</> FOR DEVELOPERS

Build an AI-Powered Django App

From idea to production: LLMs, RAG, streaming, and shipping a real AI product with Django.

BUILT WITH

- Django
- Python
- PostgreSQL + pgvector
- Claude API
- Celery
- HTMX

RAG & Embeddings | Streaming Responses | Guardrails & Safety | Cost Control

TECHNOVIZE PUBLISHING

</> djangozen.com

Build an AI-Powered Django App

Build an AI-Powered Django App

From idea to production: LLMs, RAG, streaming, and shipping a real AI product with Django

KC Ramo

Technovize Publishing
djangozen.com

Build an AI-Powered Django App

From idea to production: LLMs, RAG, streaming, and shipping a real AI product with Django

Contents

PREFACE

Preface

Adding "AI" to an application is easy to demo and hard to ship. A weekend project that calls a language model and prints its reply is impressive in a screenshot and useless in production, because the gap between that demo and a real product is enormous: grounding the model in your own data so it stops making things up, streaming responses so the app feels alive, defending against prompt injection, controlling cost before it controls you, evaluating quality when there is no single correct answer, and operating all of it with your eyes open. This book is about crossing that gap. It builds one real AI product, end to end, on Django — and every chapter adds a layer that a demo skips and a product cannot.

Who should read this book

This book is for Django developers who want to build a genuine AI-powered product. If you are comfortable with Django — models, views, templates, the ORM, migrations — and you have used, or are willing to use, a large language model API, you have what you need. You do not need a machine-learning background; we treat models as powerful, fallible tools you call over an API, not as things you train. What you will gain is the ability to build an AI feature that is grounded, safe, affordable, testable, and operable — the difference between a party trick and software people rely on.

What you will build

Throughout the book we build **DocuMind**, an AI document assistant: users upload their documents and then ask questions and get answers grounded in those documents, with citations. It is a deliberately representative product, because the machinery behind it — calling a model, engineering prompts, retrieving relevant context, streaming answers, guarding against abuse, controlling cost, evaluating quality, and shipping it all — is the same machinery behind almost every serious AI application, whether it is a support assistant, an internal knowledge tool, or a research aide. Learn to build DocuMind well and you can build your own AI product from the same parts.

How the book is organized

The book follows the order you would actually build in, grouped into five parts. **Part I, Foundations**, sets up the project and gives you a working developer's model of how language models behave. **Part II, Talking to the Model**, covers calling the LLM API from Django, engineering prompts for production, and streaming responses. **Part III, Grounding the AI in Your Data**, is the heart of the book: embeddings and vector search with pgvector, a full Retrieval-Augmented Generation pipeline, and ingesting documents. **Part IV, Making It Production-Grade**, adds guardrails and safety, cost control and rate limiting, and evaluation. **Part V, Shipping & Operating**, covers observability, security and privacy, deployment and scaling, and launch. Each chapter ends with a summary and key takeaways, and the appendices gather the checklists and references you will return to.

Conventions and a note on non-determinism

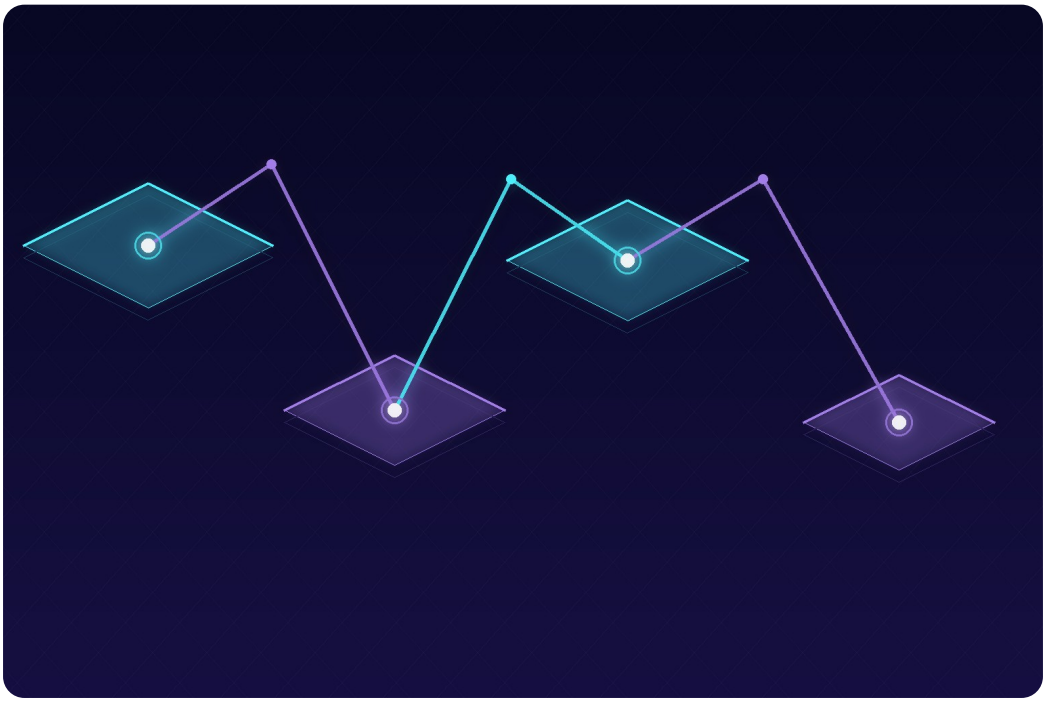
Code appears in monospaced blocks, shown where it lives, with imports trimmed once a pattern is established. Filenames are given relative to the project root. The recurring principle is that **production-ready means being honest about trade-offs** — where a decision matters, the book weighs the alternatives so you can choose differently when your context differs. One trait shapes everything in this book and deserves flagging now: language models are **non-deterministic**. The same input can produce different outputs, and no output is guaranteed correct. That single fact is why AI applications need grounding, guardrails, evaluation, and observability in ways ordinary software does not — and it is the thread that runs through every chapter that follows.

