting for a post-mortem, I recommend starting with a pre-mortem. Before development begins, imagine that the project has already failed. Then ask the team a simple question:

"What went wrong?"

The answers may reveal unclear priorities, hidden dependencies, data issues, unrealistic expectations, missing ownership or risks nobody has been comfortable raising.

Reviewing that list - and addressing the most important items before you start - can significantly increase your chances of success.

The problem is usually not a lack of talent or technology. It is that people have different definitions of success, the data isn't ready, dependencies haven't been identified, priorities aren't clear, or nobody has the authority to make the difficult tradeoffs.

A good engineering team can solve a lot of technical problems. What it cannot easily overcome is organizational ambiguity.

That is why the first few weeks of a project are so important.

2.1 Define the Outcome Before You Define the Scope

One of the first questions we like to ask is surprisingly simple:

> *What would make you call this project a success 90 days from now?*

The answers can be revealing.

One executive may want to demonstrate an AI capability to customers. Operations may want to eliminate a manual process. The product team may want a production-ready feature.

Engineering may think the objective is to modernize an aging architecture.

Everyone can be enthusiastic about the same project while actually supporting four different projects.

Before creating a backlog, I prefer to establish two or three measurable outcomes. Maybe we want to reduce customer onboarding from five days to one. Automate 70% of a manual process. Prove that an AI model can achieve an acceptable level of accuracy. Or put a working product in the hands of 50 customers.

Scope should follow the desired outcome - not the other way around.

Once everyone agrees on what success looks like, deciding what belongs in the first 90 days becomes much easier.

2.2 Bridge the "Mismatched Mental Models"

Business executives, domain experts, product owners and software architects naturally see the same problem differently.

An executive might say,

"We need to streamline customer onboarding."

A domain expert thinks about forms, approvals and exceptions.

A data engineer thinks about customer records coming from multiple systems.

A software architect may immediately see APIs, identity management and database changes.

None of them is necessarily wrong.

The problem begins when everyone walks out of the meeting believing they agreed.

We try to eliminate this false consensus early by establishing a shared domain language. What exactly do we mean by a "customer," an "order," an "approval" or a "qualified lead"? Where does each process begin and end? Who owns the information? What business event triggers the next step?

Software architecture eventually has to represent the real world of the business. The sooner the business and technical teams develop the same mental model of that world, the less rework we create later.

2.3 Be Realistic About Your Data

This is particularly important with AI projects.

We frequently hear some variation of:

"We want to build an AI solution, but our enterprise data is a mess."

That may be true. But I don't believe every AI initiative should wait for a company-wide data cleanup or a perfect data warehouse.

For an MVP, the better question is:

A Successful Product
Identity and Access Management.
Security reviews
APIs controlled
Legacy Systems
Data Permissions
Infrastructure
Compliance
Procurement
Third-party vendors
An executive approval

> *What is the minimum viable data we need to determine whether this idea works?*

Sometimes that means extracting a targeted subset of information from existing systems. Sometimes it means building a pragmatic pipeline rather than an enterprise-wide data platform. And in appropriate use cases, synthetic data can help us develop and test portions of the solution while access to production data is being resolved.

This doesn't mean ignoring data quality. Quite the opposite.

It means isolating the data problem that actually matters to the MVP rather than making the MVP responsible for fixing every data problem the organization has accumulated over the past ten years.

Perfect data may take years.

Learning whether your idea has value shouldn't.

2.4 Look Below the Surface The Dependency Iceberg

The feature everyone wants to build is usually the visible part of the project.

Below the surface are the dependencies.

Identity and access management. Security reviews. APIs controlled by another team. Legacy systems. Data permissions. Infrastructure. Compliance. Procurement. Third-party vendors. An executive approval nobody realized was required.

Any one of these can turn a 90-day project into a nine-month project.

That is why we try to identify dependencies during the first weeks rather than discovering them halfway through development.

I like separating dependencies into three categories:

What we control.

What we can influence.

What we cannot control.

The first category belongs to our plan. The second needs an owner and active management. The third needs either a contingency or an alternative path.

A dependency you know about is something you can manage.

A dependency you discover in Week 10 can become your launch date.

2.5 Treat the MVP as an Experiment, Not a Smaller Final Product

The term MVP is probably one of the most overused and misunderstood terms in software development

An MVP isn't simply the final product with fewer features.

Its job is to answer important questions.

Will customers use this? Can we technically make it work? Is our data sufficient? Does it improve the process? Will someone pay for it?

If an MVP isn't helping us learn something important, we should question why we are building it.

This distinction also helps prevent scope creep. Instead of asking, "Would this feature be useful someday?" we can ask:

> *Do we need this feature to validate what we're trying to learn in the next 90 days?*

Those are very different questions.

2.6 Give the Project a Complexity Budget

Every feature has a cost beyond the hours required to build it.

It introduces another decision for the user, another interaction to design, another component to test, another potential failure mode, and often another concept the development team must carry in its head.

Complexity accumulates quickly.

For an MVP, we therefore try to maintain what I think of as a **cognitive or complexity budget.** The initial product should make a small number of important things exceptionally clear and easy.

This requires discipline because stakeholders will always have good ideas.

Rather than rejecting those ideas, create a **"Not Now" list.**

A new request can go there, or it can replace something already committed to the MVP. What it shouldn't do is quietly expand the scope every week.

This changes the conversation from:

"Is this a good idea?"

to:

"Is this more important than something we have already committed to?"

Almost everything sounds important when considered independently. Prioritization only becomes real when there is a tradeoff.

2.7 One Product Needs One Owner

There is another dependency that rarely appears on an architecture diagram:

Decision-Making.

I have seen talented engineering teams lose days waiting for a decision that should have taken 20 minutes.

Projects don't always move slowly because development is slow. Sometimes **decisions are slow.**

When five stakeholders have equal authority and different priorities, even a five-person engineering team can become paralyzed.

Every project needs an empowered product owner, someone who understands the desired business outcome, listens to the stakeholders, works closely with engineering and, most importantly, has the authority to make tradeoffs.

That person needs to be comfortable saying three things:

Yes. No. And not now.

The last one may be the most important.

Speed isn't simply how quickly your engineers can write code. Speed is how quickly the entire organization can make good decisions and remove obstacles for them.

Start With Alignment, Then Start Building

None of this means spending months creating specifications, architecture diagrams and project plans before development begins

That would defeat the purpose.

The goal is exactly the opposite:

establish enough clarity to start learning quickly.

Agree on the outcome. Establish a common language. Understand the minimum viable data requirement. Expose the critical dependencies.

Define what the MVP needs to prove. Protect its complexity budget. Empower someone to make decisions.

Then start building.

You won't know everything on Day 1 - and you shouldn't try to.

But you should know enough to make sure everyone is running in the same direction.

2.8 Five Questions to Ask Before Development Begins

Before approving the first sprint of an important software or AI initiative, I recommend asking these five questions:

01 What measurable outcome would make us call this project successful 90 days from now?

02 Do the business, product and engineering teams actually mean the same thing when they describe the problem and the solution?

03 What are the three biggest dependencies - data, systems, people or approvals, that could prevent us from launching?

04 What can we deliberately leave out of the MVP and still validate our most important assumptions?

05 Who has the authority to make a difficult scope or priority decision within 24 hours?

If your team cannot answer these questions clearly, I would be cautious about starting development.

> ***Writing code creates the appearance of progress. Alignment creates the conditions for progress.***

CHAPTER 3

Choose the Right AI Architecture Without Overengineering It

AI technology is moving incredibly fast.

New models appear almost every week. Yesterday's most capable model may be overtaken tomorrow. New agent frameworks, vector databases, development tools and AI platforms are constantly entering the market.

This creates an understandable challenge for anyone starting an AI initiative:

> ***How do we choose the right technology when the technology keeps changing?***

I don't think the answer is to predict which model, framework or platform will win.

In fact, trying to design the perfect architecture for where AI might be three years from now can create exactly the kind of complexity that prevents you from launching anything in the next 90 days.

I prefer a different approach.

Make the best decisions you can with what you know today, but design the architecture so those decisions can change tomorrow.

You don't need to predict what AI will look like three years from now.

You need an architecture that lets you change your mind when it does.

> *You don't need to predict what AI will look like three years from now. You need an architecture that lets you change your mind when it does.*

3.1 Start With the Workload, Not the Model

One of the easiest mistakes in an AI project is starting with the model.

> *Should we use GPT? Claude? Gemini? Llama? A smaller model? Should we fine-tune something ourselves?*

Those are legitimate questions.
But they aren't the first questions.

I would start with:

> ***"What exactly are we asking the AI to do?"***

Classifying a support email is different from analyzing a complex legal document. Extracting fields from an invoice is different from conducting research across hundreds of documents. Summarizing a meeting is different from reasoning through a multi-step business problem.

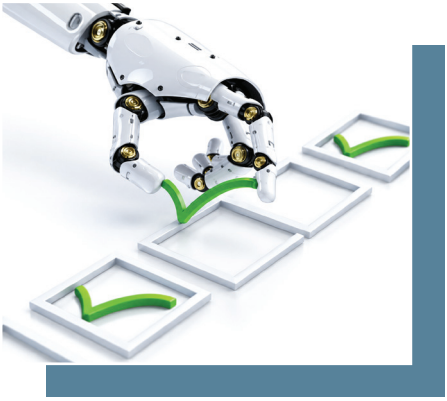
They don't necessarily require the same model.

This matters because the most capable model is often also the most expensive or slower model and many business tasks simply don't need it.

I think of this much like hiring people.

You wouldn't ask your most senior architect to perform every task simply because they are the most capable person on the team.

AI should work the same way.



> *Use the right intelligence for the right job.*

That decision alone can have an enormous impact on the economics and performance of an AI application.

3.2 Buy the Intelligence, Build It, or Blend the Two?

Once we understand the workload, we can make a more informed decision about where the intelligence should come from.

Third-party AI APIs offer tremendous advantages.

They allow us to start quickly, access highly capable models without building infrastructure and benefit from billions of dollars of research and investment being made by the major AI companies.

For many MVPs, that can be the right place to start.

But there are tradeoffs.

As usage grows, API costs can become significant. Certain workloads may have latency requirements. Some organizations have strict privacy, security or data-residency requirements. And relying heavily on one provider can create vendor dependence.

Smaller open-weight models offer a different set of tradeoffs.

They can provide greater control, potentially lower economics at sufficient scale, deployment within private environments and the ability to optimize models for specific workloads.

But that control isn't free.

Someone must deploy, operate, monitor and maintain the infrastructure. The organization also becomes responsible for more of the model lifecycle.

So, I don't think the decision should be ideological.

> ***"We only use commercial APIs"*** *and* ***"We should own all our models"*** *can both be bad architectural principles.*

> The right question is:
>
> *What combination gives us the best balance of quality, speed, cost, security and control for this business problem?*
>
> And increasingly, the answer may be both.

3.3 Don't Fall in Love With Your First API

There is something I think of as the **API Honeymoon.**

At the beginning of an AI project, everything looks wonderful.

Connect to an API. Send a few hundred requests. Get impressive responses. The cost is negligible. The development team moves quickly.

Then the application succeeds.

A few hundred calls become hundreds of thousands. Different use cases emerge. Some requests need sophisticated reasoning while others are simple classifications. Latency begins to matter. API costs start appearing on someone's financial report.

Success changes the economics.

That is why I like introducing a lightweight **model gateway or AI router** early in the architecture.

