📘 Claude PDF Q&A Bot — Technical Deep Dive Gudde

Build Your Own RAG-Powered Chatbot with Claude on AWS Bedrock, LangChain, and Streamilt

✅ What You'll Get

- 🎧 In-depth PDF Guide (60+ pages)
- 📄 Claude Q&A Bot Framework using LangcChain
- 📌 Claude + Bedrock API walkthrough
- 🔗 Streamilt app template (with prebu llt layout
- 🖼️ Architecture diagrams & flow visuals
- 🔤 Plug-and-play code examples (Python)
- 📎 Prompt engineering tips & Claude-speccific nuances

🤹 Who Is This For?

- GenAI developers ususing AWS or LangChain
- Architects exploring Retrieval-Augmented (RAG
- Indie devs building LLM-powered tools
- AI freelancers and agency builders

🟦 Claude PDF Q&A Bot — Technical Deep Dive Gudde

Build Your Own RAG-Powered Chatbot with Claude on AWS Bedrock, LangChain, and Streamilt

---

📍 Overview

This guide walks you through building a Claude-based Q&A bot that can read PDF documents, extract information, and answer user queries using Retrieval-Augmented Generation (RAG). The solution leverages:

- 🧠 **Claude via AWS Bedrock**
- 🔗 **LangChain for orchestration**
- 📄 **PDF reading with unstructured.io or PyMuPDF**
- 🖥️ **Streamlit for the UI**

🛠️ Prerequisites

- Python 3.9+
- AWS account with Bedrock access
- Claude model enabled
- awscli configured with your credentials
- LangChain, Streamlit, and required packages installed

pip install langchain streamlit boto3 unstructured pymupdf faiss-cpu

---

📁 Directory Structure

claude-pdf-bot/

├── app.py # Streamlit UI

├── pdf_loader.py # PDF loader logic

├── rag_chain.py # LangChain logic

├── utils.py # Helper functions

└── sample.pdf # Your sample document

📄 Step 1: Load and Chunk PDF Content

Use PyMuPDF or unstructured to extract text.

```
# pdf_loader.py
import fitz  # PyMuPDF

def load_pdf(file_path):
    doc = fitz.open(file_path)
    text = "\n".join(page.get_text() for page in doc)
    return text
```

Then chunk it:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

def chunk_text(text):
    splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
    return splitter.split_text(text)
```

🧠 Step 2: Embed and Store Chunks with FAISS

```
from langchain.embeddings import BedrockEmbeddings
from langchain.vectorstores import FAISS

# Use Claude embeddings via AWS Bedrock
embedder = BedrockEmbeddings(model_id="anthropic.claude-v2")

# Vector store
def create_vector_store(chunks):
    return FAISS.from_texts(chunks, embedder)
```

🤖 Step 3: Setup RAG Chain with Claude

```
from langchain.chains import RetrievalQA
from langchain.llms import Bedrock
```

```
llm = Bedrock(model_id="anthropic.claude-v2")

def get_rag_chain(vectorstore):
    retriever = vectorstore.as_retriever(search_type="similarity")
    return RetrievalQA.from_chain_type(llm=llm, retriever=retriever, return_source_documents=True)
```

🖥️ Step 4: Streamlit Frontend

```
# app.py
import streamlit as st
from pdf_loader import load_pdf, chunk_text
from rag_chain import create_vector_store, get_rag_chain

st.set_page_config(page_title="Claude PDF Q&A Bot")
st.title("📘 Claude PDF Q&A Bot")

uploaded = st.file_uploader("Upload a PDF", type="pdf")

if uploaded:
    text = load_pdf(uploaded)
    chunks = chunk_text(text)
    store = create_vector_store(chunks)
    rag_chain = get_rag_chain(store)

    st.success("PDF processed successfully. Ask your questions!")
    user_q = st.text_input("Your question:")