A note on models and change

The AI field moves quickly. Specific model names, prices, and API details will change after this book is written, and that is fine: the patterns here — grounding with RAG, engineering and versioning prompts, streaming, guardrails, cost control, and evaluation — are durable, and swapping in a newer or cheaper model is exactly the kind of routine change the practices in this book are designed to absorb. Learn the patterns, and the details will follow. With that, let us begin by getting clear on what we are building and how the pieces fit together.

PART I

Foundations



CHAPTER 1

Introduction

There is a peculiar gap between the demo that impresses your team on a Friday afternoon and the system that survives a Monday morning in production. The demo streams a confident answer to a well-behaved question. The production system faces a malformed PDF, a user in a different timezone asking something the documents never covered, a Claude API timeout at the worst possible moment, and a finance team that wants to know why last month's bill was three times larger than the month before. This book is about closing that gap. We are going to build one real application, from an empty directory to a hardened deployment, and we are going to be honest at every step about what works, what breaks, and what it costs.

The application is called **DocuMind**. It is an AI document assistant: users upload their documents, and then they chat with those documents in natural language. Ask "What was our refund policy in the 2023 contract?" and DocuMind answers using the actual text of the actual contract, with citations pointing back to the source, streaming the answer token by token the way you would expect a modern AI product to behave. Under the hood it is a **Retrieval-Augmented Generation** (RAG) pipeline built on Django 5.2, PostgreSQL with the pgvector extension, Redis, Celery, and the Anthropic Claude API. It is not a toy. By the time you finish this book, DocuMind will handle real documents, scope every byte of data to the user or organization that owns it, degrade gracefully when the model provider has a bad day, measure its own answer quality, and tell you exactly how much each conversation cost.

What DocuMind Is, and What It Will Do by the End

Let us be concrete about the finished product, because a clear destination makes the journey legible. When you have worked through all five parts of this book, DocuMind will do the following.

A user signs up and lands in an organization-scoped workspace. They upload a document: a PDF contract, a Markdown runbook, a plain-text policy, a Word file exported from someone's laptop. DocuMind ingests it asynchronously. That ingestion is not a single step; it is a pipeline. The file is parsed into text, the text is split into overlapping **chunks** sized to balance

retrieval precision against context, each chunk is turned into an **embedding** (a vector of floating-point numbers that captures its meaning), and those vectors are stored in PostgreSQL via pgvector alongside the chunk text and metadata. The user watches progress update live because the work runs in a Celery worker and the UI polls or receives server-sent events. When ingestion finishes, the document is ready to query.

Now the user asks a question. DocuMind embeds the question, performs a similarity search against the stored vectors to retrieve the handful of chunks most relevant to what was asked, assembles those chunks into a carefully structured prompt, and sends that prompt to Claude. The answer streams back into the browser over SSE, rendered progressively with HTMX and Alpine.js, no heavyweight frontend framework required. The answer includes citations: "According to Section 4.2 of contract-2023.pdf..." because we pass the source chunks to the model and instruct it to ground its response in them and to say so when it cannot. Every request is logged with its token counts and estimated cost. Answers are checked against a small evaluation suite so that when we change the prompt or swap the model, we can tell whether we made things better or worse. Abusive input is rate-limited and guarded. Long documents do not blow up memory. A provider outage returns a useful error instead of a stack trace.

That is the whole arc: **upload, ingest, retrieve, generate, stream, ground, guard, measure, deploy**. Each of those verbs is a chapter's worth of decisions, and most of those decisions involve a trade-off we will name out loud rather than paper over.

The Anatomy of an AI Application

Before we write a line of code, it helps to have a shared mental model of what an AI application actually is. A surprising number of teams fail not because they cannot call an API, but because they think the API call *is* the application. It is not. The model call is one organ in a body of many. Here is the anatomy, organ by organ.

