

FREE PREVIEW

From Python Developer to AI Engineer

Explore selected chapters from The AI Engineer — Volume I.

Learn how production software engineering connects with modern AI, from Python and FastAPI to embeddings, semantic search, and Retrieval-Augmented Generation.

This preview gives you a hands-on look at the engineering approach used throughout the book.

THE AI ENGINEER

Build. Test. Deploy. Scale.

Volume I --- Build the Foundations

Free Preview

Selected Chapters and Excerpts

Author's Note

Welcome to **The AI Engineer --- Volume I: Build the Foundations.**

This preview is designed to give you a practical taste of the engineering approach used throughout the book.

You have already seen the opening chapter. The pages that follow move directly into the technical foundations that connect professional software engineering with modern AI application development.

The goal is not to treat AI as a black box.

The goal is to understand the systems built around AI: APIs, databases, security, testing, embeddings, semantic search, retrieval, and grounded generation.

This preview deliberately moves across several layers of the system so you can see how the pieces connect.

You will begin with a production API, move into the mathematics and representation behind modern AI systems, build an intuition for embeddings and semantic search, and finally see how retrieval becomes the foundation of Retrieval-Augmented Generation.

The complete volume continues this journey through LLM application architecture, AI processing pipelines, caching, background jobs, and performance engineering.

Ahmed Adawy

PART 1

VOLUME I
CHAPTERS 1 – 16

THE AI ENGINEER

Build the Foundations

LEARN / BUILD / DEPLOY

I ENGINEERING MINDSET
Chapters 1 – 3
From Python Developer to AI Engineer

II BUILDING THE BACKEND
Chapters 4 – 6
FastAPI, Databases, and Security

III TESTING AND SOFTWARE QUALITY
Chapters 7 – 8
Testing and Reliability Engineering

IV AI ENGINEERING FOUNDATIONS
Chapters 9 – 12
Embeddings, Semantic Search, RAG, LLM Architecture

V BUILDING THE AI APPLICATION
Chapters 13 – 16
Knowledge Platform, Processing Pipeline, Caching, Performance



FROM CODE TO INTELLIGENCE

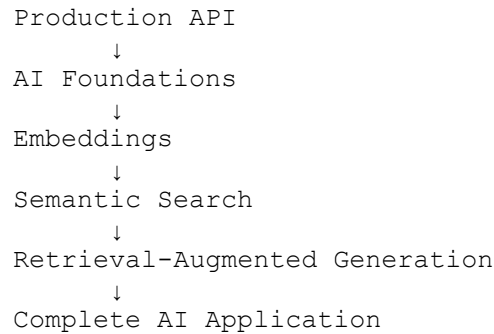
THE AI ENGINEER

Build. Test. Deploy. Scale.

**A Project-Based AI
Engineering Bootcamp in
a Book**

Ahmed Adawy

Preview Roadmap



The important idea is that these are not isolated technologies.

They are layers of one engineering system.

TABLE OF CONTENTS

THE AI ENGINEER

Build. Test. Deploy. Scale.

A Project-Based AI Engineering Bootcamp in a Book

PART I — THE ENGINEERING MINDSET

Chapter 1 — From Python Developer to AI Engineer

- What AI Engineering Really Means
- Software Engineering vs. AI Engineering
- The Production AI Engineer
- The AI Engineering Lifecycle
- From Prototype to Production
- The Architecture of Our Project
- Development Environment Setup
- Engineering Standards and Conventions

Chapter 2 — Building a Professional Python Project

- Project Structure
- Virtual Environments
- Dependency Management
- Configuration Management
- Environment Variables
- Type Hints
- Dataclasses and Models
- Logging
- Error Handling
- Clean Code Principles
- Packaging and Reusable Modules

Chapter 3 — Python for Production Performance

- Efficient Data Structures
- Iterators and Generators
- Memory Management
- Vectorized Operations
- NumPy for Engineering Workloads
- Profiling Python Applications
- Measuring Performance
- CPU-Bound vs. I/O-Bound Work
- Concurrency Fundamentals
- Performance Optimization Workflow

PART II — BUILDING THE BACKEND

Chapter 4 — Designing a Production API with FastAPI

- REST API Architecture
- FastAPI Project Structure
- Routes and Endpoints
- Request and Response Models
- Validation
- Dependency Injection
- HTTP Status Codes
- Error Handling
- API Documentation
- Authentication Foundations
- Building the First Production API

Chapter 5 — Databases and Persistent Data

- Relational Database Fundamentals
- PostgreSQL
- Database Schema Design
- Tables and Relationships
- SQL Queries
- Indexes
- Transactions
- Migrations
- Connection Management
- Repository Patterns
- Database Performance

Chapter 6 — Authentication, Authorization, and Security

- Authentication vs. Authorization
- Password Hashing
- JSON Web Tokens
- Access Control
- API Security
- Secrets Management
- Input Validation
- Rate Limiting
- Common API Vulnerabilities
- Secure Production Configuration

PART III — TESTING AND SOFTWARE QUALITY

Chapter 7 — Testing Production Python Systems

- Why Production Systems Need Tests
- Unit Testing
- Integration Testing
- End-to-End Testing
- pytest
- Fixtures
- Mocking
- Testing FastAPI
- Testing Database Operations
- Testing AI Components
- Code Coverage
- Designing a Test Strategy

Chapter 8 — Debugging and Reliability Engineering

- Reading Stack Traces
- Reproducing Bugs
- Structured Logging
- Failure Analysis
- Defensive Programming
- Timeouts
- Retries
- Circuit Breakers
- Idempotency
- Graceful Failure
- Reliability Checklists