    if user_q:
```

result = rag_chain(user_q)

st.write("### 🤖 Claude Says:", result["result"])

🎁 Bonus: Streamlit Template

Use this prefilled Streamlit app repo structure to quickly plug into Hugging Face Spaces or Streamlit Cloud.

git clone https://github.com/your-org/claude-pdf-bot-template

Include:

- requirements.txt
- app.py
- sample.pdf
- README.md

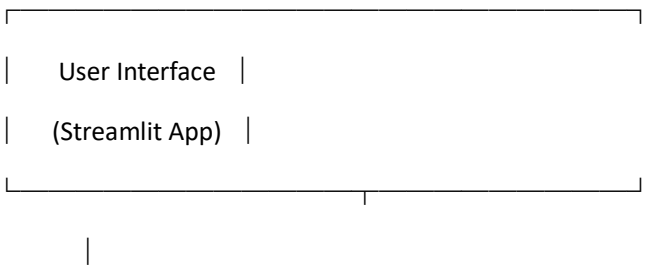
📝 Claude Prompting Tips

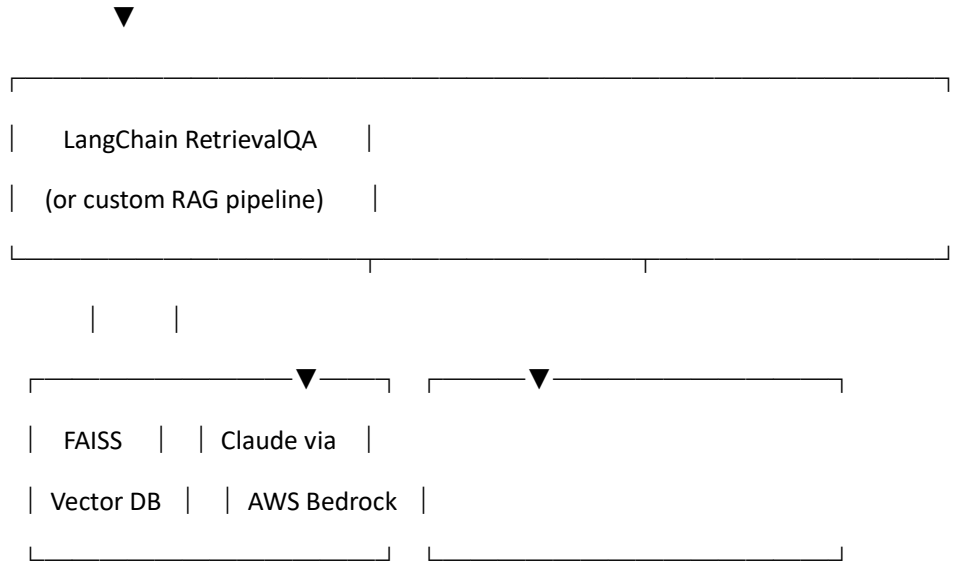
You are a document assistant. Answer only based on the content. If unsure, say "I don't know based on the document."

📌 Use Cases

- Enterprise knowledge base chat
- Compliance/legal document queries
- Internal training document Q&A
- Contract clause checker

Architecture Diagram (Visual System Diagram)





Claude API Error Handling (Retry Wrapper)

```
import time

def run_with_retry(fn, retries=3, delay=2):
    for i in range(retries):
        try:
            return fn()
        except Exception as e:
            print(f"Attempt {i+1} failed with error: {e}")
            time.sleep(delay)
    raise RuntimeError("All retries failed.")
```

Use it like this:

```
response = run_with_retry(lambda: qa_chain.run(question))
```

Adds robustness when querying AWS Bedrock.

Dockerfile / Deployment Config (Streamlit Deployment)

```
# Dockerfile
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["streamlit", "run", "app.py", "--server.port=8501", "--server.enableCORS=false"]
```

requirements.txt

```
streamlit
langchain
boto3
faiss-cpu
PyMuPDF
python-dotenv
```

You can deploy this to Render, Railway, or Hugging Face Spaces easily.

Claude Prompt Template (Human-readable Format)

