by Chandi Datta

AWS BEDROCK AGENTS

Building Production-Ready Autonomous Workflows for the Enterprise

A Practical Guide for Developers & Architects:
Covers Production Architecture, IAM, Data Modeling, and Cost Engineering

Enterprise AI Agents: From POC to Production on AWS

Chandi Datta

This book is available at https://leanpub.com/enterprise-ai-agents

This version was published on 2026-04-12

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To the engineers and platform teams who build the hard things behind the scenes.

Contents

Preface

This book started as a Slack message.

After nine months of building AI agents on AWS Bedrock in a large enterprise, I sent a message to my team lead: "We should write down everything we learned. Nobody else is writing about this stuff — the real stuff, not the tutorial stuff."

The "real stuff" was everything that happened after the demo. The demo took five days. Getting the agent to production took the rest of the year. And the gap between those two milestones was filled with problems that no AWS documentation, no Medium article, no conference talk had prepared us for.

VPC endpoints that silently dropped connections. IAM policies with explicit denies that blocked every deployment path we tried. A managed service that promised stateful workflows but forgot your state every 60 seconds. A database that slowed to a crawl because we used DELETE instead of TTL. A proxy configuration change at 2 AM that took the entire agent offline — and left zero error logs to explain why.

None of this was in the Bedrock getting-started guide.

Who This Book Is For

You are an engineer, architect, or tech lead at a company with more than a few hundred engineers. Your company uses AWS. Someone — maybe you, maybe your VP — decided that AI agents are the next thing to ship. You built a demo, it worked great, and now you need to put it in production.

That is where this book picks up.

I am not going to teach you what a large language model is. I am not going to walk you through the AWS console to create your first agent. AWS has tutorials for that, and they are fine. What AWS does not have is a guide to surviving the enterprise reality: the IAM policies that block you, the networking that silently breaks, the state management you have to build yourself, the cost tracking you forgot until it was too late, and the security review that takes longer than the actual coding.

This book is that guide.

One piece of framing before we start: an AI agent is not a new kind of software. It is a Lambda function where the decision-making happens to be a language model instead of a switch statement. It reads inputs, picks an action, executes it, and checks the result. The LLM replaces the if/else tree — it does not replace

the infrastructure around it. You still need IAM roles, VPC endpoints, deployment pipelines, monitoring, and cost controls. Everything you already know about building production systems still applies. The LLM is a component, not a revolution.

This matters because the marketing around AI agents implies you are building something fundamentally new. You are not. You are building automation. The sooner your team internalizes that framing, the faster you will ship.

But here is what makes this automation different from every integration you have built before: it can reason. A traditional automation fails on an unexpected API response and calls you at 3 AM. An agent reads the error, correlates it with what it knows about the system, and decides whether to retry, escalate, or take a different path — all with guardrails that prevent it from doing anything destructive. That gap between "automation that follows a script" and "automation that understands the situation" is where enterprise AI agents live. And that gap is worth the engineering effort it takes to get there.

How This Book Is Organized

Part 1 (Chapters 1-2) sets the stage. What makes enterprise different, how Bedrock Agents actually work under the hood, and where the managed service stops and your code begins.

Part 2 (Chapters 3-5) covers building the agent. Prompt engineering that works in production (not in demos), action groups and tool integration — including the "Agent Factory" pattern that lets non-coders build agents — and data architecture for real-time workflows.

Part 3 (Chapters 6-9) is the hard part. IAM and security, enterprise networking, deployment automation, and cost engineering. These chapters exist because they represent where we spent 70% of our time. The code was the easy part. Getting it through security review, deploying it behind VPC endpoints, and keeping the LLM bill under control — that was the job.

Part 4 (Chapters 10-13) covers production operations. Testing non-deterministic systems, observability that actually helps you debug agent behavior, a production checklist, and the full list of lessons we learned the hard way.

The Appendices contain ready-to-use templates: CloudFormation, IAM policies, agent instruction samples, a cost calculator, and a troubleshooting guide with the 13 most common errors we hit.