Instead of allowing application logic to become tightly coupled to one AI provider, the application communicates through an abstraction layer.

That layer can decide which model should handle which workload.

A complex reasoning request might go to a highly capable frontier model.

A simple classification task might go to a smaller, less expensive model.

A privacy-sensitive workload might be routed to a model running within a controlled environment.

And over time, workloads may move between providers or to smaller, optimized models without requiring major changes throughout the application.

This isn't about creating an elaborate AI platform on Day 1.

It is about preserving **optionality.**

The goal isn't to make switching models effortless. Models have different capabilities and behaviors.

The goal is to avoid making your entire application dependent on a decision you made during Week 1.

> ***The best AI architecture doesn't necessarily choose the perfect model. It preserves your ability to choose again.***

3.4 Keep Probabilistic AI Away From Deterministic Business Rules

There is another architectural boundary I think is extremely important.

AI models are probabilistic. Enterprise business systems frequently are not.

Suppose an AI application receives an invoice.

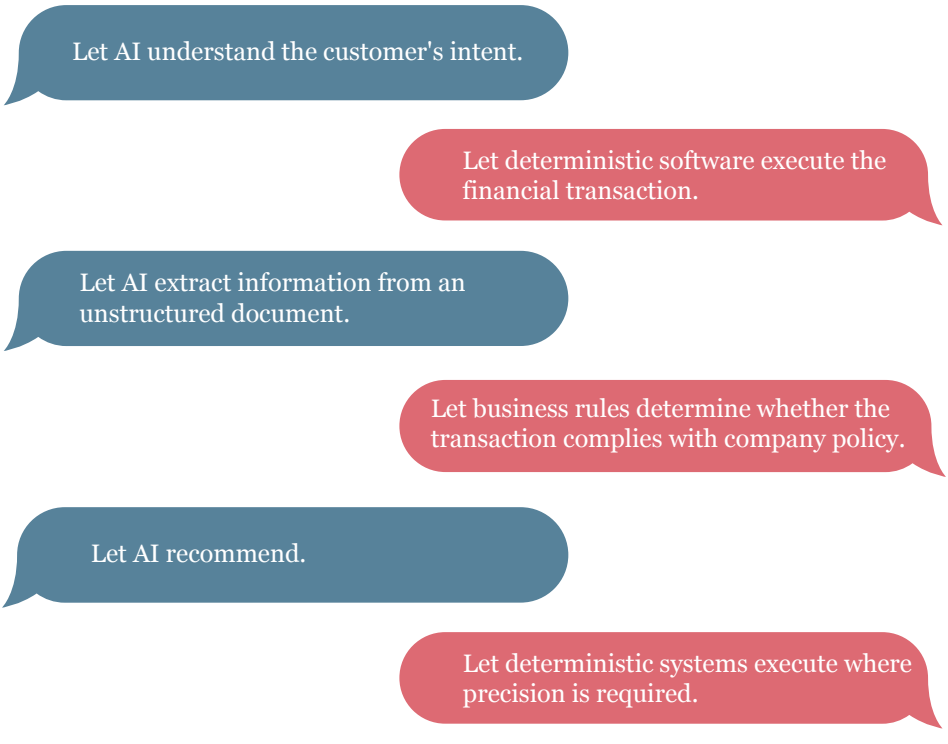
AI may be excellent at understanding the document, identifying the vendor, extracting line items and determining what type of expense it represents.

But should we ask the language model to calculate the final accounting entry according to a precise set of financial rules?

Probably not.

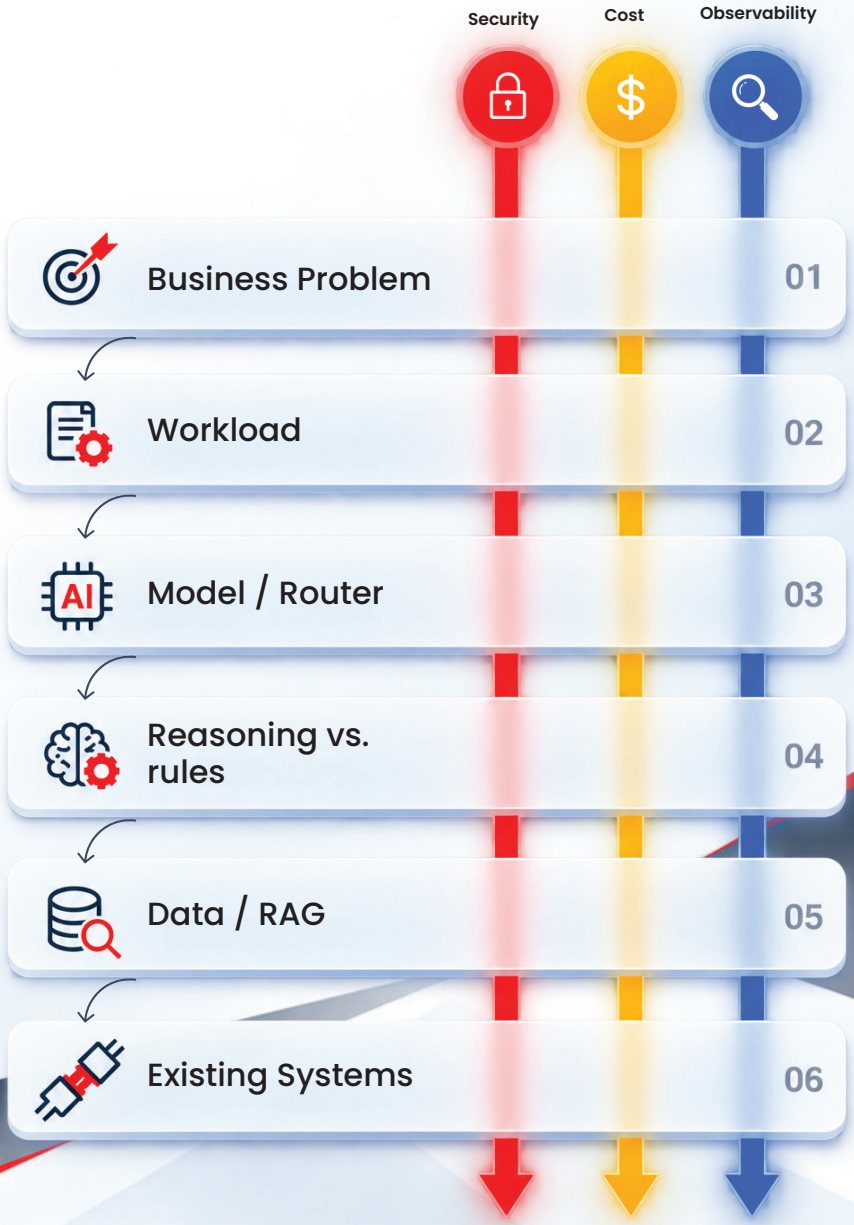
We already know how to build software that performs exact calculations reliably.

The same principle applies to many applications.



Chapter 3

The AI Architecture Decision Stack



I think of this as establishing a clear architectural boundary between the Probabilistic and Deterministic layers. Enterprise software becomes fragile when non-deterministic LLMs are tasked with deterministic business rules. The architecture must explicitly separate reasoning from execution:

System Layer	*Operational Responsibility*	*Use Cases*
Probabilistic Reasoning Layer	Handling unstructured inputs, context, and ambiguous data.	Document intent parsing, line-item extraction, query summarization.
Deterministic Execution Layer	Executing exact business logic, calculations, & rules.	Accounting tax calculations, policy compliance checks, database writes.

The framework, whether Mastra, LangChain, Google's ADK or something that appears six months from now, is less important than the architectural principle.

> *Choose architecture principles first. Frameworks second.*

Enterprise software becomes fragile when we ask probabilistic technology to perform jobs where the business expects deterministic outcomes.

Use AI where ambiguity requires intelligence.

Use software where certainty requires rules.

3.5 Retrieve Before You Train

Another common question we hear is:

> *Should we fine-tune a model on our company's data?*

Sometimes the answer is yes.

But often the underlying requirement is simply:

> *We want the AI to know our information.*

Those are not necessarily the same problem.

If the AI needs access to product documentation, company policies, customer information, technical manuals or other information that changes regularly, we may be better served by retrieving the relevant information at the time of the request and providing it to the model.

This is the basic idea behind **Retrieval-Augmented Generation, or RAG.**

Fine-tuning serves a different purpose.

It can be useful when we want a model to behave differently, follow specialized patterns or perform a narrowly defined task more effectively.

I like a simple mental model:

> *Use retrieval to give AI knowledge. Use fine-tuning when you need to change behavior.*

There are exceptions, of course. AI architecture is rarely that absolute.

But the distinction can prevent a team from investing weeks training and maintaining something that could have been solved more simply.

For a 90-day initiative, simplicity matters.

> *Don't train what you can **retrieve.***
> *Don't build what you can **configure.***
> *And don't create complexity until the **problem requires it.***

3.6 Modernize at the Seams

Not every AI initiative has the luxury of starting with a modern technology stack.

In fact, some of the most valuable AI opportunities exist inside organizations with decades of legacy technology.

That creates another dangerous assumption:

> *Before we build the new AI application, we need to modernize the old platform.*

Perhaps eventually.

But turning a 90-day AI initiative into a two-year modernization program is usually a good way to make sure the AI initiative never launches.

Instead, I prefer incremental modernization.

There is a well-established software architecture approach called the Strangler Fig Pattern. Rather than replacing a large monolithic system all at once, new capabilities are gradually built around it.

Over time, pieces of the old system can be replaced as the business case justifies it.

The same principle works well for AI.

Expose the information the new application needs through APIs. Introduce a new service where necessary. Build the AI capability around the existing core rather than immediately replacing it.

I sometimes think of this as modernizing at the seams.

Find the places where the old and new systems can safely meet.

Build there first.

Then let actual business value determine what gets modernized next.

A fragile legacy system shouldn't dictate your future architecture.

But neither should an exciting AI project force you to replace systems that are still doing their jobs.

3.7 Design for Failure

Architecture diagrams usually describe what happens when everything works.

Production systems need to answer another question:

What happens when it doesn't?
An AI provider will eventually time out.
An API may reach its rate limit.
Retrieval may return the wrong document or nothing at all.
A model may return an answer in the wrong format.
An agent may be unable to complete a task.
A dependency will eventually fail.
That shouldn't automatically mean the application fails.

Perhaps the request can be routed to another model. Perhaps we retry. Perhaps the application falls back to a deterministic workflow. Perhaps we tell the user that we aren't confident enough to answer. Perhaps the request goes to a human.

The correct response depends on the use case.

But the response shouldn't be invented during the incident.

> *Failure is not an edge case. Failure is part of the architecture.*

For a production AI application, I want to know both the happy path and the fallback path.

3.8 Make Cost, Performance & Quality Visible From Day 1

Finally, if we want the architecture to evolve intelligently, we need to know what is happening inside it.

Traditional application monitoring tells us whether systems are available, how long requests take and where errors occur.

AI applications require another layer of visibility.

- Which models are being used?
- How much is each interaction costing?
- How long does each model take to respond?
- How often does retrieval return useful information?
- Where are requests failing?
- Which workloads are consuming most of the tokens or compute?
- Which model is performing best for a particular task?

Without this information, architecture decisions become opinions.

With it, they become engineering decisions.

For example, we may discover that 70% of requests don't require our most expensive model. Or that a smaller model performs just as well for a particular classification task. Or that most of our latency isn't coming from the model at all, it's coming from data retrieval.