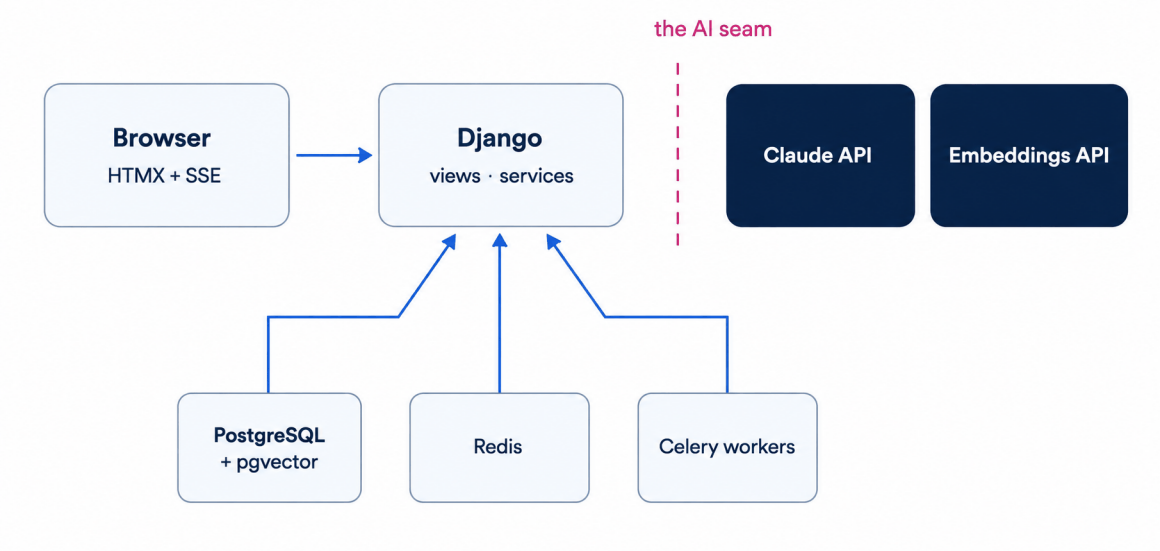


Figure 1.1 — The anatomy of an AI application: ordinary Django on one side of the seam, metered non-determinism on the other.

The LLM. At the center is a large language model, in our case Claude, accessed through the Anthropic Messages API. The model is a function that takes a sequence of messages and produces the next message. It is stateless between calls: it remembers nothing unless you send it. It is probabilistic: the same input can yield different outputs. And it is a paid external dependency with latency, rate limits, and occasional failures, which means from an engineering standpoint you should treat it much the way you treat a payment gateway or an email provider, not the way you treat a local function call.

Prompting. The model does what the prompt tells it to do, and getting the prompt right is real engineering, not incantation. A prompt has structure: a system prompt that sets role, constraints, and output format; the retrieved context; the conversation history; and the user's current question. Small changes in wording change behavior, which is why we will treat prompts as versioned artifacts that live in code and are tested, not as strings we tweak in production and hope.

Grounding and RAG. A raw LLM answers from its training, which means it can be confidently wrong about your documents, which it has never seen. **Grounding** is the practice of supplying the model with the specific facts it should reason over, and **Retrieval-Augmented Generation** is the dominant technique for doing that at scale: retrieve the relevant pieces of your data, put them in the prompt, and instruct the model to answer only from them. RAG is the backbone of DocuMind. It is also where most of the subtle failures live, and we will spend real time on chunking strategy, embedding choice, retrieval quality, and what to do when retrieval returns nothing useful.

Streaming UX. Nobody wants to stare at a spinner for eight seconds while a paragraph generates. Streaming the response as it is produced transforms perceived performance and is now table stakes for AI products. But streaming complicates everything downstream: your view can no longer return a tidy JSON blob, your error handling must cope with a failure halfway through a response, and your infrastructure must keep a connection open. We will implement streaming with server-sent events, which fit Django's request model far more comfortably than WebSockets for this one-directional case.

Guardrails. Users will paste in things you did not anticipate. They will try to make the assistant ignore its instructions, generate abuse, or run up your bill. Guardrails are the input validation, rate limiting, prompt-injection defenses, output filtering, and scope enforcement that keep the system safe and affordable. In an AI app, guardrails are not a nice-to-have bolted on at the end; they are load-bearing, because the core component is a system that will, by design, try to be helpful with whatever you hand it.

Cost. Every model call costs money, priced per token of input and output. A careless RAG design that stuffs twenty chunks into every prompt, or a chat feature that resends the entire conversation history on each turn, can multiply your bill without improving answers. Cost is an engineering constraint on the same footing as latency and correctness, and we will instrument it from the beginning so that it is a number you watch, not a surprise you receive.