A Note on Code Samples

All code in this book reflects real-world enterprise patterns that I have encountered and worked with. Company names, internal URLs, application identifiers, and API schemas are fictional — created to illustrate the architecture without referencing any specific organization. The patterns, the failure modes, and the solutions are drawn from hands-on experience. The names and numbers are made up.

Every CloudFormation template, IAM policy, and code sample uses parameterized placeholders (marked with `# REPLACE` comments). Copy them, fill in your values, and they work. We designed them that way on purpose — a template with your company's ARNs hardcoded is useless to everyone else. A template with `${AWS::AccountId}` placeholders is useful to everyone.

Acknowledgments

This book would not exist without the team that built the system it describes. We argued about architecture in pull request comments, debugged proxy issues at midnight, and spent three weeks in meetings about a single IAM permission. The lessons in this book are theirs as much as mine.

Thanks also to the enterprise platform and security teams who said "no" to our first seventeen deployment attempts. You made the final architecture better. We did not enjoy the process.

* * *

This book reflects the state of AWS Bedrock Agents as of early 2026. AWS moves fast. Some limitations described here may be addressed by the time you read this. The architectural patterns and enterprise constraints, however, tend to stick around.

Chapter 1: Why Enterprise AI Agents Are Different

1.1 POC Is Easy. Production Is War.

Building an AI agent demo takes a week. Getting it to production in an enterprise takes a year. This book is about that year.

I know this because I lived it. I spent over a year building AI agent systems for a large enterprise — real agents that manage real infrastructure, make real decisions, and fail in real ways that no tutorial prepared me for. By the time I write this, I have deployed agents that automate infrastructure operations across multiple environments, handle long-running workflows that span hours, and make autonomous decisions about whether to retry, escalate, or fix problems on their own.

The hard part was never the AI. Not even close. The hard part was everything around the AI.

Every AWS tutorial shows you how to create a Bedrock agent in 15 minutes. And it works! You get a functional demo that answers questions, calls tools, and looks impressive in a slide deck. Your VP nods approvingly. Your product manager starts planning the rollout.

Then reality hits.

Your IAM team says the agent role does not conform to their managed policy structure. The security team requires KMS encryption on everything — and the Bedrock API throws cryptic errors when you misconfigure the key. The networking team insists on private endpoints only — no public internet, no exceptions. The compliance team wants audit trails for every decision the agent makes. The finance team sees the first LLM bill and schedules an urgent meeting.

That 15-minute demo is now a 15-month project. And the AI part? It is maybe 20% of the work.

The Demo That Fooled Us

Our first demo was embarrassingly simple:

1. Create a Bedrock Agent in the AWS console

2. Attach a Lambda function as an action group
3. Give it a simple instruction: "You are a helpful assistant that manages infrastructure"
4. Invoke it from the Bedrock console
5. Watch it call the Lambda and return a response

Total time: 20 minutes. Total lines of code: about 50. We were excited. We thought we would be in production by next quarter.

We were wrong by roughly four quarters.

The Timeline Nobody Shows You

Here is roughly how our year went, so you know what to expect:

Month	What Happened
Month 1	Built the POC. Agent calls tools, returns answers. "We'll be done in a quarter."
Month 2	Hit the Lambda timeout wall. First real workflow exceeds 8 minutes. Started building custom state management.
Month 3	IAM gauntlet begins. Three weeks fighting `iam:PassRole` denies and KMS key policies. First successful CloudFormation deployment.
Month 4	Networking nightmares. VPC endpoints, proxy config, `NO_PROXY` debugging. Lambda hangs silently — no errors, just timeouts.
Month 5	Architecture pivot. Collapsed three agents into one with nine tools. Debugging time drops 80%.
Month 6	First deployment to non-prod. Everything breaks differently outside of dev. Config differences, API endpoint changes, IAM policy variations across accounts.
Month 7–8	Stabilization. Built error classification, notification formatting, retry logic. Added per-call token tracking after the first cost surprise.
Month 9–10	Testing infrastructure. Golden datasets, evaluation pipelines, LLM-as-a-Judge. Learned that testing non-deterministic systems is its own discipline.
Month 11–12	Production hardening. Observability, alerting, production checklist reviews with security and platform teams. Human-in-the-loop approval flows for high-risk actions.