PART IV — AI ENGINEERING FOUNDATIONS

Chapter 9 — Understanding AI Systems from the Inside

- What Happens Inside an AI System
- Vectors and Vector Spaces
- Embeddings
- Similarity and Distance
- Dot Product
- Cosine Similarity
- Probability and Model Outputs
- Neural Network Fundamentals
- The Role of Mathematics in AI Engineering

Chapter 10 — Building Embedding and Semantic Search Systems

- Document Ingestion
- Text Cleaning
- Chunking Strategies
- Embedding Generation
- Vector Representations
- Similarity Search
- Top-K Retrieval
- Ranking
- Search Evaluation
- Building Semantic Search from Scratch

Chapter 11 — Retrieval-Augmented Generation

- Why RAG Exists
- The RAG Architecture

- Retrieval Pipelines
- Context Construction
- Prompt Construction
- Generation
- Grounded Answers
- Hallucination Control
- Retrieval Quality
- RAG Evaluation
- Building a Production RAG Pipeline

Chapter 12 — LLM Application Architecture

- LLM APIs
- Model Selection
- Prompt Engineering
- Structured Outputs
- Function Calling
- Streaming Responses
- Token Management
- Context Windows
- Cost Management
- Latency Optimization
- Designing Reliable LLM Applications

PART V — BUILDING THE AI APPLICATION

Chapter 13 — Building the Knowledge Platform

- Project Requirements
- System Architecture
- User Management
- Document Management
- Document Upload
- Document Processing
- Metadata Management
- Search API
- AI Question Answering
- Application Workflow

Chapter 14 — The AI Processing Pipeline

- Ingestion Architecture
- Background Processing
- Document Extraction
- Chunking
- Embedding

- Indexing
- Retrieval
- Generation
- Pipeline Orchestration
- Failure Recovery
- Pipeline Monitoring

Chapter 15 — Caching and Background Jobs

- Why Background Processing Matters
- Redis Fundamentals
- Caching Strategies
- Cache Invalidation
- Job Queues
- Worker Processes
- Async Processing
- Retry Policies
- Job Monitoring
- Designing Reliable Background Tasks

Chapter 16 — Performance Engineering for AI Applications

- Measuring Latency
- Throughput
- Bottleneck Identification
- Database Optimization
- Vector Search Optimization
- Caching
- Batch Processing
- Async I/O
- Model Latency
- Token Optimization
- Load Testing
- Performance Budgets

PART VI — CONTAINERIZATION AND DEVOPS

Chapter 17 — Dockerizing the AI Application

- Why Containers Matter
- Docker Fundamentals
- Images and Containers
- Dockerfiles
- Multi-Stage Builds
- Dependency Isolation
- Environment Configuration

- Docker Compose
- API Container
- Database Container
- Redis Container
- Worker Container
- Local Production-Like Environment

Chapter 18 — Linux and Production Operations

- Linux Fundamentals for Engineers
- Processes
- Services
- Filesystems
- Permissions
- Environment Variables
- Networking
- Ports
- Logs
- Resource Monitoring
- Process Management
- Production Troubleshooting

Chapter 19 — Git, GitHub, and Engineering Workflows

- Git Fundamentals
- Branching Strategies
- Commits
- Pull Requests
- Code Reviews
- Repository Structure
- Issues and Project Tracking
- Release Management
- Versioning
- Professional Git Workflows

Chapter 20 — Continuous Integration and Continuous Deployment

- What CI/CD Really Means
- GitHub Actions
- Automated Testing
- Linting and Quality Checks
- Dependency Checks
- Building Docker Images
- Automated Releases
- Deployment Pipelines
- Secrets and Environment Variables
- Rollbacks
- Production Deployment Workflow

PART VII — CLOUD-NATIVE AI ENGINEERING

Chapter 21 — Deploying AI Systems to the Cloud

- Cloud Architecture Fundamentals
- Application Deployment
- Managed Databases
- Environment Configuration
- Secrets
- HTTPS
- Domains
- Health Checks
- Deployment Strategies
- Scaling
- Cost Awareness
- Production Readiness

Chapter 22 — Kubernetes for AI Applications

- Why Kubernetes
- Kubernetes Architecture
- Pods
- Deployments
- Services
- ConfigMaps
- Secrets
- Health Probes
- Resource Requests and Limits
- Scaling
- Rolling Deployments
- Deploying the AI Platform

Chapter 23 — Observability and Production Monitoring

- Logs
- Metrics
- Traces
- Health Monitoring
- Application Metrics
- Database Metrics
- AI-Specific Metrics
- Latency Monitoring
- Error Rates
- Token Usage
- Cost Monitoring
- Alerts

- Incident Response

PART VIII — SECURITY, SCALE, AND PRODUCTION

Chapter 24 — Securing Production AI Systems

- Threat Modeling
- API Security
- Authentication
- Authorization
- Secrets
- Dependency Security
- Container Security
- Data Protection
- Prompt Injection
- Retrieval Security
- AI Application Abuse
- Security Testing

Chapter 25 — Scaling the AI Platform

- Vertical vs. Horizontal Scaling
- Stateless Services
- Load Balancing
- Database Scaling
- Caching at Scale
- Queue-Based Architecture
- Worker Scaling
- Vector Search Scaling
- Model Scaling
- Capacity Planning
- Cost vs. Performance

Chapter 26 — AI Engineering Architecture Patterns

- Monolith vs. Microservices
- Modular Monoliths
- Event-Driven Architecture
- Pipeline Architecture
- Service-Oriented Architecture
- Retrieval Architecture
- AI Gateway Patterns
- Worker Architecture
- Data Processing Architecture
- Choosing the Right Architecture