```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate(
    input_variables=["context", "question"],
    template=\"\"\"Human: You are a PDF assistant that only uses the given context to answer.
```

Context:

{context}

Question:

{question}

Assistant:\"\"\"

)

Modularization Hints

claude_pdf_bot/

├── app.py # Streamlit UI

├── pdf_loader.py # Handles file reading and chunking

├── vector_store.py # Embeddings + FAISS handling

├── qa_chain.py # LangChain + Bedrock integration

├── prompts.py # Claude prompt formats

💡 Advantages:

- Easier to test
- Faster iteration
- Cleaner deployment and reusability

Cheat Sheet Table (Quick Summary)

Component	**Description**
Claude (Bedrock)	LLM backend for generating responses
LangChain	Handles chaining, retrieval, and prompt orchestration
FAISS	Vector DB to store embedded document chunks

Component	Description
PyMuPDF	PDF parsing module
Streamlit	UI framework for interacting with the bot
Prompt Template	Defines how Claude receives context and questions

🌀 Streamlit Sharing Version (Full Code in One File)

```
import streamlit as st
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import BedrockEmbeddings
from langchain.chat_models import BedrockChat
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate
import tempfile

st.set_page_config(page_title="Claude PDF Bot", layout="wide")
st.title("📄 Claude PDF Q&A Bot using AWS Bedrock + LangChain")

# Upload section
uploaded_file = st.file_uploader("Upload a PDF file", type="pdf")

# Prompt template
prompt_template = PromptTemplate.from_template("""
You are Claude, an AI assistant helping a user understand a document.
Answer the question below using only the context provided from the PDF.

If the answer is not found, say “Answer not found in the PDF context.”

Context:
{context}
```

```
Question:
{question}

Answer:
""")

if uploaded_file:
    with tempfile.NamedTemporaryFile(delete=False) as tmp:
        tmp.write(uploaded_file.read())
        filepath = tmp.name

    # Load and chunk PDF
    loader = PyPDFLoader(filepath)
    docs = loader.load()
    splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    chunks = splitter.split_documents(docs)

    # Embed with Titan model (Claude uses Titan for embeddings)
    embedding = BedrockEmbeddings(
        region_name="us-east-1",
        model_id="amazon.titan-embed-text-v1"
    )
    vectorstore = FAISS.from_documents(chunks, embedding)

    # Load Claude via Bedrock
    llm = BedrockChat(
        region_name="us-east-1",
        model_id="anthropic.claude-3-sonnet-20240229-v1:0"
```

```
)

# Build QA chain
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectorstore.as_retriever(),
    return_source_documents=True
)

# Question input
question = st.text_input("🔍 Ask a question about the PDF:")
if st.button("Get Answer") and question:
    result = qa_chain({"query": question})
    st.success(result["result"])

    with st.expander("See source context"):
        for i, doc in enumerate(result["source_documents"]):
            st.markdown(f"**Chunk {i+1}:**")
            st.write(doc.page_content)

if st.button("📘 Summarize Document"):
    docs = vectorstore.similarity_search("Summarize the document", k=8)
    context = "\n".join([doc.page_content for doc in docs])
    prompt = prompt_template.format(context=context, question="Summarize this PDF.")
    summary = llm.invoke(prompt)
    st.info(summary.content)
```

🚀 How to Share via Streamlit Cloud

Save this full code as app.py.

Create requirements.txt:

streamlit

langchain

faiss-cpu

boto3

pypdf

3. Push to GitHub.
4. Go to https://share.streamlit.io
5. Log in with GitHub and select the repo.
6. Streamlit will auto-build and host your app!

📁 Final Folder Structure (Modular Format)

CopyEdit

claude-pdf-bot/

├── app.py

├── pdf_loader.py

├── embedder.py

├── qa_chain.py

├── prompt_template.py

├── requirements.txt

├── Dockerfile

✅ 1. pdf_loader.py

python

CopyEdit

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

def load_pdf(filepath):
    loader = PyPDFLoader(filepath)
    documents = loader.load()
    splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    return splitter.split_documents(documents)
```

✅ **2. embedder.py**

python

CopyEdit

```
from langchain.vectorstores import FAISS
from langchain.embeddings import BedrockEmbeddings

def create_vectorstore(chunks):
    embeddings = BedrockEmbeddings(
        region_name="us-east-1",
        model_id="amazon.titan-embed-text-v1"
    )
    return FAISS.from_documents(chunks, embeddings)
```