Month	What Happened
Month 13+	Production. Agents running real workflows. Started building the Mini-MCP framework so SREs could create agents without writing Lambda functions.

If your timeline is shorter, great — you probably have lighter enterprise constraints. If it is longer, you are not alone. The point is that the AI part (month 1) is a small fraction of the total effort.

What Nobody Warns You About

The gap between "it works in the console" and "it's running in production" is filled with problems that no tutorial covers:

- **State management**: Your agent needs to run jobs that take 30+ minutes. Lambda times out at 15. Where does the state go between invocations? (Answer: you build it yourself — Chapter 5 covers the full data architecture.)
- **Persistent memory**: The agent completes step 3 of a 9-step workflow and the Lambda container dies. How does the next invocation know what already happened? (Answer: S3 checkpoints and optimistic locking — see Chapter 5.4.)
- **IAM hell**: You need `iam:PassRole` to attach a role to the agent. Your enterprise has an explicit deny on `iam:PassRole`. The CloudFormation service role can bypass it, but only if someone in the cloud platform team configures it. That takes three weeks and four meetings. (Chapter 6 has the full escape hatch.)
- **Networking**: The agent needs to call external APIs (build servers, traffic managers) AND AWS services (Bedrock, S3, SNS). Each goes through a different network path. Get it wrong and your Lambda hangs silently for 30 seconds, then times out. (Chapter 7 covers VPC endpoints, proxies, and the debugging flowchart.)
- **Cost surprises**: Each LLM call has input tokens and output tokens, priced differently. A long-running agent workflow can make 800+ LLM calls. Without tracking, you discover the cost at month-end. (Chapter 9 covers per-call tracking, caching, and cost attribution.)

1.2 The Demo-to-Production Gap

When we first deployed our agents, they worked fine for simple tasks — status checks, information retrieval, quick operations. Anything under 5-8 minutes, no problems.

Then we ran the first real automation job. The agent needed to:

1. Fetch application details from multiple APIs
2. Disable traffic at the load balancer
3. Verify traffic was actually disabled
4. Trigger a CI/CD pipeline
5. Poll the pipeline until completion (20-30+ minutes)
6. Re-enable traffic
7. Send notifications and summaries

The agent got through step 4, triggered the pipeline… and the Lambda timed out at ~8 minutes waiting for the pipeline to complete. The next scheduled invocation fired up a new Lambda. But the agent had zero memory of what happened before. It tried to start the entire workflow from scratch. The pipeline was already running. Chaos.

That was when it sank in: **Bedrock Agent has no built-in state management for long-running workflows.** No pause/resume. No checkpointing. Nothing that remembers what happened five minutes ago.

When we started this journey, AWS Agent Core did not exist. There was no managed solution for workflow state persistence or resumable agents. The documentation painted a picture of agents as stateless request-response systems. And for conversational AI, that is fine. But we were not building a chatbot – we were building autonomous infrastructure automation.

So we built our own. Custom checkpointing. S3-based state management. EventBridge as a heartbeat that triggers the Lambda every 5 minutes. Execution locking. Optimistic concurrency control. The Lambda does not run for an hour – it runs 12 times for 5 minutes each. And between each invocation, the state is persisted, checkpointed, and resumable.

That is the kind of thing this book teaches you. Not how to create an agent – but how to make one survive in the real world.

1.2.1 “Why Are We Writing So Much Code?”

The Demo-to-Production Gap has a social dimension that nobody warns you about either.

When we started building the custom state management layer – the S3 checkpoints, the EventBridge heartbeat, the execution locks – there was real pushback from the team: “Why are we writing so much code? Bedrock is supposed to handle

this. If we just use the managed service properly, development time will be way faster."

It was a fair argument on the surface, and I struggled to persuade people otherwise. The AWS marketing materials and tutorials show agents that work end-to-end with minimal code. The natural conclusion is that if you are writing thousands of lines of custom orchestration, you are doing it wrong.