Evaluation. How do you know your AI app is good? Not by vibes. **Evaluation** is the discipline of measuring answer quality against a curated set of questions with known-good expectations, so that a prompt change or model swap produces a number that went up or down rather than a feeling. Traditional software has unit tests; AI software needs those too, plus evals, because the interesting behavior is not deterministic and cannot be pinned by an assertion of exact equality.

Observability. When a user reports "it gave me a wrong answer," you need to reconstruct what happened: what they asked, what was retrieved, what prompt was assembled, what the model returned, how long it took, what it cost. Observability for AI apps means logging the whole chain, not just HTTP status codes. Without it you are debugging a probabilistic system blind, which is a special kind of misery we will help you avoid.

Hold this anatomy in mind. Every chapter of this book is, in effect, developing one of these organs and wiring it to the others. The application only becomes production-ready when all of them are present and cooperating.

Why Build It on Django

You could build an AI assistant in a dozen stacks. Plenty of tutorials reach for a thin Python script wrapped in a minimal async framework, or a notebook that never becomes a product. We are deliberately choosing Django, and the reasons are practical rather than tribal.

First, an AI product is still a *product*, and products need the unglamorous machinery Django provides out of the box: authentication, sessions, an admin interface, an ORM with migrations, forms and validation, CSRF protection, a mature security posture, and a permissions system. RAG is the interesting 20 percent; the other 80 percent is a web application, and Django has spent two decades making that 80 percent boring in the best sense.

Second, DocuMind is fundamentally about **data that belongs to someone**. Documents are private. Chats are private. Getting multi-tenant data isolation right, so that organization A can never retrieve organization B's chunks, is a first-class concern, and Django's ORM plus a disciplined query layer gives us the tools to enforce that scoping rigorously and to test it.

Third, PostgreSQL is a first-class citizen in the Django world, and PostgreSQL with the **pgvector** extension lets us store embeddings and run similarity search in the same database that holds our users, documents, and chats. That is a genuine architectural simplification: no separate vector database to operate, back up, secure, and keep consistent with your relational data. For a great many production workloads, "just use Postgres" is the correct and honest answer, and we will discuss where that stops being true.

Fourth, Django's ecosystem includes Celery for background work and integrates cleanly with Redis, which we need because document ingestion and long-running AI calls do not belong in a synchronous request-response cycle. The moment you have work that takes longer than a user is willing to wait, you need a task queue, and this is a well-trodden path in Django.

The design philosophy we bring to that Django code matters as much as the framework choice. We follow a **service layer** architecture: business logic lives in `services.py` modules, reads go through **selectors**, and views and models stay thin. This is not ceremony. When your application's core is a multi-step pipeline calling an external, fallible, expensive API, you want that logic in plain, testable functions that do not depend on the request cycle, not smeared across fat views and model methods. A service function can be called from a view, a Celery task, a management command, or a test with equal ease. This discipline pays for itself the first time you need to reuse ingestion logic in a batch importer.

The Technology Stack and Why Each Piece

Here is the full stack, with a one-line justification for each choice, because a stack you cannot defend is a stack you will regret.

- **Python 3.12+** — the current stable Python, with the performance and typing improvements we want; the AI ecosystem lives in Python, so this is barely a choice.
- **Django 5.2 LTS** — a long-term-support release giving us a stable, secure foundation and the batteries-included machinery described above.
- **PostgreSQL + pgvector** — one database for relational data and vector search, cutting operational complexity and keeping embeddings transactionally consistent with the rows they describe.
- **Redis** — the message broker for Celery and a fast cache; used for rate limiting, session storage options, and transient state.
- **Celery** — the task queue that moves document ingestion and other slow work out of the request cycle so users are never left waiting on a synchronous HTTP request that might time out.
- **Anthropic Claude API (Messages API)** — our large language model, accessed through a well-documented API with streaming support and current models spanning the Claude Sonnet and Opus families, letting us trade cost against capability per use case.
- **HTMX + Alpine.js** — server-rendered HTML with just enough client-side interactivity for streaming and dynamic updates, avoiding the weight and build complexity of a single-page-application framework we do not need.
- **Server-sent events (SSE)** — the transport for streaming model output to the browser: one-directional, HTTP-native, and a far simpler fit than WebSockets for our streaming case.
- **Docker, Gunicorn, Nginx** — the deployment triad: reproducible containers, a production WSGI server, and a battle-tested reverse proxy handling TLS, static files, and buffering concerns that matter for streaming.

Notice what is *not* here. There is no bespoke vector database, no orchestration framework wrapping the LLM in layers of abstraction, no frontend build pipeline. Every one of those can be justified in some context, and we will note where they earn their place. But a core theme of this book is that production readiness comes from understanding your system, not from

accumulating dependencies, and the smaller the stack you fully understand, the more reliably you can operate it at 3 a.m.