These are valuable discoveries.

But we can only make them if the architecture gives us visibility.

> *You can't optimize what you can't see.*

This also gives the team something even more important: the evidence required to change the architecture as the product grows.

Security Is a Design Constraint, Not a Separate Architecture

There is one consideration that cuts across every decision in this chapter: security and privacy.

What data can leave our environment? Where is it processed? Is it retained? Which models can access sensitive information? What permissions do agents have? Where are credentials stored? Do regulatory or data-residency requirements constrain our choices?

I don't believe security should be something we bolt onto the architecture after everything else has been decided.

It should influence model selection, data architecture, routing, retrieval integrations and deployment from the beginning.

The same is true of cost and performance.

These aren't problems to solve after the product becomes successful.

> *They are architectural constraints that should be understood early without allowing them to paralyze the MVP.*

That balance matters.

We want enough architecture to protect our future options.

We don't want so much architecture that we never reach the future.

> *Build for Today*
> *Preserve Tomorrow*

There is a natural tendency among good engineers to think ahead.

What happens when we have a million users? What if we switch models? What if we expand internationally? What if this service needs to process ten times the volume? What if we eventually replace the legacy platform?

These are good questions.

But they don't all need to be solved on Day 1.

For a 90-day initiative, I prefer distinguishing between two types of architectural decisions:

> ***Decisions*** *that are* ***expensive*** *to reverse* ***later.***
> ***and***
> ***Decisions*** *we can* ***safely defer*** *until we* ***know more.***

Spend your architectural energy on the first group.

Create clean boundaries. Protect the data. Separate probabilistic reasoning from deterministic execution. Avoid unnecessary provider lock-in. Make the system observable. Design sensible failure paths.

Then start building.

The objective isn't to create an architecture that anticipates every possible future.

It is to create one that **doesn't unnecessarily prevent the future from changing.**

3.9 Five Questions to Ask Before Choosing Your AI Architecture

Before committing an important AI initiative to a particular model, platform or architecture, I recommend asking these five questions:

01 What does this workload require in terms of intelligence, quality, latency, security and cost, and are we using more AI than the problem needs?

02 If the model, provider or economics change 12 months from now, how difficult would it be for us to change our choice?

03 Have we clearly separated the work that requires probabilistic AI reasoning from the business rules and transactions that require deterministic execution?

04 Are we solving a knowledge problem that requires retrieval, a behavior problem that may justify fine-tuning, or are we adding complexity simply because the technology makes it possible?

05 What happens when the AI, API, retrieval system or legacy dependency fails, and will we be able to see enough in production to understand why?

There will always be another model. There will always be another framework. And there will always be another technology promising to make today's architecture obsolete. I don't think we should fear that. We should design for it.

> ***The best architecture** isn't the one that perfectly predicts the future. It's the one that gives you the freedom to adapt when the future arrives.*

Building an MVP?

Email us at

sales@ccube.com

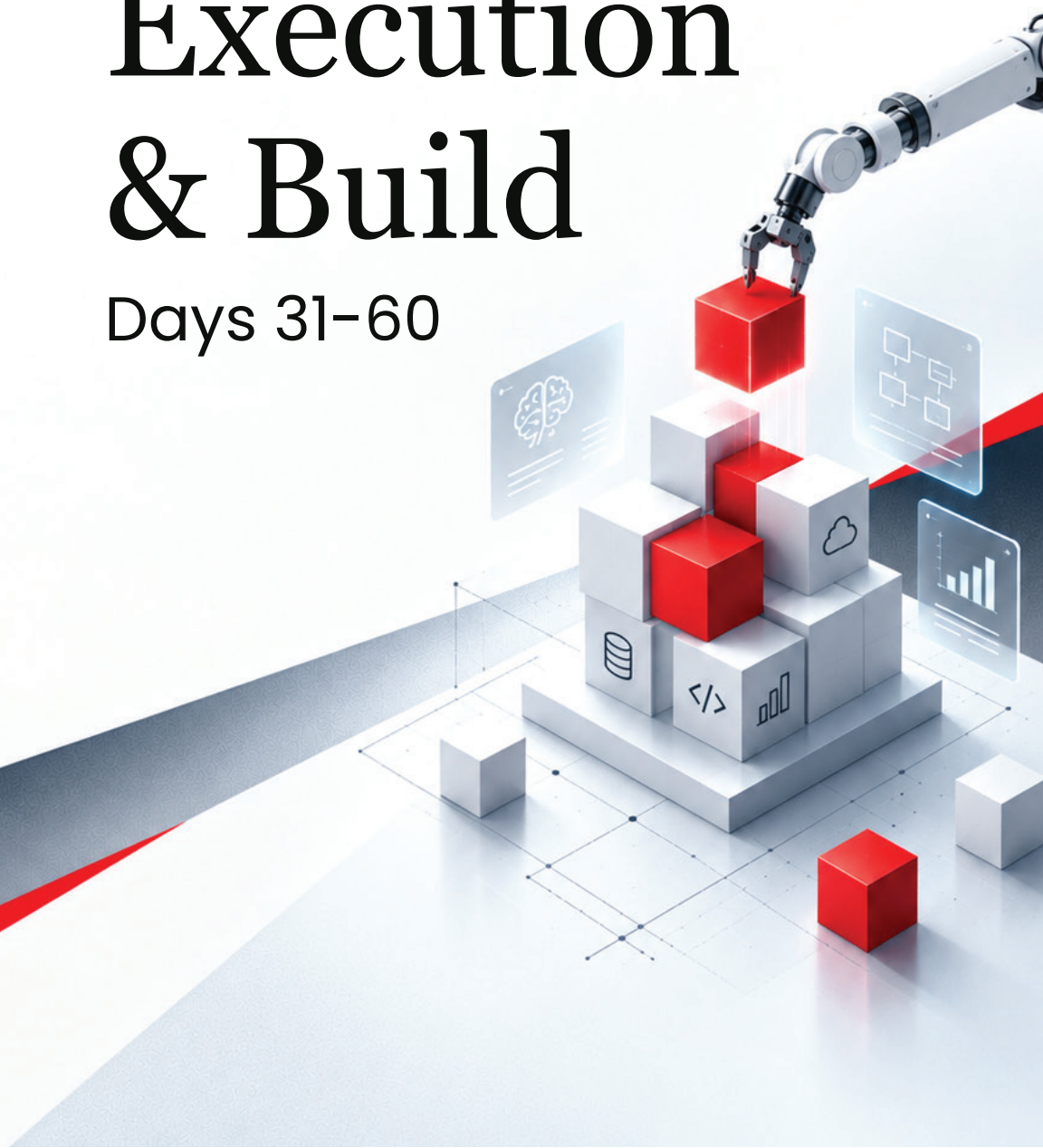
and we will send you a free 90-Day Launch Readiness Scorecard to see where your plan may be at risk.

Or scan this ***QR code:***

Part II

Targeted Execution & Build

Days 31-60



CHAPTER 4

Build Lean Teams That Move Fast

There is a well-known concept in the technology world that a team should be small enough to be fed with two pizzas.

A small team doesn't automatically move fast. A five-person team that depends on six other teams, attends meetings all day, waits three days for every decision and hands work from one specialist to another can be remarkably slow.

The real advantage of a small team is not its size. It is its **ability to operate with autonomy, ownership and very few handoffs.**

When we put together teams for fast-moving product initiatives, we try to give a small group of people enough skills, context and authority to take something from an idea through architecture, development, testing and ultimately into the hands of users.

> *Optimize for fewer dependencies, not simply fewer people.*

4.1 Build Around Outcomes, Not Job Titles

Software teams have traditionally been organized around specialization.

We need front-end developers. Back-end developers. Data engineers. AI/ML engineers. QA. DevOps. Architects. Business analysts. Project managers.

Of course, these skills are still important. But if we aren't careful, every specialization creates another handoff.

For an MVP or a 90-day initiative, I prefer building a **small product engineering POD** around the outcome we are trying to achieve.

Depending on the problem, that may be four, five or seven people with complementary skills. Some will naturally be stronger in AI, data, user experience, architecture or testing. But collectively, **the team owns the product and the outcome.**

That creates an important change in mindset.

> *Instead of :*
> ***"My code is done."***
> *We want to hear :*
> ***"Our product works."***

> *That small distinction says a lot about the culture of an engineering team.*

4.2 Give Engineers the Problem, Not Just the Ticket

One of the easiest ways to waste good engineering talent is to turn engineers into ticket takers.

"Build this API."

"Change this workflow."

"Add this field."

"Create this screen."

The engineer may execute every ticket perfectly without ever understanding why the customer needs it.

I would rather give engineers context.

What problem is the customer experiencing? Why does it matter? What happens today? What are we trying to change? What would make the user say, "This is much better"?

When engineers understand the business problem, something interesting happens.

They start challenging requirements.

> *"Do we really need another screen?"*

> *"We already have this data - why are we asking the customer for it again?"*

> *"Instead of building this workflow, could we automate the decision?"*

Those conversations are valuable.

A strong engineer who understands the problem can make hundreds of small decisions without waiting for someone else to tell them exactly what to do.

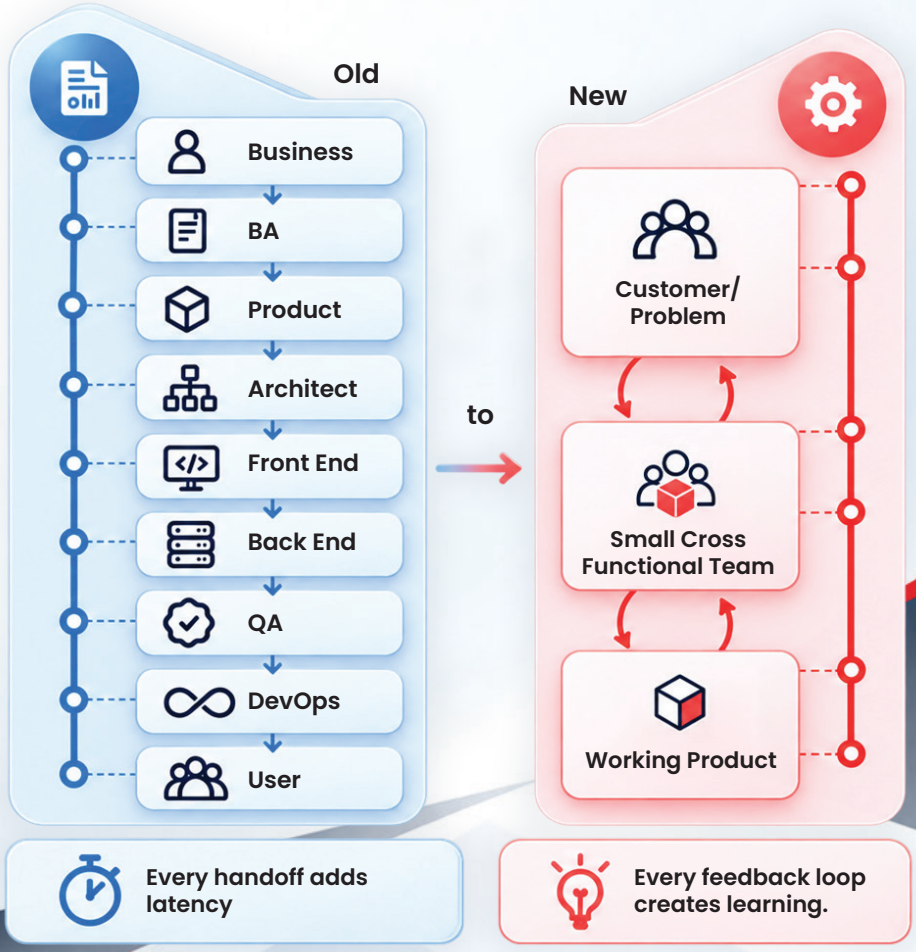
> ***Don't turn engineers into ticket takers. Turn them into product thinkers.***

Chapter 4

Handoffs vs. Flow

Traditional Team

Lean Product POD



4.3 The Architect Should Be a Player Coach

I also believe strongly that architecture should stay close to execution.