PART IX — ENGINEERING THE COMPLETE SYSTEM

Chapter 27 — Integrating Everything

- Final System Architecture
- Connecting the API
- Connecting the Database
- Connecting Redis
- Connecting the AI Pipeline
- Connecting Semantic Search
- Connecting the LLM
- Authentication
- Testing
- Docker
- CI/CD
- Cloud Deployment
- Monitoring

Chapter 28 — Production Readiness Review

- Code Quality Review
- Security Review
- Performance Review
- Database Review
- API Review
- AI Quality Review
- Testing Review
- Deployment Review
- Observability Review
- Disaster Recovery
- Production Readiness Checklist

PART X — THE ENGINEERING CHALLENGE

Chapter 29 — The Final Project Specification

- The Business Problem
- Functional Requirements
- Technical Requirements
- API Requirements

- AI Requirements
- Database Requirements
- Security Requirements
- Testing Requirements
- Deployment Requirements
- Monitoring Requirements

Chapter 30 — Build the Final Production System

- Architecture Design
- Database Design
- API Implementation
- Authentication
- Document Processing
- Semantic Search
- RAG
- Background Workers
- Testing
- Docker
- CI/CD
- Cloud Deployment
- Monitoring
- Performance Optimization

Chapter 31 — The Final Engineering Challenge

- Requirements Without a Solution
- Architecture Challenge
- Implementation Challenge
- Debugging Challenge
- Performance Challenge
- Security Challenge
- Deployment Challenge
- Production Incident Simulation
- Scaling Challenge

Chapter 32 — Reference Architecture and Complete Solution

- Final Architecture
- Complete Project Structure
- Complete Backend
- Complete AI Pipeline
- Complete Database Layer
- Complete Test Suite
- Complete Docker Configuration
- Complete CI/CD Pipeline
- Complete Deployment Configuration
- Monitoring Configuration
- Production Checklist

APPENDICES

Appendix A — Python Engineering Cheat Sheet

Appendix B — FastAPI Cheat Sheet

Appendix C — SQL and PostgreSQL Cheat Sheet

Appendix D — pytest Cheat Sheet

Appendix E — Docker Cheat Sheet

Appendix F — Git and GitHub Cheat Sheet

Appendix G — GitHub Actions Cheat Sheet

Appendix H — Kubernetes Cheat Sheet

Appendix I — AI and LLM Engineering Cheat Sheet

Appendix J — Semantic Search and RAG Cheat Sheet

Appendix K — Production Troubleshooting Guide

Appendix L — AI Engineering Interview Questions

Appendix M — Production Readiness Checklist

Appendix N — Final Project Evaluation Rubric

Appendix O — Complete Project Repository Structure

Appendix P — Glossary of AI Engineering Terms

Chapter 1 — From Python Developer to AI Engineer

What AI Engineering Really Means

Artificial Intelligence has moved from research laboratories into production software.

Today, an AI-powered application is rarely just a model.

A production AI system is a complete software system that combines application code, data, machine learning models, APIs, databases, infrastructure, security, testing, monitoring, and operational processes.

This distinction is fundamental.

A developer can write a Python script that sends a prompt to a language model and receives an answer in a few lines of code.

An AI Engineer must answer much harder questions:

- Where does the data come from?
- How is the data validated?
- How is it stored?
- How are documents processed?
- How are embeddings generated?
- How are relevant documents retrieved?
- How is context supplied to the model?
- How are users authenticated?
- How are requests validated?
- What happens when the model is unavailable?
- How do we test the system?
- How do we measure performance?
- How do we control cost?
- How do we deploy the application?
- How do we monitor it after deployment?
- How do we scale it when the number of users increases?
- How do we secure the system against attacks and misuse?

These questions define the difference between an AI demo and an AI product.

An AI Engineer does not simply make a model produce an answer.

An AI Engineer builds the system around the intelligence.

Software Engineering vs. AI Engineering

Traditional software engineering usually follows a relatively deterministic path.

A request enters the system, application logic processes the request, data is retrieved or modified, and a response is returned.

An AI system introduces additional sources of complexity.

The system may depend on:

- probabilistic model behavior
- large datasets
- embeddings
- retrieval systems
- external model APIs
- GPU or specialized compute
- token limits
- model latency
- model cost
- changing model versions
- generated outputs that require validation

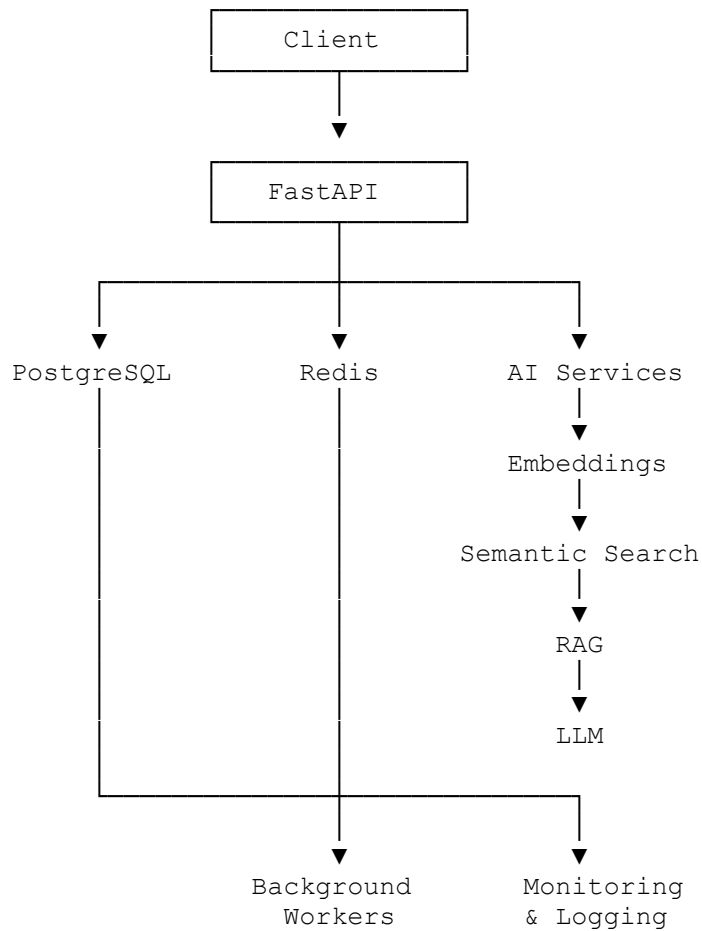
Consider a traditional API:

```
Client
  ↓
API
  ↓
Business Logic
  ↓
Database
  ↓
Response
```