What Makes an AI App "Production" Versus a Demo

This distinction is the spine of the book, so let us draw it sharply. A demo and a production system can produce identical output for the same input on a good day. They diverge entirely on the bad days, and bad days are the norm at scale. Here is what separates them.

A demo **trusts its inputs**; production **validates and sanitizes** everything, because a fraction of your uploads will be corrupt, enormous, encrypted, or hostile. A demo **assumes the model call succeeds**; production has timeouts, retries with backoff, and a coherent fallback when Claude is slow or unreachable. A demo **runs everything in the request**; production pushes slow work to Celery so a thirty-second ingestion does not tie up a web worker or hit a gateway timeout. A demo **ignores cost**; production meters tokens and knows the unit economics of a conversation. A demo **has no concept of who owns what**; production scopes every query so that data isolation is provably enforced and tested. A demo **cannot tell you if a change helped**; production has evaluations that turn "seems better" into a measured delta. A demo **logs nothing useful**; production records the full chain of a request so failures are diagnosable. A demo **hard-codes secrets and runs the dev server**; production manages configuration and secrets properly and runs behind Gunicorn and Nginx with sensible security headers.

None of these are exotic. They are the difference between a system that demos and a system that *runs*. And crucially, most of them are cheap to build in from the start and expensive to retrofit, which is exactly why we build DocuMind the right way from Chapter 1 rather than promising to "productionize it later." Later never comes; later is a myth that ships technical debt.

A Note on Non-Determinism, and How It Changes Engineering

This deserves its own section because it is the single biggest shift in mindset for a developer coming from conventional software. The systems you have built until now are, at least in aspiration, **deterministic**: the same input yields the same output, and you write a test asserting exactly that. A large language model breaks this contract on purpose. The same prompt can produce different wording, different emphasis, occasionally different substance, from one call to the next. This is not a bug to be fixed; it is the nature of the tool.

Non-determinism ripples through your engineering practice in ways worth naming.

Your **tests** change shape. You cannot assert that the model returned an exact string. Instead you assert on structure ("the response is valid JSON with these keys"), on properties ("the answer cites at least one source"), and on behavior of the deterministic parts of your pipeline (retrieval returns the correct chunks for a known query). For the model's actual output quality, you use evaluations that score against expectations with tolerance, run over a set of cases, and report an aggregate you can track over time.

Your **debugging** changes. "It worked on my machine" and "I can't reproduce it" take on new meaning when the same input can behave differently. This is precisely why observability is not optional: you capture what actually happened on the failing request because you may not be able to summon it again on demand.

Your **error handling** changes. Beyond the usual network and HTTP failures, you must handle the model doing something valid-but-unwanted: returning malformed output when you asked for structured data, refusing a benign request, hallucinating a citation. You defend against these with careful prompting, output validation, and graceful degradation rather than assuming the response is always well-formed.

Your **product expectations** change, and so must your users'. A grounded RAG system dramatically reduces hallucination but does not eliminate it, and honest product design surfaces uncertainty, shows sources, and makes it easy for users to verify rather than pretending the machine is an oracle. Being honest about this in the UI is itself a production concern.

The engineering response to non-determinism is not to fight it but to build a deterministic *scaffold* around a non-deterministic *core*. Retrieval, prompt assembly, scoping, logging, cost accounting, and guardrails are all deterministic and testable. We make those rock-solid, and we treat the model's output as the one place where we design for variance. That framing, deterministic edges around a probabilistic center, will guide nearly every architectural decision in this book.

Who This Book Is For

This book is written for the working Django developer who wants to build a real AI product and is tired of tutorials that stop at a "hello world" chat box. You should be comfortable with Django's fundamentals: models, views, templates, migrations, the ORM, and how a request

flows through the framework. You should be able to read and write Python confidently and be at ease on the command line. You do not need a background in machine learning, and you certainly do not need to train models; we treat the LLM as a powerful external service and focus on the engineering of building a dependable product around it.

You will get the most from this book if you have felt the specific frustration of getting an AI prototype working and then having no idea how to make it safe, affordable, testable, and deployable. That gap, between prototype and product, is exactly what we fill. If you are a technical lead evaluating how to structure an AI initiative, the architecture and trade-off discussions will serve you even if you delegate the typing. If you are a solo founder building your first AI SaaS, DocuMind is close enough to a real product that you can adapt it directly.

If you have never used Django before, start with the official Django tutorial first; we move too fast on the framework basics to serve as your introduction to them. And if you want a gentle, uncritical tour that never mentions cost or failure, this is the wrong book, cheerfully so. Our entire premise is that production readiness means being honest about trade-offs, and honesty sometimes reads as bad news.

How the Book Is Organized

The book is divided into five parts, each building directly on the last. We do not present disconnected recipes; every chapter leaves DocuMind in a working, if incomplete, state, and the next chapter picks up exactly there.