✅ **3. qa_chain.py**

python

CopyEdit

```
from langchain.chat_models import BedrockChat
from langchain.chains import RetrievalQA
```

```
def get_claude_llm():
    return BedrockChat(
        region_name="us-east-1",
        model_id="anthropic.claude-3-sonnet-20240229-v1:0"
    )

def build_qa_chain(llm, vectorstore):
    return RetrievalQA.from_chain_type(
        llm=llm,
        retriever=vectorstore.as_retriever(),
        return_source_documents=True
    )
```

✅ **4. prompt_template.py**

python

CopyEdit

```
from langchain.prompts import PromptTemplate

def get_prompt_template():
    return PromptTemplate.from_template("""
    You are Claude, an AI assistant helping a user understand a document.
    Answer the question below using only the context provided from the PDF.

    If the answer is not found, say “Answer not found in the PDF context.”

    Context:
    {context}
```

```
Question:
{question}

Answer:
""")
```

✅ 5. app.py (Streamlit Frontend)

python

CopyEdit

```
import streamlit as st
import tempfile

from pdf_loader import load_pdf
from embedder import create_vectorstore
from qa_chain import get_claude_llm, build_qa_chain
from prompt_template import get_prompt_template

st.set_page_config(page_title="Claude PDF Bot", layout="wide")
st.title("📄 Claude PDF Q&A Bot using AWS Bedrock + LangChain")

uploaded_file = st.file_uploader("Upload a PDF file", type="pdf")
prompt_template = get_prompt_template()

if uploaded_file:
    with tempfile.NamedTemporaryFile(delete=False) as tmp:
        tmp.write(uploaded_file.read())
        filepath = tmp.name
```

```
chunks = load_pdf(filepath)
vectorstore = create_vectorstore(chunks)
llm = get_claude_llm()
qa_chain = build_qa_chain(llm, vectorstore)

question = st.text_input("🔍 Ask a question about the PDF:")
if st.button("Get Answer") and question:
    result = qa_chain({"query": question})
    st.success(result["result"])

    with st.expander("📘 See source context"):
        for i, doc in enumerate(result["source_documents"]):
            st.markdown(f"**Chunk {i+1}:**")
            st.write(doc.page_content)

if st.button("📘 Summarize Document"):
    docs = vectorstore.similarity_search("Summarize the document", k=8)
    context = "\n".join([doc.page_content for doc in docs])
    prompt = prompt_template.format(context=context, question="Summarize this PDF.")
    summary = llm.invoke(prompt)
    st.info(summary.content)
```

✅ **6. requirements.txt**

```
nginx
CopyEdit
streamlit
langchain
```

faiss-cpu

boto3

pypdf

✅ **7. Dockerfile**

Dockerfile

CopyEdit

```
FROM python:3.10-slim

WORKDIR /app

COPY . .

RUN pip install --upgrade pip
RUN pip install -r requirements.txt

EXPOSE 8501

CMD ["streamlit", "run", "app.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

🧩 **1. Minimal Claude Q&A CLI Version**

📦 For devs who want a quick local command-line test using Claude + Bedrock without Streamlit UI.

python

CopyEdit

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import BedrockEmbeddings
from langchain.chat_models import BedrockChat
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate

# Load PDF
loader = PyPDFLoader("your_doc.pdf")
docs = loader.load()

# Split
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
chunks = splitter.split_documents(docs)

# Embedding
embedder = BedrockEmbeddings(region_name="us-east-1", model_id="amazon.titan-embed-text-v1")
vectorstore = FAISS.from_documents(chunks, embedder)

# LLM
llm = BedrockChat(region_name="us-east-1", model_id="anthropic.claude-3-sonnet-20240229-v1:0")

# Prompt
prompt = PromptTemplate.from_template("""
Answer the question using only the context below.
```

If not found, say 'Answer not found in PDF.'