In some traditional development models, an architect spends the first few weeks creating diagrams, standards and specifications and then moves on. The development team is left to implement the architecture.

That model doesn't work particularly well when you're trying to move fast.

For a small product team, I prefer the architect to operate as a **player coach.**

They should establish the important architectural principles but remain engaged with the team, reviewing difficult implementations, helping resolve technical uncertainty, mentoring engineers and getting involved when the team encounters an important design decision.

They should also be willing to say:

> ***"We don't need that yet."***

That may be one of the architect's most valuable contributions to an MVP.

There is always a temptation to build for every future possibility - more scalability, more abstraction, more services, more flexibility. Some of it may eventually be necessary.

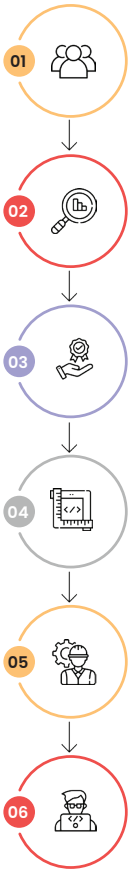
But every architectural decision carries cost and complexity today.

I believe in **just enough architecture:** enough to avoid expensive mistakes and provide a solid foundation, without trying to perfectly design a product that hasn't yet encountered real users.

> *Architecture should create velocity, not become a gate to velocity.*

4.4 Shorten the Distance Between the Builder and the User

Think about how requirements sometimes travel through a large organization:



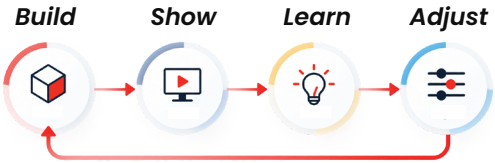
Every person in that chain is trying to help.

But every handoff also creates an opportunity for context to disappear.

By the time the requirement reaches the developer, the original customer problem may have been translated several times.

For a fast-moving team, I prefer engineers to participate in discovery, demonstrations and customer feedback whenever possible.

The loop should be simple:



And that loop needs to be short.

Don't wait six weeks to show the customer what you've built. Show something in days. It doesn't need to be perfect. It needs to be good enough to generate useful feedback.

This is an important distinction when we talk about development speed.

> *Speed isn't how* ***quickly*** *you write code.*
> *It is how* ***quickly*** *you discover whether you are building the* ***right thing.***

A team that writes software for four weeks and then discovers it misunderstood the requirement wasn't moving fast, no matter how productive those four weeks appeared.

4.5 Every Important Decision Needs One Owner

Small teams work exceptionally well when ownership is clear.

They work terribly when it isn't.

There is a difference between **ownership and participation.** Many people can participate in a decision. Many people should provide expertise and perspective.

But ultimately, someone needs to own it.

This applies to product priorities, architecture, data, user experience and the overall delivery.

One of the biggest sources of delay in software development isn't development at all.

It is **decision latency.**

An engineering team can lose two days waiting for a 20-minute decision. Repeat that ten times during a project and suddenly several weeks have disappeared from the schedule.

I like a simple rule:

> *Every important decision needs one owner. Not three.*

Ownership doesn't mean ignoring other people's opinions. It means someone is accountable for listening, deciding and moving the team forward.

4.6 Protect the Team From Organizational Gravity

Even a great team can become slow when the organization around it makes moving fast difficult.

Meetings. Status reports. Multiple approval chains. Competing priorities. Support requests. Shared resources. Committees. Constant interruptions.

I sometimes think of these forces as organizational gravity.

The larger the organization, the stronger that gravity can become.

If you give a team an aggressive 90-day objective, leadership has an important responsibility: protect the team's ability to execute.

Give them clear priorities. Remove unnecessary meetings. Resolve escalations quickly. Secure the resources they need. Keep unrelated work away from them. Don't change the objective every two weeks.

In effect, create an execution bubble around the team.

We often tell engineering teams they need to move faster.

Sometimes the better question for leadership is:

> *What are we doing that is preventing them from moving faster?*

4.7 AI Changes the Economics of a Lean Team

AI has made this operating model even more interesting.

Today, a good engineer can use AI to accelerate prototyping, code generation, unit testing, documentation, debugging, refactoring, code reviews and even understanding unfamiliar code.

That doesn't eliminate the need for engineers.

It increases the **leverage of good engineers.**

Work that previously required several specialists or considerable time can increasingly be accomplished by a smaller number of capable people using AI effectively.

But there is another side to this.

AI can generate code quickly. It can also generate the **wrong code quickly.**

It can help you implement a good architecture faster, and help you implement a bad architecture faster.

So as AI increases execution speed, engineering judgment, fundamentals, architecture and technical leadership become more important, not less.

> ***AI amplifies engineering capability. It also amplifies engineering decisions.***

That is another reason I would rather have a smaller team of strong, curious problem solvers with good technical leadership than simply add more people to a project.

4.8 Measure Flow, Not Activity

Software organizations have developed many ways to measure productivity.

Hours worked. Tickets completed. Story points. Lines of code. Sprint velocity.

Some of these metrics can be useful internally, but none of them is the reason we started the project.

For a 90-day initiative, I am much more interested in the flow:

> *How long does a decision sit unresolved?*

> *How quickly can an idea become something the user can experience?*

> *How quickly can feedback become an improvement?*

> *How often are we putting working software in front of users?*

These tell me more about the health of an execution team than the number of tickets it closed last week.

> ***Ultimately, working software and learning matter more than activity.***

Lean Teams Need Big Trust

There is a common thread running through all of this.

Small teams move fast when we trust them.

We trust engineers with the business context rather than giving them only requirements. We trust architects to make pragmatic tradeoffs. We trust product owners to make decisions. We let people talk directly to users. And we give the team enough autonomy to solve problems without asking permission at every step.

In return, the team needs to accept something equally important:

Ownership.

Autonomy without accountability creates chaos. Accountability without autonomy creates bureaucracy.

High-performing teams need both.

When you get that balance right, four or five highly engaged people can sometimes accomplish what organizations assume requires fifteen.

Not because they type faster.

Because they spend more of their time **building, learning and deciding, and less of their time waiting, handing off and coordinating.**

A Lesson in What Lean Teams Can Accomplish

Sonam Bansal
Vice President, Ccube

Some projects teach you more in 90 days than others do in years.

A healthcare startup came to us with a clear but ambitious goal: build a mobile application for both iOS and Android from scratch and launch it within 90 days.

There was no existing codebase to build on. Just an idea, a deadline and a lot of work ahead.

We deliberately kept the team small: one backend and AI engineer, one frontend developer, one QA engineer and one designer.

Four people. Clear ownership. Very few handoffs.

For the next 90 days, they designed, built, tested and iterated together. The designer worked closely with the developers, and QA was embedded from Day 1 not brought in at the end to find everything that had gone wrong.

Then, as we approached launch, we hit an unexpected obstacle.

Our Google Play submission required an additional 14-day closed testing period with real users before the app could be approved for production.

We had a choice: treat it as a delay or figure out how to absorb it into the plan.

We quickly mobilized additional testers, organized a structured testing cycle and used those 14 days to gather more feedback and improve the product. Instead of simply waiting for the requirement to pass, the team turned it into another development cycle.

Both the iOS and Android applications launched successfully.

What stayed with me from this project wasn't just the speed. It was how the team responded when the plan changed.

There was no large organization to coordinate and no series of handoffs to renegotiate. The same people who understood the product also understood the problem and could immediately adjust.

And because quality had been part of the process from the beginning, an unexpected testing requirement became useful rather than disruptive.
The lesson for us was simple:

> *Build lean teams with clear ownership and quality from Day 1, and surprises become speed bumps not roadblocks.*

4.9 Five Questions to Ask About Your Execution Team

Before adding more people to a project that isn't moving fast enough, I recommend asking these five questions:

01 Can this team take an idea from concept to production without depending on five other teams?

02 Do our engineers understand the business problem and customer outcome—or only the tickets assigned to them?

03 Does every important product and technical decision have one clear owner who can make it quickly?

04 How many days typically pass between building something and showing it to a real user?

05 If we removed 30% of our meetings, handoffs and approvals tomorrow, would delivery suffer—or would the team actually move faster?

Before you add another engineer to a slow-moving project, look at the system around the engineers.

> ***Sometimes the fastest way to increase a team's capacity isn't to add more people. It's to remove what is slowing down the people you already have.***

CHAPTER 5

AI Doesn't Replace Great Engineering

There has probably never been a more exciting time to build software.

An idea that once required a team of engineers and several weeks to prototype can now sometimes be demonstrated in a matter of hours. AI can generate code, create interfaces, write tests, troubleshoot errors and help an engineer work in technologies they may barely know.

I think this is a tremendous opportunity.

But it also creates a dangerous illusion:

> *If software has become easier to create, perhaps software engineering has become less important.*

I believe exactly the opposite is happening.

AI has dramatically lowered the cost and time required to create software. But it hasn't lowered the cost of creating **bad software.**

In fact, when we can build faster than ever before, the choices we make about architecture, security, quality and maintainability become even more important.

AI can get us to **"working"** remarkably fast.

Great engineering gets us from **working to trustworthy.**

5.1 Vibe Coding Is Great - Until It Isn't

I am a fan of the experimentation that AI has made possible.

Describe what you want, generate some code, see what happens, make another prompt, change the interface and try again. What has become known as "vibe coding" can be an incredibly effective way to explore ideas and create prototypes.

For an entrepreneur trying to visualize an idea or a product team testing a concept, this is a major step forward.

The problem begins when we confuse:

> ***"It works"***
> ***with:***
> ***"It's ready."***

A prototype has one primary responsibility: prove that something is possible or worth pursuing.

Production software has many more.

Can we maintain it? Can we secure it? Can we monitor it? Can it handle failures gracefully? Can another engineer understand it six months from now? Can it scale? Can we change one part without unexpectedly breaking three others?

AI-generated code doesn't eliminate these questions.

It simply allows us to reach them much faster.

5.2 Reuse Before You Rebuild

Moving fast also doesn't mean starting every project with an empty repository.

Most applications contain elements that aren't particularly unique to the business: authentication, APIs, data ingestion, monitoring, deployment pipelines, common AI patterns, guardrails and other foundational components.

Why rebuild them every time?

Our philosophy is fairly simple:

> *Innovate where you differentiate. Reuse where you don't.*

This is one of the reasons we created accelerators within our **LuminAI Lab.** As we solve problems across AI, data and applications, we try to turn repeatable engineering patterns into reusable building blocks.

That doesn't mean forcing every client into the same architecture.

It means starting with things we already know work and spending more of the team's time on the parts of the product that actually create competitive advantage.

The fastest engineering team isn't necessarily the team that writes code fastest.