Part One — Foundations. We establish the ground: setting up the Django project with a clean, service-oriented structure; provisioning PostgreSQL with pgvector, Redis, and Celery; modeling documents, chunks, chats, and messages with correct multi-tenant scoping; and building the authentication and organization boundaries that everything else depends on. By the end of Part One you have a real Django application with no AI in it yet, and that is deliberate: the boring foundation is what makes the interesting parts reliable.

Part Two — The RAG Pipeline. Here we build the ingestion side: parsing documents, chunking text with strategies that actually affect answer quality, generating embeddings, and storing vectors in pgvector. Then we build retrieval: similarity search, ranking, and the crucial question of what to do when retrieval is weak or empty. This is the heart of DocuMind and the part most tutorials get superficially wrong.

Part Three — Generation and the Model. We integrate the Claude Messages API in earnest: assembling grounded prompts from retrieved context, versioning and testing those prompts, calling the model through a clean service interface, handling its failures, and streaming responses to the browser over SSE with HTMX and Alpine.js. This is where DocuMind starts to feel like a product.

Part Four — Production Concerns. Now we make it safe and sustainable: guardrails against prompt injection and abuse, rate limiting, input and output validation, cost instrumentation and control, the evaluation suite that measures answer quality, and the observability that lets you debug a probabilistic system. This part is the difference between the demo and the real thing, and it is where the book's title earns its keep.

Part Five — Deployment and Operations. Finally we ship: Dockerizing the application and its workers, running under Gunicorn behind Nginx with attention to the special needs of streaming, managing configuration and secrets, database migrations and backups, and the operational practices that keep an AI product healthy after launch. We finish with a candid look at what to monitor and how to keep costs and quality under control over time.

You can read straight through, and we recommend it, because the value compounds. You can also treat Parts Two through Four as a reference for their respective concerns once you understand the whole. What you should not do is skip Part One and expect the later parts to make sense; the architecture we establish early is load-bearing throughout.

A Preview of the Code Style

So that the later chapters feel familiar, here is a taste of the service-layer style we will use throughout. Business logic lives in functions that take plain arguments and can be called from anywhere:

```
# documind/documents/services.py

def ingest_document(*, document_id: int) -> None:
    """Parse, chunk, embed, and store a document. Runs in a Celery task."""
    document = _get_document_for_ingestion(document_id)
    text = extract_text(document.file)
    chunks = split_into_chunks(text, size=800, overlap=100)
    embeddings = embed_chunks(chunks)
    store_chunks(document=document, chunks=chunks, embeddings=embeddings)
    document.mark_ready()
```

Notice that the function is keyword-only, returns nothing surprising, and reads as a sequence of named steps. Each of those steps is itself a testable function. Reads, by contrast, go through selectors that never mutate state:

```
# documind/documents/selectors.py

def get_relevant_chunks(*, organization, query_embedding, limit: int = 6):
    """Return the most similar chunks for this organization only."""
    return (
        Chunk.objects
        .filter(document__organization=organization)  # scoping is not optional
        .order_by(CosineDistance("embedding", query_embedding))[:limit]
    )
```

That single `.filter(document__organization=organization)` is a production concern hiding in plain sight: it is what stops one tenant from ever retrieving another's data, and we will make sure it is enforced and tested everywhere it matters. Small lines carry large responsibilities in this kind of system, and part of the craft is learning to see them.

A Tour of the Finished Experience

It is worth walking the finished product end to end, from a real user's chair, because the whole book is in service of this one flow and it helps to picture where we are going before we start digging foundations.

A new user arrives and **signs up**. They are not dropped into a shared free-for-all; they land in an organization-scoped workspace, and everything they do from this moment is invisibly tagged with that boundary. This is the quiet decision that makes DocuMind a multi-tenant product rather than a personal script, and it is why authentication and organization modeling come before any AI in the build. The user sees an empty document library and an invitation to add something.

They **upload a document**: a forty-page PDF services contract exported from a legal team's laptop. DocuMind accepts the file, validates that it is a type and size it can handle, stores it, and hands the slow work to a Celery worker rather than making the user wait on a spinning request. The library row shows a live status: *parsing*, then *chunking*, then *embedding*, then *ready*. Behind that modest progress bar, the ingestion pipeline has extracted the text, split it into overlapping chunks, turned each chunk into an embedding, and written those vectors into pgvector alongside their source text and metadata. The user does not see any of that

machinery, and that is the point: production systems hide their complexity behind a status they can trust.

Now the user **asks a question** in plain language: "Does this contract auto-renew, and how much notice do I need to give to cancel?" DocuMind embeds the question, retrieves the handful of chunks most relevant to renewal and notice terms, assembles them into a grounded prompt, and streams Claude's answer back into the browser token by token. Within a second the first words appear, and the answer arrives as a short, direct paragraph: the contract auto-renews for successive one-year terms unless either party gives *sixty days*' written notice before the renewal date. Crucially, the answer carries a **citation**: "According to Section 9.2 of services-agreement.pdf." The user clicks it and sees the exact source passage, so they are verifying a claim, not trusting an oracle.