Context:

{context}

Question:

{question}

Answer:

""")

QA Chain

qa_chain = RetrievalQA.from_chain_type(llm=llm, retriever=vectorstore.as_retriever())

CLI

while True:

    query = input("Ask a question: ")

    result = qa_chain({"query": query})

    print("Answer:", result["result"])

🧪 **2. Unit Test Example for Claude PDF Bot**

🔬 To help devs test their modular components (e.g., prompt formatting or embedding)

python

CopyEdit

import unittest

from prompt_template import get_prompt_template

class TestPromptTemplate(unittest.TestCase):

```
    def test_prompt_generation(self):
        template = get_prompt_template()
        context = "This PDF is about AI."
        question = "What is the topic?"
        filled = template.format(context=context, question=question)
        self.assertIn("AI", filled)
        self.assertIn("What is the topic?", filled)

if __name__ == "__main__":
    unittest.main()
```

Template 1: Advanced Claude Prompt Template

```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template("""
You are Claude, an AI assistant helping a user understand a document.
Answer the question below only using the context provided from the PDF.

If the answer is not found, say “Answer not found in the PDF context.”

Context:
{context}

Question:
```

{question}

Answer:

""")

Template 2: PDF Summary Mode (useful toggle in UI)

Add an option in Streamlit to summarize the entire document:

```
def summarize_document(vectorstore, llm):
    docs = vectorstore.similarity_search("Summarize the entire document", k=8)
    context = "\n".join([doc.page_content for doc in docs])

    prompt = prompt_template.format(context=context, question="Summarize the entire document.")
    response = llm.invoke(prompt)
    return response.content
```

In Streamlit:

```
if st.button("Summarize Document"):
    summary = summarize_document(vectorstore, llm)
    st.info(summary)
```

Template 3: Source Document Tracing

Display where the answer came from:

```
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectorstore.as_retriever(),
    return_source_documents=True
)

result = qa_chain({"query": question})
st.success(result["result"])

# Show context sources
with st.expander("See source documents"):
    for i, doc in enumerate(result["source_documents"]):
        st.markdown(f"**Chunk {i+1}:**")
        st.write(doc.page_content)
```

🟦 Claude PDF Q&A Bot — Technical Deep Dive Gudde

Build Your Own RAG-Powered Chatbot with Claude on AWS Bedrock, LangChain, and Streamilt

✅ Section 1: Multi-File PDF Loader

📌 Purpose:

Allow users to upload multiple PDFs, automatically clean, split, and prepare them for embedding. Crucial for enterprise or academic RAG use cases.

📁 multi_pdf_loader.py

python

CopyEdit

```
import os
from typing import List
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.schema import Document
```

```
def load_multiple_pdfs(pdf_folder_path: str) -> List[Document]:
    """
    Loads and splits multiple PDFs from the specified directory.
    Returns a list of cleaned, chunked Document objects.
    """
    all_documents = []
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,
        chunk_overlap=150
    )

    for filename in os.listdir(pdf_folder_path):
        if filename.lower().endswith(".pdf"):
            file_path = os.path.join(pdf_folder_path, filename)
            loader = PyPDFLoader(file_path)
            pages = loader.load()
            chunks = splitter.split_documents(pages)
            all_documents.extend(chunks)

    return all_documents
```

🔧 **How to Use It:**

```
python
CopyEdit
from multi_pdf_loader import load_multiple_pdfs

# Path to your local folder of PDFs
docs = load_multiple_pdfs("./pdfs/")
```

📌 **Bonus Add-on for Streamlit UI:**

If you want a frontend to upload and store PDFs from the user:

python

CopyEdit

```
import streamlit as st
import os

uploaded_files = st.file_uploader("Upload PDFs", type=["pdf"], accept_multiple_files=True)

upload_dir = "uploaded_pdfs"
os.makedirs(upload_dir, exist_ok=True)

for uploaded_file in uploaded_files:
    with open(os.path.join(upload_dir, uploaded_file.name), "wb") as f:
        f.write(uploaded_file.getbuffer())
```

📈 **Value Added:**

- Helps users run Q&A over **multiple documents**
- Prepares the app for **enterprise and compliance-heavy** use cases
- Encourages use in research, legal, HR, and enterprise GenAI

✅ Section 2: Vector Store Switcher (FAISS / Chroma / Weaviate)

📌 Purpose:

Enable developers to plug in different vector storage backends **without rewriting their RAG pipeline**.