An AI application may look more like this:

```
Client
  ↓
API
  ↓
Authentication
  ↓
Business Logic
  ↓
Data Retrieval
  ↓
Semantic Search
  ↓
Context Construction
  ↓
LLM
  ↓
Output Validation
  ↓
Response
```

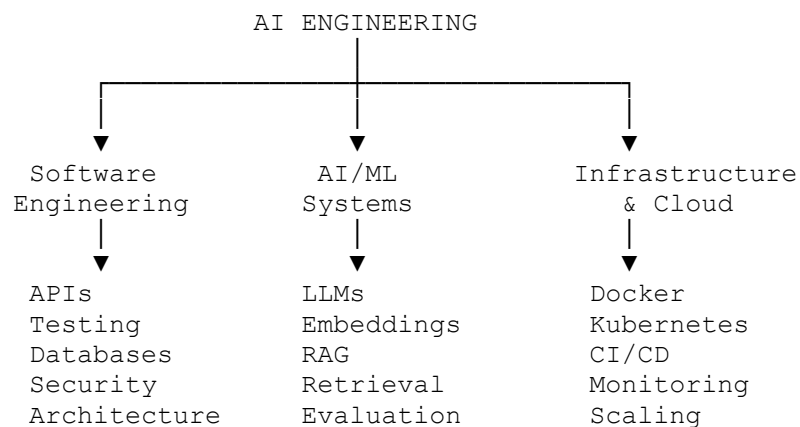
And the production infrastructure surrounding it may look like:



The AI Engineer must understand every major layer.

The Production AI Engineer

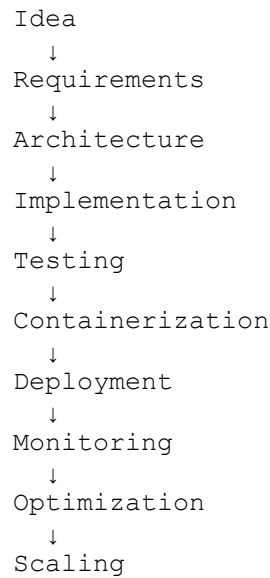
An AI Engineer works at the intersection of several disciplines.



The goal is not to become an expert in every possible technology.

The goal is to understand how the technologies work together to create a reliable system.

A production AI Engineer should be able to move through the complete lifecycle:



This book follows that lifecycle.

The AI Engineering Lifecycle

Every serious AI application should be treated as an engineering system rather than a collection of prompts.

The lifecycle begins with a problem.

Step 1 — Define the Problem

Before choosing a model or framework, define what the system is supposed to accomplish.

For example:

Build a platform that allows users to upload technical documents and ask questions about their contents.

This requirement is intentionally simple.

The engineering work begins when we translate it into technical requirements.

The system must be able to:

- register users
- authenticate users
- upload documents
- process documents
- extract text

- split documents into chunks
- generate embeddings
- store searchable representations
- retrieve relevant information
- generate answers
- expose an API
- test the application
- deploy the application
- monitor the application

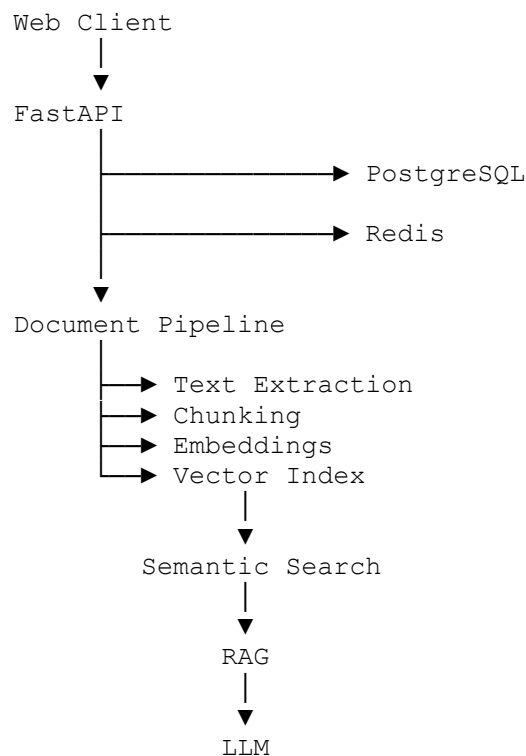
The original business requirement has now become an engineering system.

Step 2 — Design the Architecture

The next question is:

What components are required?