Then the user asks something the document simply does not cover: "What is the penalty for late payment?" There is no such clause. A demo would confidently invent one. DocuMind, grounded and instructed to admit ignorance, says it could not find anything about late-payment penalties in this document and suggests the user check the original. That honest "I don't know" is not a failure of the product; it is one of its most important features, and it is the direct result of engineering we will do deliberately. Meanwhile, invisible to the user, every one of these turns was logged with its retrieved chunks, assembled prompt, token counts, latency, and cost, so that if anything looked wrong we could reconstruct exactly what happened.

What "Done" Means for an AI Feature

In conventional software, a feature is roughly done when it produces the right output for valid input and handles the obvious errors. AI features raise the bar, and it is worth being explicit about the bar so you know when you have actually cleared it rather than merely reached a working happy path.

Consider the single feature "answer a question about an uploaded document." The happy path, question in, grounded answer out, is perhaps a fifth of the work. A feature like this is **done** when all of the following are true:

- **Grounding.** The answer is drawn from the user's actual documents, not the model's training, and the system retrieves the right context to make that possible.

- **Honesty.** When the documents do not contain the answer, the feature says so instead of fabricating one, and it exposes sources so claims are verifiable.
- **Guardrails.** Hostile input, prompt-injection attempts buried in an uploaded file, and abusive volume are handled without compromising other tenants or the bill.
- **Cost.** The unit economics of an answer are known and bounded; a single question cannot silently trigger a runaway sequence of expensive calls.
- **Evaluation.** There is a way to measure whether answers are good, so a future prompt tweak produces a number that moved rather than a vibe that shifted.
- **Observability.** When a user reports a bad answer, you can reconstruct the entire chain, what was asked, retrieved, prompted, and returned, after the fact.
- **Failure handling.** A model timeout, a rate limit, or a malformed response degrades into a useful message, not a stack trace or a hung request.

None of these is optional in a system people rely on, and here is the uncomfortable part: the happy path is the cheap fifth, and the remaining four-fifths are what the word "production" in this book's title actually refers to. A team that calls the feature done when the happy path works has not built four-fifths less product; they have built a demo that will fail on contact with real users. We define done to include the whole list from the outset, because every item on it is far cheaper to build in than to retrofit once the shape of the code has hardened around its absence.

Common Misconceptions About Building With LLMs

A handful of tempting but wrong beliefs cause more failed AI projects than any technical difficulty. They are worth naming and dismantling now, before they quietly shape decisions you will regret.

"A demo is a product." The most expensive misconception of all. A working demo proves the happy path exists; it says nothing about the corrupt uploads, hostile inputs, provider outages, cost overruns, and data-isolation requirements that constitute most of the actual engineering. The distance from demo to product is not a final polish step, it is the majority of the work, and pretending otherwise is how teams promise a two-week ship date on a two-month system.

"The model already knows my data." It does not. Claude was trained on a broad slice of public text with a knowledge cutoff, and it has never seen your contracts, your runbooks, or last week's policy update. Ask it about your documents directly and it will answer from general knowledge, confidently and often wrongly. Everything DocuMind does to be useful, the entire RAG pipeline, exists precisely because the model does *not* know your data and must be handed the relevant facts at question time. Internalizing this reframes retrieval from a nice-to-have into the load-bearing core of the product.

"You can't really test AI software." You can; you just cannot test it the way you test a pure function. The deterministic majority of the system, retrieval, scoping, prompt assembly, cost accounting, guardrails, is testable with ordinary assertions, and you should test it ruthlessly. The non-deterministic model output is testable too, through evaluations that score answers against expectations with tolerance across a set of cases and report an aggregate you can track. "It's non-deterministic so I can't test it" is an excuse, not a fact, and it is usually a prelude to shipping something no one can safely change.

"You need to fine-tune a model." Almost certainly not, and reaching for it first is a classic beginner's detour. Fine-tuning is expensive, slow to iterate, and solves problems, style adaptation, narrow classification, that DocuMind does not have. For a document-question-answering product, retrieval plus good prompting gets you the overwhelming majority of the value with a fraction of the complexity, and it updates the instant a user uploads a new file, something fine-tuning could never do. We build the RAG approach that is correct for this problem and note honestly the rare cases where fine-tuning would earn its keep.

"More context is always better." Tempting, because the models keep advertising larger context windows, but wrong in practice. Stuffing twenty marginally relevant chunks into every prompt dilutes the signal, can degrade answer quality, raises latency, and multiplies cost on every single call. Good retrieval is about relevance and restraint, not volume, and a disciplined RAG design that sends six well-chosen chunks routinely beats a lazy one that sends fifty.