The sharper version of this argument came from one of our senior engineers: "Why can't we just tell the LLM in the prompt to keep going until the job is done? Write in the instructions: *if the task isn't complete, continue from where you left off*. The model is smart enough to loop."

It sounds reasonable until you trace what actually happens. When you call `InvokeAgent`, you are making an HTTP request. That request has a 60-second timeout — not a soft timeout you can configure, but a hard limit on the API. When those 60 seconds expire, the connection dies. The LLM does not get a chance to decide whether to continue. It does not matter how clever your prompt is — you cannot instruct a language model to override an HTTP timeout. The model controls what it *thinks*. It does not control the infrastructure it runs on.

Even if you bypass `InvokeAgent` and call the foundation model directly through `InvokeModel`, your Lambda has a 15-minute maximum. Our workflows run for an hour. No amount of prompt engineering makes a Lambda run for 61 minutes. (AWS has since introduced Agent Core, which addresses some of these runtime constraints. This book does not cover Agent Core — it did not exist when we designed and deployed our system, and the architectural patterns here apply regardless of runtime.)

This is the misunderstanding at the heart of most team debates about AI agent architecture: people conflate what the model can *reason about* with what the model can *control*. The model can decide which tool to call next. It cannot keep its own process alive. That is your code's job.

We tried it their way first. We took several other use cases and attempted to build them entirely within native Bedrock Agent patterns — minimal custom code, let the managed service handle the flow. Some were simpler than our main use case. A few seemed like they would be a perfect fit.

Every one of them hit the same walls. The 60-second synchronous timeout on `InvokeAgent`. The lack of state between invocations. The inability to handle long-running processes gracefully. We tried "Return Control" (where Bedrock hands extracted parameters back to your application instead of waiting for Lambda) and built DynamoDB state tables and polling action groups — and it worked, technically. But the volume of custom code required to paper over the platform's limitations

was comparable to what we would need to just orchestrate the workflow directly.

That experience — failing with the managed approach first — was ultimately what got the team aligned. It was not my arguments that persuaded people. It was trying Bedrock's native patterns on real use cases and watching them break. Sometimes teams need to find the walls themselves before they believe the walls are there.

The lesson: if your workflow completes in under a minute and does not need memory between invocations, Bedrock Agent alone works great. The moment you need long-running processes, persistent state, or intelligent error recovery across steps — you are building custom code regardless. The question is not "managed vs. custom." It is "which parts does the managed service handle well, and where does your code take over?"

For us, the answer was: Bedrock handles reasoning, tool selection, and natural language synthesis. Our code handles everything else — state, scheduling, retries, notifications, and lifecycle management.

There is one more dimension to these debates that no technical blog post mentions: not everyone's resistance is technical.

In any team working on a system this new, architecture decisions also become territory decisions. If the agent handles error classification, what happens to the person who built the existing error classification logic? If a config-driven tool framework lets non-engineers create agent tools, what does that mean for the engineers who used to be the bottleneck for every new integration?

We had team members push for alternative approaches not because they believed the architecture was wrong, but because the proposed architecture did not include enough of their fingerprints. This is not cynical — it is human. People's sense of professional value is tied to the systems they own. When a new system threatens to make their expertise less central, the rational response is to steer the architecture toward something that preserves their role.

The solution was not to call this out in meetings. It was to make the work genuinely collaborative — pair on the hard problems, share credit visibly, make sure no one felt like the new system was being built *around* them. The four-hour architecture debates that seemed like they were about checkpointing vs. prompt loops were sometimes actually about who gets to matter in the new world. Recognizing that made us better at resolving them.

1.2.2 The Architecture Evolution: Three Agents, Then One

Our first architecture was the one that looked best on a whiteboard: a supervisor agent routing requests to three specialized sub-agents – one for infrastructure operations, one for notifications, one for error classification. It mapped cleanly to how our team was organized.

The problem was that LLMs do not think in org charts.