Sometimes it is the team that knows **what doesn't need to be written at all.**

5.3 Architecture Matters More When Code Becomes Cheap

AI is becoming extraordinarily good at creating pieces.

An API. A user interface. A database query. An agent. A test. A data transformation.

But somebody still needs to make sure all those pieces belong together.

That is architecture.

How should the system be divided? Where should the data live? What happens when a service fails? Which APIs should be exposed? How do we authenticate users? What needs to scale? What should remain simple? Where do we need flexibility, and where would flexibility simply introduce unnecessary complexity?

As generating individual pieces becomes easier, systems thinking becomes more valuable.

This is why I believe the role of experienced architects and engineers is evolving rather than disappearing.

Their value increasingly comes not from knowing how to type every line of code, but from understanding the whole system and making good choices about what should, and should not, be built.

AI can help build the pieces.

> *Engineering makes the pieces fit.*

5.4 Build Security and IP Protection in From Day 1

Speed can also tempt teams to take shortcuts that become expensive later.

This is especially important when using AI.

What proprietary information are we giving an AI system? What code is it generating? What libraries or dependencies are entering our application? Are credentials accidentally appearing in prompts or code? Do we understand the licensing implications? Could an AI agent access information or systems it shouldn't?

These aren't questions to ask two weeks before production.

They belong in the engineering process from the beginning.

Use approved enterprise AI tools. Protect secrets and proprietary data. Scan dependencies and licenses. Automate vulnerability scanning. Review generated code. Apply appropriate access controls. Test prompts and AI interfaces for potential abuse.

Whenever possible, make these **defaults in the development and deployment pipeline** rather than relying on someone to remember them.

I like to think about it this way:

> *Security added at the end becomes a gate. Security built in from Day 1 becomes engineering.*

That distinction matters enormously when the goal is to **launch in 90 days.**

5.5 AI Changes the Definition of "Done"

AI also changes something fundamental about software quality.

Traditional software is largely deterministic.

If I enter the correct password, I should be allowed into the application. If I enter the wrong one, I shouldn't. We can write a test and determine whether the software passed or failed.

AI doesn't always behave that way.

Ask a generative AI application the same question twice and you may receive two different answers. Both could be good. Or one could sound extremely convincing and be completely wrong.

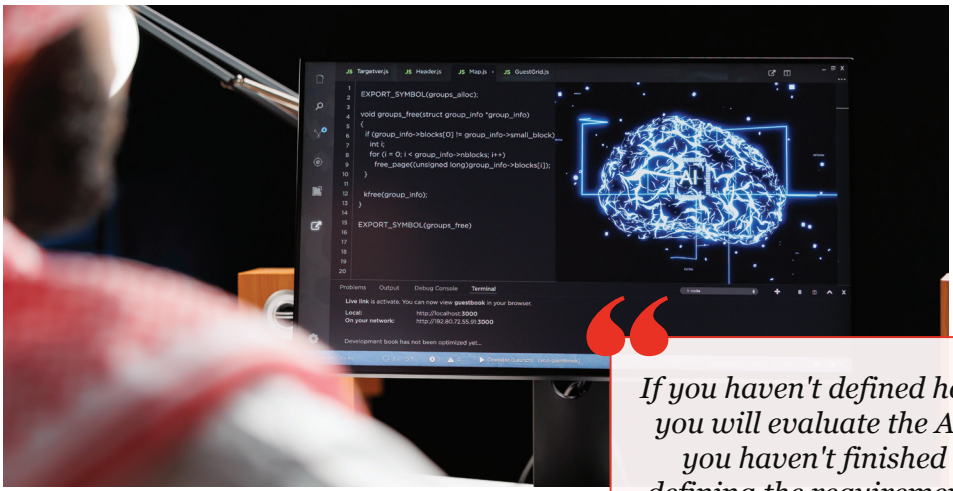
That means the traditional definition of **"the feature works" isn't enough.**

Before launching an AI product, we need to define what a good answer actually looks like.

How accurate should it be? What types of errors are acceptable? Which errors are absolutely unacceptable? How much latency can users tolerate? What should each interaction cost? How often does it hallucinate? How does it perform on difficult or unusual cases?

This is where **AI evaluations, or evals, become the equivalent of acceptance criteria.**

And I would go one step further:



> *If you haven't defined how you will evaluate the AI, you haven't finished defining the requirement.*

5.6 Don't Ask for Perfect - Define "Good Enough"

Executives often ask a reasonable question about an AI solution:

"How accurate is it?"

I think there is a better question:

"What happens when it is wrong?"

An AI assistant recommending an internal knowledge article doesn't have the same risk profile as an AI system making a financial decision or affecting someone's healthcare. So a single accuracy percentage isn't particularly meaningful without understanding the consequence of an error.

Instead, define a risk-adjusted quality threshold for the use case.

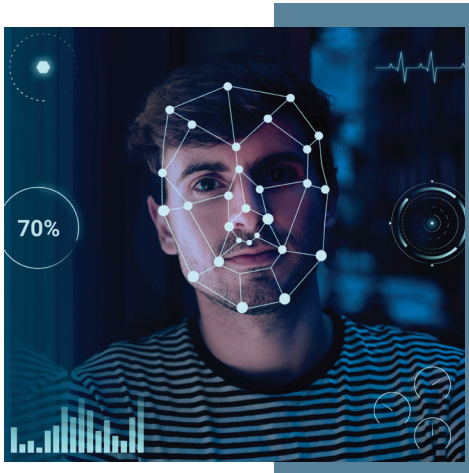
Perhaps high-confidence, low-risk decisions can be automated.

Perhaps uncertain decisions require human review.

And perhaps certain high-consequence decisions should always require explicit human approval regardless of the model's confidence.

A solution that automates **85%** of a process and sends the remaining **15%** to a knowledgeable person may create tremendous business value.

We don't necessarily need AI to replace every human decision. We need it to remove the work where human judgment isn't adding much value, and make sure human judgment is available where it is.



5.7 Know When to Keep a Human in the Loop

This is why I like the concept of **human-in-the loop AI.**

A well-engineered AI system doesn't need to pretend it knows everything.

It should know when to ask for help.

A simple framework might look like this:

High confidence + Low consequence	→	*Automate*
Uncertain + manageable consequence	→	*Human review*
High consequence	→	*Escalate or require approval*

Over time, as the system encounters more real-world examples and our evaluations improve, we may be comfortable automating more.

But I would rather start with the appropriate safety net and earn the right to automate than automate everything on Day 1 and discover the boundaries through customer failures.

The objective isn't to keep people unnecessarily involved.

> *It is to put human judgment exactly where its value is highest.*

AI Is Leverage, Not a Substitute

AI is changing software engineering, and I believe that is overwhelmingly a good thing.

Small teams can accomplish more. Engineers can move faster. Ideas can be tested cheaply. Prototypes can become products in a fraction of the time they once required.

But faster development doesn't eliminate the need for engineering discipline.

It increases the consequences of not having it.

Architecture still matters. Security still matters. Testing still matters. Maintainability still matters. Human judgment still matters.

And perhaps most importantly, someone still needs to understand why we are building the product in the first place.

AI makes great engineers dramatically more productive.

> ***It doesn't replace great engineering.***

A Lesson in What AI Doesn't Understand

Ajay Khandelwal
CTO, Ccube

A while ago, I came across an interesting situation at a large healthcare organization that stayed with me.

They were using an AI-driven system to help automate patient acquisition and marketing attribution. Privacy was understandably a critical requirement, and the system had been designed to aggressively minimize the risk of exposing protected health information.

Everything initially appeared to be working as intended.

Then, a few days into a preventive-care campaign, the analytics began showing something strange. Conversion rates appeared to have dropped dramatically, and successful marketing channels suddenly looked like they were underperforming.

An experienced data engineer decided to look beyond the dashboards and inspect what was happening inside the data pipeline.

The AI had been doing exactly what it had been instructed to do.

To minimize privacy risk, it had started removing certain timing and journey information from incoming events. From a narrow privacy perspective, the decision made sense.

But that information was also necessary to connect the steps in a patient's journey.

Without it, successful bookings could appear to the downstream system as abandoned interactions. The AI then reacted to those signals and began reducing investment in channels that were performing well.

The AI hadn't failed at its objective. It had optimized the objective too literally.

The engineering team intervened, redesigned how sensitive information was protected while preserving the signals required for attribution, and restored the integrity of the downstream analytics.

What stayed with me was the distinction between following a rule and understanding its broader consequence.

AI can optimize remarkably well for the objectives and constraints we give it. But it doesn't automatically understand the business, human or operational context surrounding those objectives.

The lesson was simple:

> *AI can optimize the rules. Great engineering understands the consequences.*

5.8 Five Questions to Ask Before Putting AI Into Production

Before moving an AI application from an impressive prototype into the hands of real customers or employees, I recommend asking these five questions:

01 Are we looking at a prototype that works, or a product we can securely operate, maintain and evolve?

02 Which parts of this solution truly differentiate our business, and which should use proven, reusable components rather than being built again?

03 What proprietary data, source code, credentials, intellectual property or security vulnerabilities could our AI development process inadvertently expose?

04 What does "good enough" mean for this specific AI use case, and have we defined measurable evaluation and quality thresholds before launch?

05 When the AI is uncertain or wrong, what happens next, and when does a human need to take over?

The question isn't whether AI can build something for us.

Increasingly, the answer to that question is **yes.**

The more important question is:

> ***Can we engineer what AI helps us build into something our customers and our business can trust?***

Part III

Launch, Guard & Scale

Days 61–90 & Beyond

CHAPTER 6

Launch Is When the Real Learning Begins

There is a natural tendency to think of launch as the finish line.

The team has spent weeks defining the problem, making architectural choices, building the product, testing it and getting it into production.

We launched.

Time to celebrate.

And we should.

But I think something even more important happens at launch:

> *Our assumptions finally meet reality.*

Until that point, we have been making educated guesses.

We believe customers have this problem. We believe they will use the product this way. We believe this workflow makes sense. We believe the AI response is useful. We believe this feature matters.

Then real users arrive.

And they have a wonderful habit of doing things we never expected.

They ignore features we thought were essential. They use other features in ways we never imagined. They abandon workflows halfway through. They ask questions our AI has never encountered. They find shortcuts. They make mistakes. And occasionally, they show us that the problem we thought we were solving isn't quite the problem they have.

That's not failure.

That's learning.

The objective of a 90-day launch isn't to prove that all our original assumptions were correct.

It is to get a responsibly engineered product into the hands of real users quickly enough that we can discover which assumptions were correct and which ones weren't.

6.1 Build for Adoption Before You Launch

I don't believe adoption should begin with a training plan two weeks before launch.

It should begin during product development.

Bring representative users into discovery. Show them early prototypes. Let them react to workflows before we've invested weeks building them. Put working software in front of them during development. Watch what confuses them.

Ask them to perform real tasks rather than simply asking:

> ***Do you like it?***

People are generally polite when shown a new product.

Behavior is less polite.

A user may tell us a dashboard looks great and then never open it again.

A feature may receive enthusiastic feedback in a demonstration but require so many steps that users return to their spreadsheet.

That is why one metric I particularly like is **Time to First Value.**

> *How long does it take a new user to experience something meaningfully better?*

Can they accomplish something useful in five minutes? Do they need two hours of training? Do they have to change three other processes before the product becomes valuable?

Especially in enterprise software, adoption is rarely just a technology problem.