A first architectural design might be:



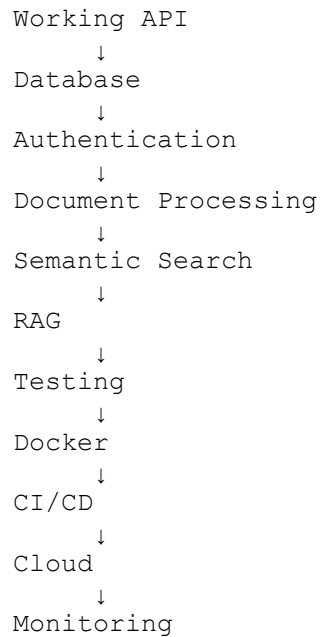
Step 3 — Build the Application

Once the architecture is defined, implementation begins.

The project will be developed incrementally.

We will not attempt to build the complete system in a single step.

Instead, we will create a working system and continuously expand it.



Step 4 — Test

A system that works once is not necessarily a reliable system.

We need tests that verify:

- individual functions
- API endpoints
- database operations
- authentication
- document processing
- search
- retrieval
- AI responses
- failure scenarios

Testing becomes part of development rather than something performed at the end.

Step 5 — Deploy

A local application is only the beginning.

The application must eventually run in a production environment.

That introduces new concerns:

- environment variables
- secrets
- HTTPS

- databases
- containers
- networking
- resource limits
- logging
- monitoring
- deployment automation

Step 6 — Operate

Production is not the end of engineering.

Once users begin interacting with the system, new questions appear.

How many requests are being received?

How long does each request take?

How often does the application fail?

How much does each AI request cost?

Which component is causing latency?

Is the database becoming a bottleneck?

Is the model producing poor answers?

Engineering continues after deployment.

From Prototype to Production

A prototype is optimized for learning.

A production system is optimized for reliability.

Consider this prototype:

```
response = model.generate(  
    "Explain this document."  
)  
  
print(response)
```

This may be enough to demonstrate that a model can generate an answer.

But a production system needs much more.

User
↓

Authentication
↓
API Validation
↓
Rate Limiting
↓
Request Processing
↓
Document Retrieval
↓
Context Construction
↓
Model Request
↓
Output Validation
↓
Logging
↓
Metrics
↓
Response

The prototype asks:

Can this work?

The production system asks:

Can this work reliably for thousands of requests under real-world conditions?

That difference is the heart of AI Engineering.

The Architecture of Our Project

Throughout this book, we will build a complete AI-powered knowledge platform.

The system will allow users to:

1. Create an account.
2. Authenticate securely.
3. Upload technical documents.
4. Process the documents.
5. Extract and normalize their text.
6. Split the text into meaningful chunks.
7. Generate vector embeddings.
8. Store searchable representations.
9. Perform semantic search.
10. Retrieve relevant context.
11. Ask questions about the documents.
12. Generate grounded answers using an LLM.
13. Monitor application activity.
14. Run automated tests.
15. Package the system using Docker.

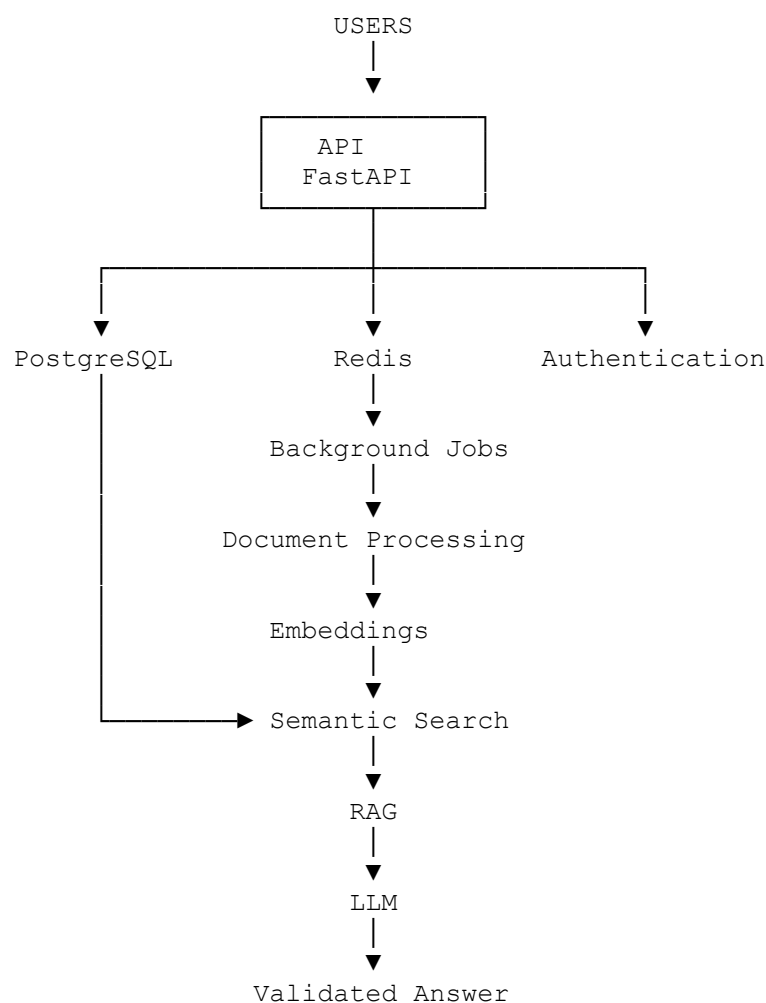
16. Automate CI/CD using GitHub Actions.
17. Deploy the application.
18. Monitor the production environment.
19. Optimize performance.
20. Scale the system.

The final architecture will evolve throughout the book.

We will deliberately avoid pretending that the first architecture is perfect.

Engineering is an iterative process.