Each agent maintained its own view of state. When one failed mid-workflow, the others had no idea what had happened. Debugging a single request meant reading logs from three different Lambda functions. Every handoff between agents meant serializing context, and nuance got lost in translation.

We collapsed all three into one agent with nine tools, and debugging time dropped immediately. The "agents" had been just tool groupings – the LLM gained nothing from the separation. Chapter 2 covers the architectural reasoning behind this in detail, but the lesson for this chapter is simpler: do not reach for multi-agent because it looks sophisticated. Reach for it when agents genuinely reason differently. If they are calling different APIs with the same reasoning style, tools are the right abstraction.

1.3 Enterprise Constraints: The Real Challenge

Enterprise constraints are not bugs you can fix. They are just how things work in this environment, and you have to build around them.

Security: The IAM Gauntlet

In a startup, you create an IAM role with `AdministratorAccess`, deploy your agent, and move on. In an enterprise, the process looks more like this:

- Your account has **managed policies** you cannot modify. They include **explicit denies** on sensitive actions like `iam:PassRole`, `iam:CreateRole`, and `iam:AttachRolePolicy`.
- Even `AdministratorAccess` cannot override an explicit deny. AWS IAM evaluates explicit denies before allows.
- You need the cloud platform team to create a service-linked role, which requires a change request, approval from the security architecture team, and a 5-business-day SLA.

- KMS encryption is mandatory. Every Bedrock agent, every S3 bucket, every SNS topic needs a KMS key. Misconfigure the key policy and you get `AccessDeniedException` with no useful error message.

We spent three weeks on IAM alone before we could deploy our first agent. Three weeks. Not because the technology was hard, but because the governance process was slow and the error messages were useless.

Networking: Everything Private

In enterprise, nothing gets a public endpoint.

- Bedrock API? Accessed through a VPC interface endpoint.
- S3? Gateway endpoint.
- SNS? Interface endpoint.
- API Gateway? Private REST API, accessible only through a VPC endpoint with a resource policy.
- External APIs like build servers? Through a NAT gateway, or more commonly through a corporate proxy.

Getting all these network paths working simultaneously is a puzzle. Your Lambda needs to reach AWS services *and* external APIs *and* internal APIs — each through a different network path. If you misconfigure the proxy settings, the Lambda hangs silently. If you forget to add `169.254.169.254` to `NO_PROXY`, the Lambda cannot even get its IAM credentials.

We built a DNS diagnostic tool into the agent itself because network debugging in Lambda is like debugging a submarine — you cannot just SSH in and run `nslookup`.

The proxy configuration was its own circle of hell. Python's `requests` library does NOT reliably honor `NO_PROXY` from environment variables. We discovered this after hours of debugging why our Lambda could reach Jenkins (through the proxy) but could not get its own IAM credentials (which go to `169.254.169.254`, a link-local address that should bypass the proxy). We ended up implementing `NO_PROXY` handling manually in our HTTP client — a problem we never expected to have to solve ourselves. Chapter 7 has the full networking setup, the code, the DNS diagnostic script, and the list of `NO_PROXY` entries that took us weeks to compile.

Cost: The CFO's Favorite Topic

LLM pricing is measured in tokens — and tokens add up fast. A single long-running workflow can make hundreds of LLM calls across error classification, notification

formatting, decision-making, and tool selection. Without tracking, costs creep up in a predictable pattern:

1. Development: "This costs nothing! Like 2 cents per run!"
2. Staging: "Hmm, that's a bit more... maybe 50 cents per run?"
3. Production at scale: "Why is our AWS bill $20,000 higher this month?"

We built per-call and per-session token tracking into our LLM client from week two. Every call logs input tokens, output tokens, estimated cost, and a prompt hash for attribution. We can tell you exactly which prompt template costs the most and which workflows are the most expensive.