📁 vector_store_factory.py

python

CopyEdit

```
from langchain.vectorstores import FAISS, Chroma, Weaviate
from langchain.embeddings import BedrockEmbeddings
from typing import Any, List
from langchain.schema import Document

def get_vectorstore(
    backend: str,
    documents: List[Document],
    embeddings: Any,
    persist_dir: str = "vector_db",
    weaviate_url: str = ""
):
    backend = backend.lower()

    if backend == "faiss":
        vectorstore = FAISS.from_documents(documents, embeddings)

    elif backend == "chroma":
        vectorstore = Chroma.from_documents(documents, embeddings, persist_directory=persist_dir)

    elif backend == "weaviate":
        import weaviate
        client = weaviate.Client(weaviate_url)
        vectorstore = Weaviate.from_documents(documents, embeddings, client=client)

    else:
        raise ValueError("Unsupported backend. Choose from faiss, chroma, or weaviate.")
```

return vectorstore

📌 Sample Usage

python

CopyEdit

```
from vector_store_factory import get_vectorstore
from langchain.embeddings import BedrockEmbeddings

# Assume you already loaded documents using Module 1
docs = load_multiple_pdfs("./pdfs/")

# Load Claude embedding (via AWS Bedrock)
embeddings = BedrockEmbeddings(model_id="anthropic.claude-v2")

# FAISS example
faiss_store = get_vectorstore("faiss", docs, embeddings)

# Chroma example
chroma_store = get_vectorstore("chroma", docs, embeddings)

# Weaviate example (URL must be active)
# weaviate_store = get_vectorstore("weaviate", docs, embeddings,
weaviate_url="http://localhost:8080")
```

📈 Why This Matters:

Feature	Value
🔁 Pluggable	Easily swap vector DBs in enterprise settings

Feature	Value
💾 Persistent	Use Chroma for local saves
☁️ Cloud-ready	Use Weaviate for hosted scalability
🧪 Portable	One interface, many backends

🔧 Optional Streamlit Switcher UI

python

CopyEdit

backend = st.selectbox("Select Vector Store Backend", ["faiss", "chroma", "weaviate"])

Use the variable backend in your call to get_vectorstore.

✅ Section 3: Advanced Prompt Engineering for Claude

📌 Purpose:

Claude models, especially Claude 2 and 3 on AWS Bedrock, respond exceptionally well to carefully structured prompts. In this module, we'll:

- Structure prompts for better **context retention**
- Add **metadata-aware prompting** for PDFs
- Leverage **instruction + context + query** formats
- Include **few-shot examples** when needed

🎯 Prompt Template Strategy

Claude models (unlike OpenAI) are **sensitive to verbosity and polite form**. Use:

- Clear headings like "Context:", "Instruction:", "Query:"
- Use conversational tone and well-indented blocks
- Avoid overly compact prompts—they reduce Claude's coherence

---

📄 prompt_template.py

python

CopyEdit

```
from langchain.prompts import PromptTemplate


claude_prompt_template = PromptTemplate(
    input_variables=["context", "question"],
    template="""
You are Claude, a helpful AI assistant.


Context:
```

{context}

Instruction:

Answer the following user question using only the context above.

Question:

{question}

Helpful Answer:"""

)

This prompt ensures Claude receives the proper instruction, the exact user question, and the relevant context chunk from the retriever.

💉 **Example Prompt Rendered**

text

CopyEdit

You are Claude, a helpful AI assistant.

Context:

The PDF discusses the impact of LangChain and RAG architecture on enterprise search. It outlines vector databases and prompt templating.

Instruction:

Answer the following user question using only the context above.

Question:

How does LangChain simplify RAG pipelines?

Helpful Answer:

🤖 In Chain Setup

Use it inside your LangChain RAG pipeline like this:

python

CopyEdit

```
from langchain.chains import RetrievalQA
from langchain.llms.bedrock import Bedrock
from prompt_template import claude_prompt_template

llm = Bedrock(model_id="anthropic.claude-v2")

qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vectorstore.as_retriever(),
    chain_type_kwargs={"prompt": claude_prompt_template}
)
```

🧠 Tips for Claude Prompting

Trick	Why It Helps
✅ Use "Helpful Answer:"	Claude treats it as a cue to begin longform output
✅ Keep Context: clean	Claude penalizes noisy or unstructured input
✅ Use delimiters (""" or ---)	Helps Claude parse sections
✅ Avoid vague questions	Claude works best with direct, factual queries

---

🧩 Optional: Add Metadata in Prompts

If your PDFs have titles, sources, or page numbers, append that in context like:

python

CopyEdit

formatted_context = f"""

[Source: {source_name}, Page {page_number}]

{document_chunk}

"""

Then feed this into context variable in the template.