It is a **workflow problem.**

The easier we make it for the new product to fit into how people work and improve that work, the better our chances of adoption.

> *Don't build the product first and figure out adoption later. Build adoption into the product.*

6.2 Instrument the Product Before You Need the Answers

Once the product reaches users, we need to be able to see what is happening.

This sounds obvious, but it is surprisingly easy to launch a product and discover later that we aren't collecting the information needed to understand how people are using it.

That is why telemetry and logging should be part of the MVP not an afterthought.

> ***After launch, some of the most valuable opinions aren't in the boardroom. They are in the system logs.***

Executives, product managers and engineers can predict how users should interact with a product.

Telemetry shows us how they interact with it.

I like looking at two categories of signals.

Explicit Signals

These are things users tell us directly.

Thumbs up or thumbs down on an AI response. Ratings. Feedback forms. Written comments. Corrections. Support requests.

They are valuable because the user is deliberately telling us something about the experience.

Implicit Signals

These are things users tell us through their behavior.

- Did they copy an AI-generated answer?
- Did they accept a recommendation or override it?
- Did they repeatedly regenerate the response?
- How far did they get through the workflow?
- Where did they abandon it?
- Did they complete the task manually after trying the AI?
- Did they come back the next day?

These behavioral signals can reveal friction that nobody reports.

If users consistently override an AI recommendation, we should understand why.

If they repeatedly abandon the same workflow, something is probably wrong.

If a feature we considered essential is almost never used, perhaps it wasn't as essential as we thought.

Designing this visibility into the product early means we can start learning as soon as real usage begins.

6.3 Listen to Behavior But Don't Stop Talking to Users

There is an important caution here.

Data doesn't always explain itself.

Suppose a user spends five minutes reading an AI-generated response.

Is that good?

Perhaps the answer was extremely valuable and deserved careful consideration.

Or perhaps it was so confusing that the user spent five minutes trying to understand it.

Suppose users frequently copy an AI response.

That might be a strong signal of value.

Or it could mean the application is missing an integration and users are forced to move the information somewhere else manually.

Telemetry tells us **what happened.**

User conversations help us understand **why.**

That is why I don't think product teams should choose between analytics and talking to customers.

We need both.

Use telemetry to identify patterns and anomalies.

Then talk to users to understand them.

I particularly like conversations that start with an observation rather than a hypothetical question:

> ***We noticed that users frequently stop at this point in the workflow. Can you show us what happens here?***

That tends to produce much more useful insight than:

> ***What features would you like us to add?***

6.4 Measure Value, Not Just Usage

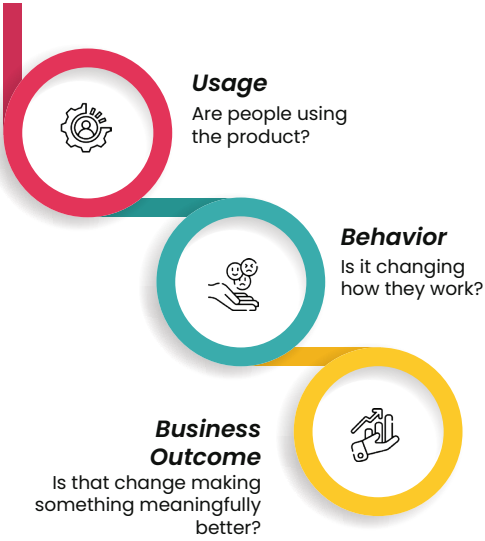
Product teams naturally measure activity.

Monthly active users. Sessions. Logins. Number of AI requests. Features used. Time spent in the application.

These are useful indicators.

But they aren't necessarily evidence that the product is creating value.

I prefer thinking about measurement at three levels:



Imagine we build an AI assistant for a customer-support team.

It could have excellent usage. Every support representative might use it every day.

That's encouraging.

But the real questions are further downstream.

- Are support requests being resolved faster?
- Are fewer issues being escalated?
- Are agents handling more requests without sacrificing quality?
- Are customers getting better answers?
- Is the cost per resolution declining?

Those are business outcomes.

A product that generates a million AI interactions but doesn't improve the underlying business process isn't necessarily successful.

This takes us back to the outcome we established before development began.

> ***Did the product change the thing we originally said mattered?***

If not, more usage may not be the answer.

Don't confuse product activity with business impact.

6.5 Treat Corrections as Learning Data

This becomes particularly interesting with AI products.

When a knowledgeable user changes an AI recommendation, we can treat that simply as an error.

Or we can treat it as **learning data.**

- Why did the user change it?
- Was the model wrong?
- Was important context missing?
- Did retrieval bring back the wrong information?
- Was the prompt unclear?
- Is there a business rule the AI doesn't understand?
- Or is the user's judgment based on knowledge we haven't captured anywhere?

The same applies when a user rejects an AI-generated response, changes a classification or completes a task manually after the AI attempts it.

These aren't just failures.

They are potentially some of the most valuable examples the system can generate.

Over time, patterns in these corrections can help us improve prompts, retrieval, workflows, evaluation datasets and business rules.

Sometimes they reveal something even more important:

> *Our original understanding of the problem was incomplete.*

That is exactly the kind of information we want after launch.

The objective isn't to hide the mistakes.

It is to build a system that **learns from where they occur.**

6.6 Don't Let Feedback Become the Roadmap

Once users start providing feedback, another problem appears.

Everyone has ideas.

One customer wants a new dashboard.

Another wants an integration.

A third wants a completely different workflow.

Someone in sales has a feature they believe will close the next deal.

An executive sees a competitor announcement and wants something added immediately.

Suddenly, the carefully controlled MVP becomes a rapidly growing backlog.

Listening to customers doesn't mean building everything they request.

We need to distinguish between different kinds of feedback.

Critical	security, safety, serious errors or incorrect business outcomes.
Friction	abandonment, repeated overrides, confusing experiences or unnecessary steps.
Improvement	opportunities to make the existing product more accurate, useful, fast or intuitive.
Feature Requests	additional capabilities that may be valuable but aren't necessarily fixing the core experience.

The first two usually deserve attention quickly.

The others require prioritization against the outcomes we are trying to achieve.

And we should look for patterns.

One user's request is interesting.

Twenty users independently struggling with the same workflow is evidence.

The goal is not to respond to every signal. It is to identify which signals matter.

Otherwise, micro-iteration quickly becomes micro-chaos.

6.7 Sometimes the Best Improvement Is Subtraction

Product development has a natural bias toward adding things.

More features. More options. More settings. More workflows. More intelligence.

Feedback loops should also help us identify what we can remove.

If telemetry shows that nobody uses a feature, ask why it exists.

If users consistently skip a step, ask whether the step is necessary.

If an AI workflow requires users to make five decisions before receiving value, ask whether three of those decisions can disappear.

If customers repeatedly work around something we've built, perhaps the workaround is telling us something.

This connects back to the complexity budget we discussed earlier.

Complexity doesn't stop accumulating after launch.

Every new feature consumes some of that budget.

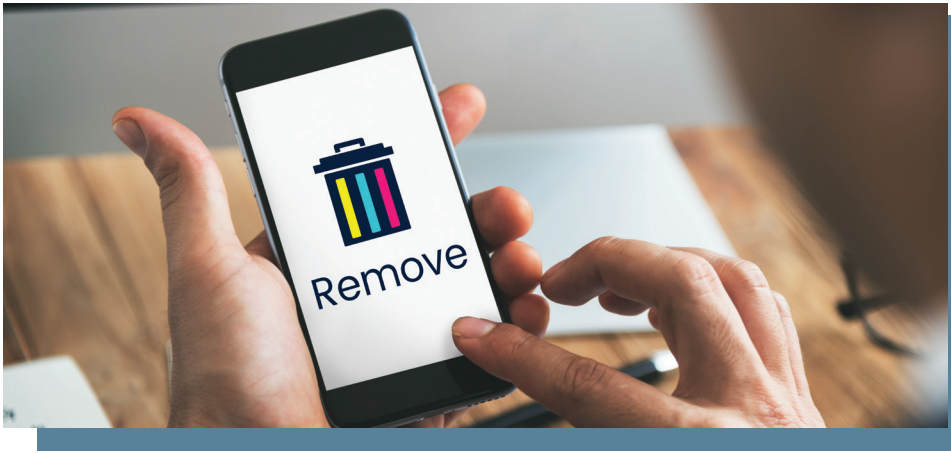
Strong product teams don't just ask:

"What should we build next?"

They also ask:

"What can we make simpler?"

Sometimes removing one step creates more value than adding another feature.



6.8 Create Short, Disciplined Learning Loops

Finally, feedback only has value if it changes something.

Collecting dashboards, customer comments and AI ratings isn't a feedback loop.

A loop must close.

Learning Loops

Product teams sometimes make a change based on feedback and immediately move on to the next item.

- But did the change work?
- Did abandonment decline?
- Did AI acceptance improve?
- Did the workflow become faster?
- Did the business outcome move?

If we don't measure again, we have simply replaced one assumption with another.

The advantage of a small, well-designed product team is that this loop can be extremely short.

We shouldn't need three months to respond to an important learning.

Sometimes the cycle can happen in days.

This is where the speed we created in the earlier chapters becomes especially valuable.

The purpose of moving fast isn't simply to launch sooner.

It is to **learn sooner.**

Day 90 Is Not the Finish Line

When we talk about launching a product in 90 days, I don't want that to suggest that the product is finished on Day 90.

Quite the opposite.

Before launch, much of what we believe about the product is still based on assumptions.

After launch, we finally have evidence.

Real users.

Real behavior.

Real mistakes.

Real corrections.

Real business outcomes.

The organizations that learn fastest from that evidence have an enormous advantage.

They don't spend six months debating what customers might want.

They put something useful in front of them, observe what happens and improve it.

That doesn't mean chasing every user request or changing direction every week.

The advantage of a small, well-designed product team is that this loop can be extremely short.

We shouldn't need three months to respond to an important learning.

> ***It means creating a disciplined system for turning behavior into insight, insight into improvement, and improvement into measurable business value.***

That is the real feedback loop.

And in my experience, it is one of the biggest differences between a product that merely launches and one that gets progressively better.

A Lesson in Solving Beyond the Request

Sonam Bansal
Vice President, Ccube

Recently a client in the food and beverage industry came to us with what seemed like a simple request.

They operate a platform that helps cafes and food businesses manage their recipes. Their team was spending hours manually creating recipe instructions, documenting every step, tool and time estimate for each item. They wanted to know whether AI could generate those instructions directly from a list of ingredients.

We could have built exactly what they asked for.

But as our team studied the actual workflow, we started asking different questions.

What happens if an ingredient is missing? What if the quantity doesn't make sense? What if the AI generates the cooking steps in the wrong sequence like adding garlic before the oil is hot?

These weren't really edge cases. They were problems that could occur every day in a busy kitchen.

So, our lean team of two AI engineers and a designer built the requested capability but went one step further.

In addition to generating recipe instructions, cooking sequences and tool recommendations, the solution validated ingredients, identified questionable quantities and flagged combinations that didn't make sense. And importantly, users could accept, edit or override every recommendation.

The AI assisted. It didn't decide.

When we demonstrated it, the conversation quickly changed from:

"Can AI do this?"
to:
"What else can AI do for us?"

They had come expecting a notepad.
We showed them a sous chef.

I have found this to be true in many projects. Clients naturally describe the problem they can see. Our responsibility as product engineers is to understand the workflow deeply enough to sometimes see the problem behind the problem.