The Build Order, and Why Each Part Comes When It Does

The sequence in which we build DocuMind is not arbitrary, and understanding the reasoning makes the whole book cohere rather than feel like a march through disconnected topics. The ordering follows a single principle: **each layer depends on the correctness of the one**

beneath it, so we make the foundation trustworthy before we build anything interesting on top.

We start with **foundations and data modeling**, and deliberately with no AI at all, because the multi-tenant boundary is the thing every later feature silently relies on. If organization scoping is wrong, then retrieval leaks one tenant's documents to another, and no amount of clever prompting fixes a data-isolation bug. You cannot bolt correct scoping onto a pipeline that was written without it; the filter has to be present in the very first query. So the boring part comes first precisely because it is load-bearing.

We build **ingestion and retrieval** next, before touching the model, because generation is only as good as what you feed it. Grounding is the whole game, and grounding is retrieval. A brilliant prompt over bad chunks produces confident nonsense, so we make retrieval correct and measurable before we let it drive an expensive model call. Building generation first would mean tuning prompts against a moving target, unable to tell whether a poor answer came from the prompt or from the retrieval feeding it.

Only then do we add **generation and streaming**, wiring the retrieved context into grounded prompts and the Claude Messages API. By this point the deterministic scaffold, scoping, ingestion, retrieval, is solid, so when an answer is wrong we can trust that the problem lives in the one genuinely variable place: the model and its prompt. This is the deterministic-edges-around-a-probabilistic-core discipline expressed as a build sequence.

Production concerns, guardrails, cost, evaluation, observability, come after the feature works, but not because they are afterthoughts; they come after because they instrument and harden a thing that must first exist to be instrumented. You cannot meter the cost of a call you have not written or evaluate answers a system does not yet produce. Sequencing them here is not "productionize later," the myth we warned against; it is building each safeguard at the first moment it has something real to protect.

Deployment and operations come last for the obvious reason, you ship what you have built, but also because deployment choices, streaming through Nginx, running Celery workers, managing secrets, are informed by everything the application turned out to need. Deciding your production topology before you know your workload is guessing. By building it last, we deploy a system we thoroughly understand, which is the only kind worth operating at 3 a.m.

Summary

We are going to build DocuMind, an AI document assistant that lets users upload documents and chat with them through a Retrieval-Augmented Generation pipeline, and we are going to build it the way a real product is built rather than the way a demo is thrown together. An AI application is not a single API call; it is an anatomy of organs, the model, prompting, grounding, streaming, guardrails, cost, evaluation, and observability, that must all be present and cooperating for the system to be trustworthy. We build on Django because an AI product is still a product that needs authentication, data isolation, background work, and a mature web foundation, and we pair it with PostgreSQL and pgvector, Redis and Celery, the Claude Messages API, and a lightweight HTMX and Alpine.js frontend, choosing a small stack we can fully understand. The line between a demo and production runs through input validation, failure handling, background processing, cost awareness, data scoping, evaluation, and observability, all cheap to build in early and expensive to retrofit. And underlying everything is non-determinism: the model's probabilistic core changes how we test, debug, and design, pushing us toward deterministic, well-tested edges around a variable center. The five parts of the book take DocuMind from an empty project to a deployed, operable product, one honest trade-off at a time.

Key Takeaways

- **One product, end to end.** The entire book builds a single real application, DocuMind, so every concept lands in the context of a system that must actually work, not an isolated snippet.
- **An AI app is an anatomy, not an API call.** The model, prompting, grounding/RAG, streaming, guardrails, cost, evaluation, and observability are all first-class components; a system missing any of them is a demo.
- **Django is the right home for an AI product** because the product still needs auth, multi-tenant data isolation, background tasks, and a secure web foundation, and Django provides them without ceremony.
- **A small, understood stack beats a large, mysterious one.** PostgreSQL with pgvector, Redis, Celery, the Claude API, and HTMX/Alpine give us everything we need with the fewest moving parts to operate at 3 a.m.
- **Production readiness is built in, not bolted on.** Validation, retries, background processing, scoping, cost metering, evals, and observability are cheap early and painful to retrofit, so we build them from Chapter 1.

- **Non-determinism changes the engineering.** You test on structure and properties rather than exact strings, you rely on evaluations for quality, and you design deterministic, testable edges around a probabilistic core.
- **Data scoping is load-bearing.** A single query filter is what keeps one tenant from reading another's documents; small lines carry large responsibilities and must be enforced and tested.
- **Honesty about trade-offs is the whole point.** Cost, latency, failure modes, and residual hallucination are named out loud, because a production-ready system is one whose limitations you understand and manage rather than hide.

End of sample

You have read the opening chapter of

Build an AI-Powered Django App

The complete book continues from here — every chapter, every appendix, in PDF and EPUB.

EUR 39.00

Get the full book

https://djangozen.com/ebooks/book/build-ai-powered-django-app/