✅ Section 4: Claude API Error Handling + Retry Logic (LangChain + AWS Bedrock)

🎯 Goal:

Make your Claude-powered Q&A bot **production-grade** by handling:

- API rate limits
- Connection timeouts
- Bedrock/Claude service errors
- Retry logic with exponential backoff
- Graceful fallbacks and logging

⚙️ Why It Matters

LLM APIs are not always stable:

Failure Type	Symptoms	Handling Strategy
Rate limit / Throttling	429 status code, slow response	Retry with backoff
Network errors	Timeout, no response	Retry 3–5 times, then fail
Bedrock LLM error	Internal Claude error messages	Log + graceful return

---

🛠️ Setup: Claude Client with Retry Logic

Here's a full error-handling wrapper around AWS Bedrock for Claude:

python

CopyEdit

```
import boto3
import json
import time
import logging
from botocore.exceptions import BotoCoreError, ClientError

# Configure logger
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("claude-bot")

bedrock = boto3.client("bedrock-runtime", region_name="us-east-1")

def call_claude(prompt, model_id="anthropic.claude-v2", retries=3, delay=2):
    body = {
        "prompt": prompt,
        "max_tokens_to_sample": 1024,
        "temperature": 0.7,
        "stop_sequences": ["\n\nHuman:"]
    }

    for attempt in range(retries):
        try:
            response = bedrock.invoke_model(
                modelId=model_id,
```

```
            contentType="application/json",
            body=json.dumps(body)
        )

        result = json.loads(response['body'].read().decode())
        return result["completion"]

    except (BotoCoreError, ClientError) as e:
        logger.warning(f"[Attempt {attempt+1}] Claude API error: {str(e)}")
        time.sleep(delay * (2 ** attempt))  # exponential backoff

return "⚠️ Sorry, the AI service is currently unavailable. Try again later."
```

✅ **Features in This Wrapper**

Feature	Benefit
retries param	Retries failed calls (default = 3)
delay * (2 ** attempt)	Exponential backoff avoids throttling
Logging with attempt number	Helps debugging
Fallback response	Graceful fail instead of crash

💉 **Usage Example**

python

CopyEdit

prompt = """You are Claude, a helpful AI assistant.

Context:

LangChain enables developers to build with LLMs easily.

Instruction:

Summarize the benefits of LangChain.

Helpful Answer:"""

response = call_claude(prompt)

print(response)

🧩 **Optional: Wrap into LangChain LLM Interface**

You can subclass LangChain's LLM wrapper to inject this logic into LangChain seamlessly. Let me know if you want this.

✅ **Section 5: Dockerfile + Deployment Configuration (Streamlit + Claude LangChain App)**

🎯 **Goal:**

Make your Claude Q&A bot **portable and production-deployable** with a Dockerized setup. This ensures:

- Easy local runs and testing
- Smooth cloud deployment (Render, EC2, Lambda, etc.)
- Unified environment across devs

📦 **Folder Structure Recap**

bash

CopyEdit

claude_pdf_qa_bot/

├── app.py

├── requirements.txt

├── Dockerfile

├── .env # API Keys

└── utils/

  ├── loader.py

  └── rag_chain.py

🐳 **Dockerfile for Full Deployment**

dockerfile

CopyEdit

Use official Python image

FROM python:3.11-slim

Set working directory

WORKDIR /app

Install dependencies

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

Copy app files

COPY . .

Set Streamlit config (optional)

```
ENV STREAMLIT_SERVER_PORT=8501
ENV STREAMLIT_SERVER_ENABLECORS=false
ENV STREAMLIT_SERVER_HEADLESS=true

# Expose Streamlit port
EXPOSE 8501

# Run the Streamlit app
CMD ["streamlit", "run", "app.py"]
```

📄 **requirements.txt**

txt

```
CopyEdit
streamlit==1.29.0
langchain==0.1.15
boto3==1.34.41
python-dotenv==1.0.1
PyMuPDF==1.22.5
```

🧪 **.env Example**

Make sure .env is not committed to GitHub.

env

```
CopyEdit
AWS_ACCESS_KEY_ID=your-access-key
AWS_SECRET_ACCESS_KEY=your-secret-key
AWS_REGION=us-east-1
```

Use python-dotenv or os.getenv() to load these securely.