A simplified final architecture will look like this:



Around this architecture we will build:

Testing
Security
Logging
Monitoring
Docker
GitHub
GitHub Actions

Cloud Deployment
Kubernetes
Performance Engineering

Development Environment Setup

Before building the system, we need a professional development environment.

The exact operating system may differ between developers, but the fundamental tools remain similar.

The project will use:

- Python
- Git
- GitHub
- PostgreSQL
- Redis
- Docker
- pytest
- FastAPI
- GitHub Actions
- Kubernetes
- an LLM provider or compatible model service

The project should be developed inside an isolated Python environment.

A typical setup begins with:

```
python -m venv .venv
```

Activate the environment.

On Linux or macOS:

```
source .venv/bin/activate
```

On Windows:

```
.venv\Scripts\Activate.ps1
```

Verify Python:

```
python --version
```

Verify pip:

```
python -m pip --version
```

Create the project directory:

```
mkdir ai-engineer-platform
cd ai-engineer-platform
```

Initialize Git:

```
git init
```

Create the basic project structure:

```
ai-engineer-platform/
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── api/
│   ├── core/
│   ├── models/
│   ├── services/
│   └── repositories/
├── tests/
├── scripts/
├── docs/
├── .env.example
├── .gitignore
├── pyproject.toml
└── README.md
```

This structure will grow throughout the book.

The important principle is that the project should remain understandable as it becomes larger.

Engineering Standards and Conventions

A production project needs consistent rules.

We will follow several principles throughout this book.

Principle 1 — Make the System Understandable

Code should be organized so that another engineer can understand the system without reconstructing its architecture from scattered files.

Principle 2 — Prefer Explicit Behavior

Hidden behavior creates debugging problems.

Prefer code whose behavior is obvious.

Principle 3 — Validate at Boundaries

External input should never be trusted automatically.

Validate:

- API requests
- uploaded files
- configuration
- database data
- external API responses
- model outputs

Principle 4 — Fail Clearly

A failure should produce enough information to determine:

- what failed
- where it failed
- why it failed
- what request caused it
- whether recovery is possible

Principle 5 — Test Important Behavior

Tests should focus on behavior that matters.

The goal is not to maximize the number of test files.

The goal is to create confidence in the system.

Principle 6 — Automate Repetitive Work

If a process is performed repeatedly, consider automating it.

Examples include:

- running tests
- checking code quality
- building containers
- publishing releases
- deploying applications
- collecting metrics

Principle 7 — Measure Before Optimizing

Performance optimization should be based on measurements.

Do not optimize because a piece of code looks slow.

Measure it.

Then identify the actual bottleneck.

Principle 8 — Security Is Part of Architecture

Security should not be added after the application has been built.

Authentication, authorization, secrets management, validation, logging, and secure communication must be considered from the beginning.

Principle 9 — Production Is a Different Environment

A program working on a developer's laptop does not prove that it is production-ready.

Production introduces:

- real users
- unpredictable traffic
- failures
- network problems
- security threats
- resource limitations
- operational costs

The system must be designed with these realities in mind.

Principle 10 — Build, Test, Deploy, Scale

The entire philosophy of this book can be summarized in four words:

```
BUILD
↓
TEST
↓
DEPLOY
↓
SCALE
```

We will repeat this cycle throughout the entire project.

The objective is not simply to finish a Python application.

The objective is to learn how to engineer an AI system that can survive outside the development environment.

Chapter 1 Lab — Define Your AI System

Before writing the first production component, define the system you are going to build.

Answer the following questions:

1. What problem does the system solve?
2. Who are the users?
3. What information does the system process?
4. What does the user send to the system?
5. What does the system return?
6. Which data must be stored?
7. Which operations require AI?
8. Which operations do not require AI?
9. Which components need to communicate with each other?
10. What could fail?
11. What should happen when an external AI service becomes unavailable?
12. How will the system be tested?
13. How will it be deployed?
14. How will its health be monitored?
15. What would need to change if the number of users increased by 100 times?

Write the answers before implementing the system.

A strong AI Engineer does not begin with a framework.

A strong AI Engineer begins with a problem, defines the requirements, designs the system, and then chooses the technology required to implement it.

Chapter 1 Checkpoint

At the end of this chapter, you should understand:

- What AI Engineering means.
- Why an AI application is more than an AI model.
- The difference between a prototype and a production system.
- The major components of a production AI architecture.
- The AI Engineering lifecycle.
- Why testing, security, deployment, and monitoring must be considered from the beginning.
- The architecture of the project we will build.
- The engineering mindset required for the rest of this book.

Selected Excerpt --- Chapter 4

Designing a Production API with FastAPI

A production API is more than a collection of endpoints.

A beginner may think of an API as:

```
Request
  ↓
Function
  ↓
Response
```

A production engineer thinks about a much larger system:

```
Client
  ↓
HTTP Request
  ↓
Authentication
  ↓
Validation
  ↓
Routing
  ↓
Business Logic
  ↓
Persistence / External Services
  ↓
Error Handling
  ↓
Observability
  ↓
HTTP Response
```

Every layer exists because something can go wrong.

A request can be malformed.

A user can be unauthenticated.

An authenticated user can attempt to access another user's resource.

A database can be unavailable.

An external AI provider can time out.

A service can become slow under load.

The purpose of production API engineering is therefore not simply to make the happy path work.

It is to make the system behave predictably when the unhappy paths occur.

Thin Routes, Strong Services

One of the most useful architectural rules is:

Keep API routes thin and business logic independent of the HTTP layer.

A route should coordinate the request rather than contain the entire application.

A simplified structure looks like this:

```
from fastapi import APIRouter, Depends

router = APIRouter()

@router.post("/documents")
async def create_document(
    request: DocumentCreate,
    service: DocumentService = Depends(get_document_service),
):
    return await service.create(request)
```

The route handles HTTP concerns.