Here is the math that catches people off guard. A single infrastructure refresh – rotating the OS image on an EC2 instance, an EKS node group, or an MQ broker – can run for up to 288 iterations (24 hours at 5-minute intervals). Each iteration might make 1–3 LLM calls. At worst case:

- 864 LLM calls per workflow
- ~864K input tokens, ~173K output tokens
- Estimated cost: ~**$5.19 per workflow**

With 20 concurrent workflows, that is potentially $103.80/day at maximum. In practice, most workflows complete in 30–60 minutes (6–12 iterations), bringing the realistic cost to $0.50–$1.00 per workflow. But without tracking, you do not know whether you are in the $0.50 case or the $5.19 case until the bill arrives.

Compliance and Audit

Every decision the agent makes is a potential audit finding. When the agent decides to restart a service at 3 AM, someone will ask: "Why? What information did it base that decision on? What alternatives did it consider?" You need:

- Structured logging with decision traces
- State archives (we archive every completed workflow to S3 for later review)
- Guardrails at the Bedrock API level (content filtering, not just prompt engineering)
- Human-in-the-loop approval for high-risk actions in production

Change Management: When Your Biggest Blocker Is not Technical

There is an enterprise constraint that technical books rarely mention: the organizational machinery itself.

Deploying a new IAM role requires a change request. The change request needs approval from a security architect. The security architect has a 5-business-day SLA. But they often have questions, which resets the clock. We had one IAM policy change — a single line adding `bedrock:InvokeAgent` to an allow list — that took 11 business days from request to deployment.

Model access is another one. Bedrock requires you to explicitly request access to each foundation model. In enterprise, that request goes through a procurement workflow because it involves a pricing commitment. Claude 3.5 Sonnet access took two weeks to provision across our AWS accounts (development, staging, and production environments).

And then there are the meetings. The architecture review board wants to understand why an AI agent is making infrastructure decisions. The security team wants a threat model. The compliance team wants to know how you handle PII in prompts. The cost governance board wants projected spend at scale. Each of these is a legitimate concern, and each requires preparation, documentation, and a 30-60 minute meeting.

None of this is unreasonable. It is how large organizations manage risk. But if you are coming from a startup background or expecting the same velocity as a personal project, the pace will shock you. Budget 4-6 weeks for governance and access provisioning before you write a single line of production code.

One piece of advice I wish I had internalized earlier: **never escalate just because it is taking time.** The IAM team, the security architects, the networking engineers, the cloud platform team — these people are not obstacles. They are the people who will save you when something goes wrong in production at 2 AM. When your agent accidentally disables traffic on the wrong load balancer, the networking team is who you are calling. When your KMS key rotation breaks your Bedrock integration on a Friday night, the security team is who knows the fix.

Build relationships with these teams early. Buy them coffee. Sit with them and explain what you are building. Ask for their input on the architecture before you submit the change request, not after. The engineers who review your IAM policies understand permission boundaries better than you do — their feedback genuinely improves your design. We caught two over-permissive policies and one missing encryption requirement during security review that would have become production incidents.

The teams that slow you down during setup are the same teams that speed you

up during incidents. Treat the governance process as relationship-building, not red tape. Chapter 6 covers the IAM gauntlet in detail, and Appendix D has IAM policy templates that survived our security review process.

1.4 What "Enterprise-Ready" Actually Means

After a year of doing this, I have developed a working definition of "enterprise-ready" for an AI agent. It is not a marketing buzzword — it is a set of requirements that your security and platform teams will actually enforce:

Security:

- [ ] IAM roles follow least-privilege principle with explicit resource ARNs
- [ ] KMS encryption on all data at rest (S3, SNS, Bedrock)
- [ ] No secrets in environment variables — use Secrets Manager
- [ ] Guardrails applied at the API level, not just in prompts
- [ ] Audit trail for every agent decision

Networking:

- [ ] All AWS services accessed through VPC endpoints
- [ ] No public internet access required for core functionality
- [ ] Proxy configuration for external services with `NO_PROXY` bypass
- [ ] Private API Gateway for agent-to-agent communication
- [ ] DNS diagnostics built into the agent

Reliability:

- [ ] State management with checkpointing for long-running workflows
- [ ] Execution locking to prevent concurrent processing conflicts
- [ ] Circuit breakers with per-step retry limits
- [ ] Graceful degradation when dependencies fail
- [ ] MAX_ITERATIONS guard to prevent runaway workflows

