PRACTICAL TECH GUIDE

Building RAG Applications

A Hands-On Guide to RAG with Python

```
python -m docs_rag_app.cli --build-index
```

Building RAG Applications

A Hands-On Guide to Retrieval-Augmented Generation with Python

Blue J. Lion
Quiet Line Press (quietlinepress.com)

First Edition: May 2026

Published by Quiet Line Press (quietlinepress.com)

Table of Contents

About This Book

Book Positioning

This book is about building a useful RAG application without turning the first half of the journey into a survey of tools, vendors, and abstractions.

RAG is often explained in broad strokes:

1. take documents

2. retrieve relevant context

3. send that context to a model

4. produce a better answer

That description is true, but it hides the decisions that shape whether the system actually works well.

In practice, the quality of a RAG application depends on a few concrete choices:

1. how you load documents

2. how you chunk them

3. how you retrieve them

4. how you decide whether the evidence is good enough

5. how you evaluate the answers you get back

Those are the decisions this book focuses on.

The companion project for the book is `docs_rag_app`, a small local RAG application that answers questions over a document folder. Its repository is:

`https://github.com/nextframedev/docs_rag_app`

It starts simple on purpose. We will improve the same application chapter by chapter instead of jumping between unrelated demos.

Intended Audience

This book is for readers who:

1. can read basic Python
2. understand prompting at a basic level
3. want to build a practical AI application
4. want to understand how retrieval changes application design

It does not assume deep machine learning knowledge.

What You Will Build

The main project in this book is a small documentation and knowledge-base assistant.

Its first version will:

1. read local Markdown and text files
2. split them into chunks
3. retrieve relevant chunks for a question
4. assemble a grounded answer with citations

Later chapters will improve that same app by adding:

1. vector retrieval
2. reranking
3. evaluation
4. refresh and re-indexing
5. a local API and browser UI

The goal is not to build the largest system possible. The goal is to build a small system that makes the important RAG ideas visible.

In this book, `corpus` simply means the collection of source documents the app can read, index, and retrieve from, and the companion project keeps that scope deliberately narrow.

`docs_rag_app` is:

1. local-first

2. small enough to inspect

3. intentionally light on external dependencies

4. not meant to stand in for a full production platform

Those limits are part of the teaching strategy. The project is easier to learn from because its moving parts stay visible.

How To Use This Book

The most useful way to read this book is to keep the companion project open while you read.

Each chapter should do two things:

1. explain one practical idea

2. improve the same project in a visible way

That structure matters. RAG becomes much easier to understand when you can connect each idea to a real change in the application.

Quick Start In 5 Minutes

Before going deeper, it helps to get one working result as quickly as possible.

Start with the smallest possible Python sketch of the idea.

This is not the real app. It is a tiny no-library example that shows the RAG pattern directly:

```
documents = [
    {
        "id": "escalation",
        "text": (
            "Frontline support should escalate when an issue
cannot be "
            "reproduced within 30 minutes or when account data
may be at risk."
        ),
    },
```

```
    {
        "id": "incident",
        "text": (
            "During a sev1 incident, only the incident lead or
a delegated "
            "communications owner should publish external
updates."
        ),
    },
]

question = "When should frontline support escalate an issue?"

def tokenize(text: str) -> set[str]:
    return {word.strip(".,?!:;").lower() for word in
text.split() if word.strip()}

def retrieve(question: str, docs: list[dict], top_k: int = 2)
-> list[dict]:
    q = tokenize(question)
    ranked = []
    for doc in docs:
        score = len(q & tokenize(doc["text"]))
        if score > 0:
            ranked.append((score, doc))
    ranked.sort(key=lambda item: item[0], reverse=True)
    return [doc for _score, doc in ranked[:top_k]]

def answer_from_context(context_docs: list[dict]) -> str:
    if not context_docs:
        return "I could not find supporting context."
    joined = " ".join(doc["text"] for doc in context_docs)
    return f"Based on retrieved context: {joined}"

results = retrieve(question, documents)
answer = answer_from_context(results)

print(answer)
print("Sources:", [doc["id"] for doc in results])
```

That tiny script already shows the core loop:

1. keep a small document set

2. retrieve relevant context

3. answer from that context

Now run the real companion project.

Clone the repo and move into it:

```
git clone https://github.com/nextframedev/docs_rag_app.git
cd docs_rag_app
python3 -m venv .venv
source .venv/bin/activate
pip install -e '.[dev]'
```

Then ask one grounded question over the sample corpus. The sample corpus is a small fictional support-team handbook written for this project, with a mix of Markdown policy pages and plain-text handoff or queue notes.

```
python -m docs_rag_app.cli \
  --corpus examples/sample_corpus \
  --question "Who should publish external updates during a sev1 incident?"
```

You should expect:

1. a short grounded answer

2. a `Citations:` section

3. one or more cited source lines

For example, the output shape will look roughly like this:

```
Based on the retrieved evidence, only the incident lead or a delegated
communications owner should publish external updates during a sev1
incident.

Citations:
- incident-communication-md-0 -> .../examples/sample_corpus/incident-
communication.md
- onboarding-checklist-md-0 -> .../examples/sample_corpus/onboarding-
checklist.md
```

If you want to see the structured payload too:

```
python -m docs_rag_app.cli \
  --corpus examples/sample_corpus \
  --question "What should a shift handoff note include?" \
  --json
```

What just happened:

1. the app loaded local documents

2. it split them into chunks

3. it retrieved relevant chunks for the question

4. it assembled a grounded answer with citations

That is already a real RAG application shape, even though the system is still small.

Do not worry yet about:

1. vector retrieval

2. reranking

3. evaluation

4. the local API and browser UI

Those come later.

The point of this quick start is simple:

get one real result first, then use the rest of the book to understand how the system produced it and how to improve it.

Part 1. Foundations

Chapter 1. What RAG Actually Is

RAG stands for retrieval-augmented generation.

The shortest practical definition is this:

RAG is a pattern where your application looks up relevant source material before asking a model to answer.

That sounds simple, but it changes the shape of the application in an important way. A prompt-only application depends almost entirely on:

1. the model

2. the immediate user input

3. whatever instructions you place in the prompt

A RAG application adds a new moving part:

it must decide what evidence to retrieve before the answer is generated.

That extra step is exactly why RAG is useful. It gives the model access to information that:

1. was not in the user prompt

2. may not be in the model's training data

3. may be too specific, private, or fresh to rely on from memory

Why LLMs Need Retrieval

Large language models are useful because they can turn context into clear answers. They are not a perfectly reliable source for every fact your application may need.

In a practical application, the problem is often not "Can the model write a sentence about this topic?" The problem is "Can the application give the model the right evidence for this specific answer right now?"

About the Author

Blue J. Lion has over 20+ years of experience in software development, with a focus on programming, data security, and privacy. He has worked across engineering and product environments, building practical solutions and tools.

Beyond software, he enjoys creating simple, thoughtful products-ranging from books and visual tools to creative projects that explore the intersection of technology and everyday life.

In his free time, he enjoys running, swimming, and working on new ideas.

Quiet Line Press

Author Portfolio