The service handles application behavior.

This separation makes the system easier to test because the business logic does not need a real HTTP request in order to execute.

Validation Is a Security Boundary

Incoming data should never be trusted simply because it came through your own frontend.

APIs may be called directly by scripts, other services, malicious clients, or outdated applications.

Pydantic models provide a clear contract:

```
from pydantic import BaseModel, Field

class DocumentCreate(BaseModel):
    title: str = Field(min_length=1, max_length=200)
    content: str = Field(min_length=1)
```

Validation gives the application a predictable internal representation.

The important engineering principle is:

```
Untrusted Input
  ↓
Validation
  ↓
Trusted Application Model
  ↓
Business Logic
```

Validation should happen as early as practical.

Dependency Injection

FastAPI's dependency system is useful because infrastructure dependencies can be separated from application logic.

For example:

```
from fastapi import Depends

async def get_current_user():
    ...

@router.get("/me")
async def get_me(user=Depends(get_current_user)):
    return user
```

The route does not need to know how authentication tokens are decoded.

That responsibility belongs to the authentication dependency.

The same idea can be applied to database sessions, services, repositories, configuration, and other infrastructure components.

Errors Are Part of the API Contract

A production API should not turn every failure into an unexplained 500 Internal Server Error.

Different failures communicate different meanings.

```
400 → Invalid request
401 → Authentication required
403 → Authenticated but not allowed
404 → Resource not found
409 → Conflict
422 → Validation failure
429 → Rate limited
500 → Unexpected server failure
503 → Dependency/service unavailable
```

The exact API contract depends on the system, but the principle remains:

Errors should be intentional, consistent, and useful.

A good API makes failure understandable to both clients and operators.

Selected Excerpt --- Chapter 9

Understanding AI Systems from the Inside

Modern AI applications often look deceptively simple from the outside.

A user submits a question.

A model returns an answer.

But between those two events is a mathematical and engineering pipeline.

One of the most important concepts to understand is the **vector representation**.

From Text to Numbers

Computers do not directly compare the meaning of two sentences.

AI systems therefore transform information into numerical representations.

An embedding is a vector:

```
embedding =  
[  
    0.21,  
   -0.07,  
    0.84,  
    ...  
]
```

The individual values are not normally meaningful by themselves.

The important property is the position of the vector within a high-dimensional space.

Conceptually:

Text

↓
Embedding Model
↓
Vector
↓
Position in Vector Space

Texts with related meanings may be represented by vectors that are close to one another.

This makes mathematical similarity a useful tool for semantic retrieval.

Cosine Similarity

One common measure is cosine similarity.

For vectors a and b :

$$\text{cosine_similarity}(a, b) = \frac{(a \cdot b)}{(\|a\| \|b\|)}$$

In Python:

```
import numpy as np

def cosine_similarity(
    a: np.ndarray,
    b: np.ndarray,
) -> float:
    denominator = np.linalg.norm(a) * np.linalg.norm(b)

    if denominator == 0:
        raise ValueError("Cannot compare zero vectors")

    return float(np.dot(a, b) / denominator)
```

The value is based on the angle between the vectors.

This is useful because semantic search is not normally asking:

"Do these two pieces of text contain exactly the same words?"

Instead, it is trying to answer:

"Are these pieces of information related in meaning?"

Similarity Is Not Relevance

This distinction is extremely important.

A vector database can return the mathematically closest vectors.

That does not automatically mean that the results are useful to the user.

A search system therefore has multiple engineering layers:

```
Query
↓
Query Representation
↓
Candidate Retrieval
↓
Filtering
↓
Ranking
↓
Top-K Results
```

The embedding model is one component of this process.

It is not the entire search system.

AI Is a System, Not a Single Model

A production AI application commonly contains:

```
Application
↓
Data
↓
Retrieval
↓
Context
↓
Model
↓
Validation
↓
Response
```

The model can be excellent and the application can still fail.

Bad data can produce bad answers.

Poor retrieval can provide irrelevant context.

Too much context can reduce useful signal.

Missing validation can allow malformed output.

Weak authorization can expose another user's information.

AI engineering therefore requires systems thinking.

Selected Excerpt --- Chapter 10

Building Embedding and Semantic Search Systems

Semantic search begins with a simple transformation:

```
Document
  ↓
Chunks
  ↓
Embeddings
  ↓
Vector Index
```

When a user asks a question, the same representation process is applied:

```
Question
  ↓
Query Embedding
  ↓
Vector Search
  ↓
Top-K Chunks
```

The result is a search system based on meaning rather than exact keyword matching.

Why Chunking Matters

A document is usually too large to treat as one retrieval unit.

Instead, it is divided into smaller chunks.

```
def chunk_text(
    text: str,
    chunk_size: int,
    overlap: int,
) -> list[str]:
    if overlap >= chunk_size:
        raise ValueError(
            "overlap must be smaller than chunk_size"
        )

    chunks = []
    start = 0

    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start = end - overlap
```

```
return chunks
```

The exact chunking strategy depends on the data.

Technical documentation may benefit from preserving headings and code blocks.

Legal documents may need section-aware boundaries.

Source code should not necessarily be split in the same way as prose.

Chunking is therefore an engineering decision, not merely a string-processing trick.

Metadata Matters

A chunk should not be stored as anonymous text.

A useful representation may contain:

```
chunk_id
document_id
user_id
chunk_index
content
source
page
embedding_model
embedding_version
created_at
```

Metadata enables filtering, authorization, debugging, and reproducibility.

Most importantly, ownership metadata must be enforced during retrieval.