Observability:

- [ ] Structured logging with phase tracking (OBSERVE/ACT/REFLECT/REASON)
- [ ] Per-call and per-session token/cost tracking
- [ ] Tool execution timing and correlation IDs
- [ ] Health check endpoints

- [ ] CloudWatch Insights queries for operational analysis

Cost:

- [ ] Token tracking at per-call and session level
- [ ] Configurable cost rates per model (for model switching)
- [ ] Concurrency limits to cap maximum spend
- [ ] Prompt optimization for input token reduction
- [ ] Cost alerting thresholds

If your agent cannot check every box on this list, it is not enterprise-ready. It is a demo.

1.5 The Roadmap: What This Book Covers

This book follows the actual path we walked — from initial architecture through production deployment. Each chapter addresses a real problem we hit and how we solved it.

Part 1: The Landscape

(Chapters 1–2) — Understanding the playing field. Chapter 2 tears apart the Bedrock Agent architecture: how tokenization works, what the ReAct loop actually does under the hood, how context caching saves (and does not save) money, and when to use `invoke_agent` vs. `invoke_model`. By the end you will know exactly where Bedrock's responsibilities end and yours begin.

Part 2: Building the Agent

(Chapters 3–5) — The craft of making agents useful. Chapter 3 covers prompt engineering with a real 800-line production instruction set annotated line by line. Chapter 4 goes deep on action groups and tool design — including the Mini-MCP pattern that let SREs create new agents without writing Lambda functions. Chapter 5 covers data architecture: Cassandra modeling for real-time workflows, S3 state management with optimistic locking, and the checkpoint pattern that makes everything resumable.

Part 3: The Hard Parts

(Chapters 6–9) — IAM policies that take weeks to approve (Chapter 6). Networking that breaks silently — VPC endpoints, proxy configuration, DNS resolution in Lambda (Chapter 7). Deployment automation that evolved through three generations: console clicks, CLI scripts, CloudFormation with CI/CD (Chapter 8). Cost engineering with per-call token tracking, caching strategies, and the math that

turns a $5 workflow into a $0.50 one (Chapter 9). Most books stop before this part; this one does not.

Part 4: Production

(Chapters 10–13) — Testing non-deterministic systems with golden datasets and LLM-as-a-Judge evaluation pipelines (Chapter 10). Observability with structured logging, OBSERVE/ACT/REFLECT/REASON phase tracking, and CloudWatch Insights queries (Chapter 11). The production readiness checklist that our security and platform teams actually enforced (Chapter 12). And finally, every lesson learned the hard way — including the three architectural bets that paid off and the two that did not (Chapter 13).

Every chapter contains things only someone who actually built this could know. Real CloudFormation templates, not pseudo-code. Real error messages with real fixes. Real cost numbers from production workloads. Real war stories about what broke at 3 AM and how we fixed it.

This is the book I wished someone had handed me before I started this project.

1.6 Who This Book Is For

This book assumes you are a competent engineer. You know Python. You have used AWS. You have deployed services to production before. You do not need a tutorial on what Lambda is or how S3 works.

What you may not know is how to build AI agents that survive enterprise reality. If any of these describe you, this book is written for you:

- **Senior engineers and architects** tasked with building AI agents on AWS Bedrock. You have read the docs, maybe built a demo, and now you need to ship something real.
- **Engineering managers** evaluating whether Bedrock Agents are viable for your team's use cases. You need to know the real effort, cost, and timeline — not the marketing pitch.
- **Operations and platform teams** responsible for productionizing AI agents that someone else prototyped. You inherited a POC and need to make it production-grade.
- **Cloud architects** designing AI infrastructure for enterprises with serious security, networking, and compliance requirements.

If you are looking for an introduction to AI or machine learning, this is not the right book. If you are building a chatbot for a personal project, this is probably

overkill. But if you need to ship an AI agent inside an enterprise — with all the IAM, VPC, KMS, compliance, and cost constraints that implies — this is the book.