The lesson was simple:

> *Don't just build what was requested. Understand why it was requested and solve the problem that matters.*

A Lesson in Starting Small

Antriksha Somani
Vice President, Ccube

A fast-growing digital visual media platform came to us with an ambitious list of ideas for using AI to help its content creators work faster and improve reader engagement.

The possibilities were everywhere, automated tagging, better search, image context generation, content optimization and several other ideas.

We could have turned that list into a six-month roadmap.

Instead, we asked a simpler question:

"Where is the biggest friction in the workflow today?"

One problem stood out. Creators were spending too much time formatting, tagging and optimizing content before they could publish it.

So we started there.

Rather than waiting for a larger data modernization effort, we extracted only the historical data needed to give the AI useful context. We built a lightweight, modular assistant around the existing platform rather than changing its core architecture.

And importantly, we kept the creators in control.

The AI could recommend tags, formatting and content improvements, but users could accept, edit or reject every suggestion with a click.

Within **four weeks,** the assistant was tested and running in production.

The results were significant. Content preparation time dropped **by more than 50%,** allowing creators to publish more frequently without sacrificing editorial control.

But something else happened that I found equally valuable.

Every accepted, edited or rejected suggestion gave us real information about what users found useful. Within days of launch, we were improving the prompts based on actual behavior rather than debating what users might want.

The lesson for me was simple:

> *Start with one meaningful problem, use only the data you need, and get the solution into real users' hands. Then let evidence—not the original wishlist—tell you what to build next.*

6.9 Five Questions to Ask After Launch

Once your product is in the hands of real users, I recommend asking these five questions:

01 What user behavior would tell us this product is genuinely creating value not simply being used?

02 Have we instrumented the product so we can see where users succeed, struggle, override the AI or abandon the workflow?

03 Are we combining what telemetry tells us happened with real user conversations that help us understand why?

04 What measurable business outcome should improve if this product is working and is it moving?

05 How quickly can we turn an important user signal into a product improvement and then measure whether the change worked?

The launch date is important.

But it isn't the moment when we finally know the answers.

It is the moment when we can finally start asking the questions with **real evidence in front of us.**

> ***Day 90 isn't where product development ends. It's where assumptions finally meet reality, and the real learning begins.***

CHAPTER 7

Build AI That Makes Economic Sense

One of the wonderful things about building with AI today is how inexpensive it can be to get started.

A development team can connect to a powerful model through an API, experiment with a few hundred requests, build an impressive prototype and spend surprisingly little money.

That creates another potential illusion:

> *If the prototype is inexpensive, the product will be inexpensive.*

Not necessarily.

A few hundred requests can become a few hundred thousand. A simple AI interaction can turn into multiple model calls, retrieval queries, tool invocations and retries. An agent can perform ten steps behind the scenes for what looks to the user like one simple request.

And suddenly, the product everybody is excited to see growing is producing an equally fast-growing infrastructure bill.

I don't think the answer is to become obsessed with AI costs during the first week of experimentation.

The objective of an MVP is still to determine whether we can create meaningful value.

But once we believe we can, another question becomes equally important:

> *Can we create that value economically at scale?*

An AI solution isn't economically viable because the prototype was inexpensive.

It's viable when its unit economics still make sense after it succeeds.

7.1 The Prototype Bill Isn't the Production Bill

During an MVP, AI costs can be deceptively small.

Imagine that **50 people** are testing an application. Each person performs **ten AI transactions** per day.

The economics probably don't attract much attention.

Now imagine **50,000 users** doing the same thing.

A technical decision that barely registered during development can suddenly become a significant operating expense.

This is why I think AI teams should model at least a few simple scenarios before launch.

What happens at:

100 users? | *10,000 users?*

1 million transactions per month?

We don't need elaborate financial models.

We need to understand whether the architecture behaves reasonably if the product does what we are hoping it will do: **grow.**

And cost doesn't always scale linearly.

Success can introduce additional infrastructure, concurrency, retrieval, storage, monitoring and support requirements.

So before celebrating a low prototype bill, ask:

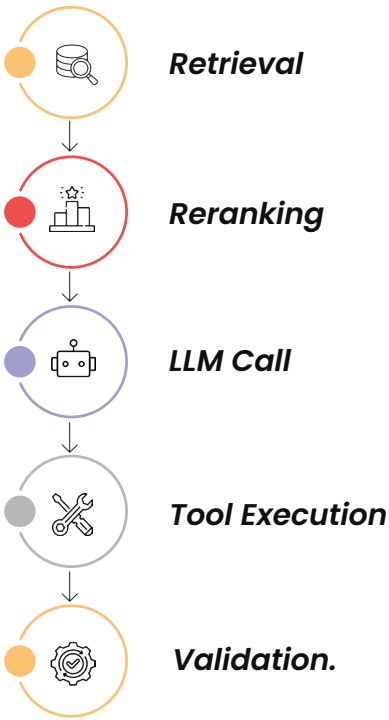
> *What would this cost if customers actually love it?*

That is a much more useful number.

7.2 Know What One Successful Outcome Actually Costs

One of the easiest mistakes is measuring only the cost of the primary AI model.

Single user queries frequently trigger multi-step architectural loops



Measure Cost-per-Successful-Outcome (CSO) across all sub-steps rather than tracking isolated LLM API token costs.

That is why I prefer measuring Cost per Successful Outcome rather than simply cost per model call or tokens consumed.

The complete cost may include:

- ✓ Model inference
- ✓ Embeddings and retrieval
- ✓ Reranking
- ✓ Tool and external API calls
- ✓ Compute and storage
- ✓ Agent steps
- ✓ Retries
- ✓ Validation
- ✓ Human review

Not every application will have all of these.

The important point is to understand the entire cost of completing the job successfully.

Suppose an AI workflow costs $1.50 but eliminates $25 of manual effort.

That may be excellent economics.

On the other hand, an interaction costing only a few cents is still wasteful if it creates no meaningful value.

Which leads to an important distinction:

Low cost isn't the objective. Good unit economics are.

7.3 Use the Least Expensive Intelligence That Reliably Does the Job

In Chapter 3, we discussed using the right intelligence for the right workload.

That principle has a significant economic benefit.

Not every task needs the smartest model available.

A sophisticated reasoning problem may justify a highly capable model.

A straightforward classification, extraction or summarization task may not.

This is where a Model Gateway or AI Router becomes particularly valuable.

Instead of sending every request to one model, the architecture can route workloads based on their requirements.

But I wouldn't optimize on cost alone.

The decision should balance:

The cheapest model isn't a bargain if its output causes customers to redo the work.

And the most capable model isn't good engineering if a model costing one-tenth as much performs the task equally well.

The objective is:

> *Use the least expensive intelligence that reliably meets the required quality threshold.*

This is also where the AI evaluations we discussed earlier become valuable.

If we have objective quality thresholds, we can compare models based on evidence rather than reputation.

Perhaps a smaller model achieves virtually the same results as a premium model for 70% of our workload.

That's not simply a model optimization.

It can fundamentally change the economics of the product.

But I wouldn't optimize on cost alone.

7.4 The Cheapest AI Call Is the One You Don't Make

There is another way to reduce AI cost that is sometimes overlooked:

> ***Don't call AI.***

Not every interaction requires fresh inference.

If the same question has already been answered and the underlying information hasn't changed, perhaps the response can be cached.

If thousands of similar tasks can be processed together, perhaps they can be batched.

If thousands of similar tasks can be processed together, perhaps they can be batched.

If a simple business rule can determine the answer reliably, use the business rule.

If information can be retrieved directly, perhaps we don't need another reasoning step.

If an expensive result can be calculated once and reused safely, don't calculate it repeatedly.

This isn't about removing AI from an AI application.

It is about using AI where intelligence is required.

Consider a workflow where AI interprets an incoming request.

Once the intent is understood, perhaps the next five steps are completely deterministic.

There is little value in asking a language model to reason through each one.

This connects directly to the architectural principle we discussed earlier:

> *Use AI where ambiguity requires intelligence. Use software where certainty requires rules.*

That principle improves reliability.

It also improves economics.

Sometimes the smartest optimization is not finding a cheaper model.

> ***It's realizing that you don't need a model at all.***

7.5 Give Every Agent a Budget

Agentic AI introduces another interesting economic challenge.

Traditional software generally follows a path we designed.

An AI agent may decide what steps to take.

It might search for information, call a tool, inspect the result, reason about what to do next, call another tool, ask another model and retry if something doesn't work.

That autonomy can be incredibly powerful.

It can also become expensive surprisingly quickly.

If we give an agent autonomy, I believe we should also give it an **economic boundary.**

Depending on the use case, that might include:

- ✓ Maximum model calls per task
- ✓ Maximum tool calls
- ✓ Maximum retries
- ✓ Maximum tokens or context size
- ✓ Maximum execution time
- ✓ Maximum spend per workflow

When the agent reaches that boundary, something deliberate should happen.

> *Perhaps it will switch to a less expensive model.*

> *Perhaps it asks the user for clarification.*

> *Perhaps it returns the best result it has.*

> *Perhaps it escalates to a human.*

What it shouldn't do is continue consuming resources indefinitely because nobody defined when it should stop.

We already put security boundaries around what agents are allowed to do.

We should put economic boundaries around **how much they are allowed to do.**

The principle is simple:

> ***Give the agent a job and a budget.***

7.6 Make the Gateway an Economic Control Point

The AI Gateway introduced during architectural design acts as your primary financial control plane. Instead of allowing components to issue unthrottled API requests, the gateway enforces strict token governance and economic policies:

- How large can an input be?
- How much output can a model generate?
- Which models can a particular workload use?
- How many requests can a user make?
- How many times can a failed request retry?
- What happens when usage suddenly spikes?
- What happens when a team or customer exceeds its budget?

These controls can include rate limits, quotas, request-size limits, model-routing rules, concurrency limits, spending thresholds and anomaly detection.

And they should be observable.

If usage suddenly increases tenfold at 2:00 a.m., someone or ideally something should notice.

In certain situations, the application may even degrade gracefully rather than simply fail.

A lower-priority workload could move to a less expensive model.

A nonessential AI feature could temporarily be limited.

An unusually expensive workflow could require approval.

The specific policy depends on the product.

The principle doesn't.

Don't let product success become an infrastructure surprise.

7.7 Connect Cost to Business Value

Ultimately, I don't think AI cost should be discussed in isolation.

Executives don't really care whether an AI transaction costs two cents or two dollars.

They care what they receive in return.

Suppose an AI agent costs $2 to complete a task that previously required an employee 30 minutes.

That could be very attractive.

Suppose another AI feature costs only $0.02 per interaction but serves millions of interactions nobody finds useful.

That may be terrible economics.

So alongside Cost per Successful Outcome, I would measure something like:

Value per Successful Outcome

The value might come from:

- ✓ Labor hours saved
- ✓ Faster customer response
- ✓ Higher conversion
- ✓ Increased revenue
- ✓ Fewer errors
- ✓ Lower support costs
- ✓ Reduced operational risk
- ✓ Greater employee capacity

Not every benefit can be measured perfectly.

But connecting cost to an outcome changes the conversation.

Instead of asking:

> ***"How do we reduce our AI bill?"***

we can ask:

> ***"How do we maximize the economic value this AI system creates?"***

Those are very different questions.

One leads primarily to cost-cutting.

The other leads to better product and engineering decisions.