Retrieval Must Respect Authorization

Consider two users:

```
User A
└─ Private Document A
```

```
User B
└─ Private Document B
```

A semantic search query from User A must never retrieve chunks belonging to User B.

Conceptually:

```
SELECT ...  
FROM document_chunks  
WHERE user_id = :user_id  
ORDER BY embedding_distance  
LIMIT :limit;
```

The ownership constraint is not an optional optimization.

It is a security boundary.

A highly relevant result that belongs to another user is still an unauthorized result.

Top-K Retrieval

Suppose the search system retrieves the five closest chunks:

1. Chunk 42	score=0.91
2. Chunk 17	score=0.88
3. Chunk 83	score=0.84
4. Chunk 11	score=0.81
5. Chunk 64	score=0.79

The application can then construct a context window from those results.

But top-k is not a universal constant.

Too few results may omit critical evidence.

Too many results may introduce noise and increase token usage.

The correct value must be evaluated against the actual workload.

A Complete Semantic Search Pipeline

```
User Query  
  ↓  
Normalize Query  
  ↓  
Create Query Embedding  
  ↓  
Apply Authorization Filters  
  ↓  
Vector Search  
  ↓  
Candidate Results  
  ↓  
Optional Re-ranking  
  ↓  
Top-K Context
```

This is already an AI system.

No generation is required yet.

The system is useful because it can retrieve information based on semantic relationships.

Selected Excerpt --- Chapter 11

Retrieval-Augmented Generation

Large language models are powerful generators.

But a model does not automatically know the private documents stored inside your application.

Retrieval-Augmented Generation addresses this problem by connecting the model to external knowledge.

The basic architecture is:

```
User Question
      ↓
Query Embedding
      ↓
Semantic Retrieval
      ↓
Relevant Documents
      ↓
Context Construction
      ↓
LLM
      ↓
Grounded Answer
```

The model generates an answer using retrieved information rather than relying exclusively on its internal parameters.

Retrieval Comes Before Generation

A common mistake is to think of RAG as:

```
Question → LLM
```

A more useful mental model is:

```
Question
      ↓
What information do we need?
      ↓
Retrieve that information
      ↓
```

Give the model the relevant context

↓

Generate an answer

This changes how the system should be engineered.

If the answer is wrong, the debugging process should not immediately start with the prompt.

The problem could be:

Document Extraction

↓

Chunking

↓

Embedding

↓

Retrieval

↓

Ranking

↓

Context Construction

↓

Prompt

↓

Model

↓

Output Validation

Each stage is a potential failure point.

Grounded Answers

A production RAG system should make it possible to determine where an answer came from.

A simple contract might be:

```
from pydantic import BaseModel
```

```
class Citation(BaseModel):  
    chunk_id: int  
    document_id: int
```

```
class GroundedAnswer(BaseModel):  
    answer: str  
    citations: list[Citation]
```

The application can validate that every citation refers to a chunk that was actually retrieved.

```
valid_ids = {result.chunk_id for result in results}
```

```
if any(
    citation.chunk_id not in valid_ids
    for citation in output.citations
):
    raise ValueError("Invalid citation")
```

This is an important engineering principle:

Never blindly trust model-generated references.

The model is a component inside the system.

The application remains responsible for validating the result.

The Context Window Is a Resource

Retrieval does not mean:

"Send the entire database to the model."

Instead, the system must construct a useful context.

```
Retrieved Chunks
↓
Filter
↓
Rank
↓
Select
↓
Compress / Normalize
↓
Prompt Context
↓
LLM
```

Context has a cost.

More text means more tokens.

More tokens can mean higher latency and higher cost.

More context can also introduce irrelevant information.

The goal is therefore not maximum context.

The goal is **useful context**.

RAG Quality Is Multi-Dimensional

A RAG system can fail even when the LLM itself is capable.

A useful quality model is:

```
AI Quality
  ≈
Data Quality
× Retrieval Quality
× Context Quality
× Generation Quality
× Validation Quality
```

If retrieval returns irrelevant documents, the generator receives poor evidence.

If the context is incomplete, the model may lack the information required to answer.

If the output is not validated, the application may accept an answer that violates its contract.

RAG is therefore an engineering pipeline rather than a single prompt.

What Comes Next

The excerpts above represent only a small part of the complete engineering journey.

The full **The AI Engineer --- Volume I** continues with:

Chapter 12 --- LLM Application Architecture

You will learn how to treat the LLM as a dependency rather than the application itself, including provider abstractions, structured outputs, context management, model routing, caching, rate limits, streaming, and failure handling.

Chapter 13 --- Building the Knowledge Platform

The individual AI components become a coherent application architecture.

Chapter 14 --- The AI Processing Pipeline

Documents move through extraction, normalization, chunking, embedding, indexing, retrieval, generation, validation, and observability.

Chapter 15 --- Caching and Background Jobs

You will learn how asynchronous processing and caching change the performance and reliability characteristics of AI systems.

Chapter 16 --- Performance Engineering for AI Applications

The volume concludes its foundation stage by applying performance engineering to AI workloads, including latency, throughput, concurrency, database performance, caching, batching, and measurement.

Continue the Journey

THE AI ENGINEER

Volume I --- Build the Foundations

Python → APIs → Databases → Security → Testing → AI → Embeddings → Semantic Search → RAG → LLM Applications

The full book takes these concepts beyond isolated examples and connects them into one production-oriented AI engineering system.

AI engineering is not about making a model produce an impressive answer once.

It is about building a system that can repeatedly produce useful results while remaining secure, testable, observable, maintainable, measurable, and recoverable.

From Code to Intelligence.

Continue reading in the full book.