7.8 Optimize After You Understand

There is also a danger in taking all of this too far.

Teams can spend weeks optimizing token usage, model selection and caching before they have established whether anyone actually wants the product.

Saving 30% on an AI workflow that shouldn't exist isn't optimization.

It's a distraction.

During the earliest stage of an MVP, I care primarily about three things:

- Can we solve the problem?
- Does the solution create meaningful value?
- Will people use it?

Once we begin answering those questions positively, economics deserves increasing attention.

Then we can look at real usage.

- Which workflows are expensive?
- Which model calls are unnecessary?
- Where are we retrying too often?
- Could a smaller model handle some requests?
- What can be cached?
- What should become deterministic?
- Where is human review adding cost without sufficient value?

Now we are optimizing based on evidence rather than assumptions.

This is another advantage of getting into users' hands quickly.

Real usage tells us where optimization matters.

Design for Successful Scale

When engineering teams talk about scalability, we naturally think about technical scalability.

- Can the system handle more users?
- Can the infrastructure process more transactions?
- Can the database support the load?

AI introduces another dimension:

Can the economics scale?

A system capable of serving one million users isn't necessarily a successful architecture if serving those users destroys the economics of the product.

That is why I believe cost belongs alongside performance, security, quality and reliability as an engineering consideration.

Not because every engineer should become an accountant.

But because architectural choices create economic consequences.

- Model selection creates cost.
- Agent autonomy creates cost.
- Retries create cost.
- Context size creates cost.
- Architecture creates cost.

And good engineering can influence every one of them.

The objective isn't to build the cheapest AI system possible.

It is to build one where ***the value created grows faster than the cost required to create it.***

7.9 Five Questions to Ask About Your AI Economics

Before scaling an AI product beyond the MVP, I recommend asking these five questions:

01 Do we know the true end-to-end cost of completing one successful AI transaction, not simply the cost of the primary model call?

02 Are we using the least expensive model that reliably meets the quality, latency and risk requirements of each workload?

03 Which AI calls could we eliminate through caching, retrieval, deterministic software, batching or reuse?

04 Have we placed appropriate economic boundaries around agents, retries, token usage, tool calls and unexpected demand?

05 If usage increases 100×, do our unit economics still make sense and what measurable business value does each successful transaction create?

I particularly like the last question.

Because the real test of an AI product isn't whether we can afford to demonstrate it to 50 people.

It is whether we would be delighted or terrified if 50,000 people started using it tomorrow.

> ***Build for value first.*** ***Measure the economics. Then make sure success is something your architecture and your business can afford.***

CHAPTER 8

From Project to Product

Day 90 is an exciting day.

The product is live. Real users are interacting with it. The team has moved from an idea to something tangible, useful and measurable.

But something important changes on Day 91.

> *You no longer have a project to finish. You have a product to run.*

That requires a different mindset.

During the first 90 days, much of the team's energy is focused on solving the problem, validating assumptions, making architecture choices and getting the product into users' hands.

After launch, the priorities begin to shift.

Is the product creating the value we expected? Are people adopting it? Is the AI still performing as intended? Are the economics working? What is changing in production? And when something goes wrong, can we respond quickly without disrupting the business?

The objective is no longer simply to build and launch.

It is to operate, learn and improve.

8.1 Production AI Doesn't Stand Still

Traditional software can change when we change the code.

AI systems can change for many other reasons.

A model provider may release a new version. A prompt may be modified. Retrieval data changes. User behavior evolves. Business policies change. The types of questions customers ask may be very different six months after launch.

And over time, the quality of an AI system can drift away from what we originally tested.

That is why I believe production AI needs more than traditional application monitoring.

In traditional software, we monitor whether the system is running. With AI, we also need to monitor whether it is still behaving as intended.

This means tracking the important elements that influence behavior.

- Which model was used?
- Which prompt version generated the response?
- What data was retrieved?
- How did the output perform against our evaluations?
- What changed?
- Who approved the change?
- Can we reproduce what happened?

This version history also becomes increasingly important for security, auditability and compliance.

The objective isn't to create bureaucracy around every prompt change.

It is to make sure that when behavior changes, we can understand why.

8.2 Make Every Change Safe to Reverse

Once a product is in production, speed has a different meaning.

During experimentation, we may be comfortable changing something and immediately seeing what happens.

When customers or employees depend on the product, the consequences are different.

A new model may improve quality on one task and unexpectedly degrade another.

A new prompt may solve one problem but create a different one.

An updated retrieval strategy may return better information for most users while failing for an important subset.

That is why production changes should have both a safe path forward and a safe path back.

Depending on the product, that may mean controlled rollouts, evaluation thresholds, versioning, automated monitoring and rollback protocols.

We don't necessarily have to expose every user to a change at once.

- *Test it.*
- *Measure it.*
- *Expand it.*

And if important quality, reliability or safety thresholds are crossed, be prepared to return quickly to the previous stable version.

Moving fast in production isn't about making changes quickly. It's about making changes that are safe to reverse.

8.3 A Product Needs an Owner After the Project Ends

There is another risk after launch that has nothing to do with technology.

Ownership becomes unclear.

During development, everyone knows who is responsible for getting the product launched.

After launch, responsibility can quietly become distributed across product, engineering, operations, security and business teams.

And when everybody owns something, sometimes nobody really owns it.

Every important product needs someone who remains accountable for its outcomes after launch.

- Who owns adoption?
- Who owns AI quality?
- Who owns the roadmap?
- Who watches the economics?
- Who decides whether a production issue requires intervention?

> Who makes the tradeoff between a new feature and improving something users are already struggling with?

Different people may perform these activities.

But someone needs to own the overall outcome.

Launch shouldn't transfer ownership from everybody to nobody.

8.4 Keep the Operating Scorecard Simple

By Day 91, there can be an enormous amount of data available.

The temptation is to create dashboards containing dozens of metrics.

I prefer something simpler.

At the leadership level, I would want to understand six things:

Value	Is the product improving the business outcome we built it to improve?
Adoption	Are the right users using it?
Quality	Is the AI or software producing results that meet our defined standards?
Reliability	Is the product available & performing consistently?
Economics	Is the cost per successful outcome sustainable as usage grows?
Risk	Are security, privacy, compliance or AI-related risks changing?

Each product will have different metrics underneath these categories.

The important thing is not the dashboard.

It is whether leadership can look at a small number of indicators and answer a simple question:

> *Is this product getting healthier or less healthy?*

8.5 Know the Difference Between Learning and an Incident

AI systems will make mistakes.

The goal isn't to pretend they won't.

The goal is to know what to do when they do.

A user correcting a low-risk recommendation may simply become valuable learning data.

Repeated poor answers in one workflow may indicate that a prompt, retrieval process or model needs improvement.

A significant quality degradation across the system requires more immediate intervention.

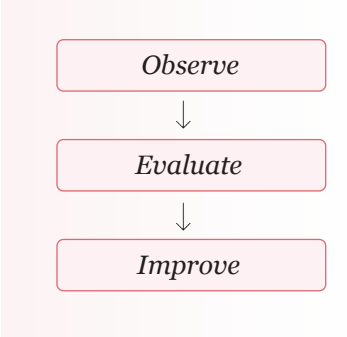
A security, privacy, compliance or high-consequence business failure may require the system or part of it to be stopped or rolled back immediately.

Not every mistake should create an emergency.

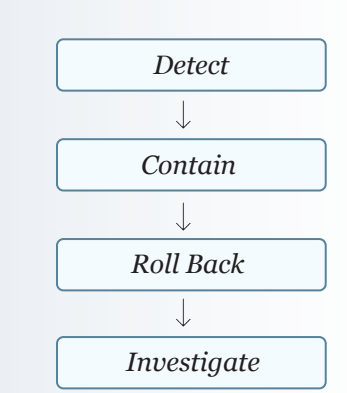
And not every mistake should be treated as another backlog item.

Teams need to know the difference.

For normal product learning:



For a material production incident:



That distinction helps teams continue learning quickly without becoming careless about the failures that matter.

Day 91 Is Where the Product Earns Its Future

The purpose of everything we have discussed in this book isn't simply to reach a launch date.

It is to create something useful enough to deserve a life beyond the project that created it.

The first 90 days give us a disciplined way to move from an idea to a real product without spending months or years trying to anticipate every possible requirement.

But Day 90 doesn't eliminate uncertainty.

It gives us something much more valuable:

real evidence.

From Day 91 forward, our job is to use that evidence well.

- Keep learning from users.
- Keep measuring business value.
- Keep improving quality.
- Keep watching economics.

Make changes safely.

And maintain clear ownership of the outcome.

8.6 Five Questions to Ask on Day 91

01 Who owns this product and its business outcomes after Day 90?

02 Can we identify exactly what changed when AI quality or behavior changes?

03 Can we safely roll back a model, prompt or application change without disrupting the business?

04 Are we continuously measuring value, adoption, quality, reliability, economics and risk?

05 Does the team know which failures are learning opportunities, and which require immediate intervention?

The goal of 90 days was never simply to finish a project.

It was to create something valuable enough to become a product worth operating, learning from and continuously improving.

> *Day 90 is the launch. Day 91 is where the product earns its future.*

Have an idea you're considering?

Bring us your concept, architecture or existing plan. In a Strategy Session, we'll help pressure-test the scope, team, architecture and 90-day roadmap.

No pitch. Just a practical working session.

Email us at

sales@ccube.com

Or scan this ***QR code:***

About the Authors

Scan to Connect on LinkedIn

Mahesh Lalwani *is the Founder and CEO of Ccube, a technology services company helping organizations turn data into products, intelligence into action, and AI into measurable business outcomes. He has spent more than 25 years building and leading technology initiatives across startups and global enterprises, including Apple, Lam Research, Genentech, Mphasis and Wipro.*

Throughout his career, Mahesh has worked at the intersection of business and technology helping teams simplify complex problems, modernize data and software platforms, and turn emerging technologies into practical solutions. Today, much of his work focuses on helping organizations move from ideas to production-ready AI, data and digital products through small, highly engaged engineering teams and a 30-60-90 day execution philosophy. He is also passionate about storytelling, mentoring technology leaders, and helping teams imagine and build what is possible with AI.

Scan to Connect on LinkedIn

Antriksha Somani *is an engineering leader at Ccube. An active open-source contributor to projects like Sourceloop, Antriksha is passionate about engineering products that deliver real business value. His philosophy centers on focusing on outcomes rather than just activity, utilizing a 30-60-90-day approach to help organizations validate ideas quickly and turn concepts into scalable, practical solutions.*

What if you could go from idea to launch in 90 days without cutting corners?

AI has dramatically changed how quickly we can build software. But faster coding doesn't automatically create better products.

The real challenge is making the right choices: what problem to solve, what not to build, how to choose an architecture without overengineering it, how to assemble a lean team that can move fast, and how to turn an early idea into something customers can trust and use.

From Idea to Launch in 90 Days shares practical lessons from more than 25 years of building technology solutions across startups and global enterprises.

Inside, you'll learn how to:

- Start with the problem not the technology
- Make pragmatic AI and architecture choices without getting locked in
- Build lean, highly engaged teams that move fast
- Turn real user behavior into faster product learning
- Use AI to accelerate engineering without sacrificing quality, security or trust
- Build AI products with sustainable economics and a plan for Day 91 and beyond

This isn't a book about building more software.

It's about making better choices, learning faster, and building what matters.

Visit Our Website:

Mahesh Lalwani
Founder & CEO, Ccube