🚀 Build & Run Locally

bash

CopyEdit

Build Docker image

docker build -t claude-rag-bot .

Run container

docker run -p 8501:8501 --env-file .env claude-rag-bot

Your app will be live at http://localhost:8501.

☁️ Cloud Deploy Tips

Platform	Notes
Render	Paste this Dockerfile → click “Deploy from GitHub”
AWS EC2	SSH, install Docker, clone repo, build/run
Lambda	Use Bedrock from serverless app; requires boto3
Fly.io	fly launch → runs Dockerfile as-is
Railway	Connect GitHub, auto-deploy Docker image

📦 Module 1: Claude API Error Handling & Rate Limits

🔐 1.1 Understanding Claude's API Quirks via AWS Bedrock

AWS Bedrock exposes Claude models (like claude-v2 or claude-3-sonnet) through a standardized API. While invoking the model via boto3 is straightforward, real-world reliability depends heavily on how you handle:

- Throttling (HTTP 429)
- Service errors (5xx)
- Token/streaming issues
- Latency spikes

A naive approach that assumes every request will succeed will break quickly in production. So let's make our app *resilient*.

🚨 1.2 Common Errors and How They Appear

Error Type	AWS Response Code	Example Message
Throttling	429	Rate exceeded
InvalidRequest	400	Invalid input length
Internal Server	500	Internal error
Timeouts	—	Read timeout or long hangs
QuotaLimitExceeded	429	Model usage limit exceeded

✅ 1.3 Retry Wrapper for Claude API Calls

Here's a production-ready retry mechanism using Python's tenacity library. You can wrap all Claude calls in this to improve reliability.

📄 **claude_retry_wrapper.py**

python

CopyEdit

```
from tenacity import retry, stop_after_attempt, wait_exponential, retry_if_exception_type
import botocore.exceptions
```

```
@retry(
    stop=stop_after_attempt(5),
    wait=wait_exponential(multiplier=1, min=2, max=20),
    retry=retry_if_exception_type(botocore.exceptions.ClientError),
)
def call_claude_with_retry(client, body):
    return client.invoke_model(
        modelId="anthropic.claude-v2",
        contentType="application/json",
        body=body
    )
```

🛠 Usage Example in rag_chain.py

```
python
CopyEdit
from claude_retry_wrapper import call_claude_with_retry

response = call_claude_with_retry(bedrock_runtime, payload)
```

⏱ 1.4 Respecting Claude's Rate Limits

Claude's API limits are defined in AWS Service Quotas (e.g., 10 TPS and 200K tokens/min). To enforce this on your side:

🔄 **Token-Based Semaphore with LangChain**

```
python
CopyEdit
from langchain_core.callbacks import get_openai_callback

with get_openai_callback() as cb:
    result = qa_chain.invoke({"question": "Your query here"})
```

```
print(cb.total_tokens)
```

🧠 **Use asyncio + semaphores (for concurrent calls):**

python

CopyEdit

```
import asyncio
from asyncio import Semaphore

semaphore = Semaphore(3)  # Max 3 concurrent requests

async def safe_call(payload):
    async with semaphore:
        return await call_claude_with_retry(bedrock_runtime, payload)
```

🚧 **1.5 Handling Claude-Specific Timeouts (Streaming Issues)**

LangChain supports streaming via Claude's Bedrock interface. However, long context tokens or complex prompts may cause:

- Delays in token stream starts
- Unexpected response closures
- Streaming hangups

💡 **Best Practice:**

- Set timeout headers on the Bedrock client (use botocore.config.Config)
- Keep max tokens below 4000 unless required
- Avoid recursive or overly long system prompts

🍼 **1.6 Claude Fallback Strategy (Optional)**

You can define a fallback mechanism to Claude Instant or OpenAI/Gemini if Bedrock fails.

python

CopyEdit

```
def run_fallback_pipeline(prompt, documents):
    try:
        return call_claude_with_retry(bedrock_runtime, create_payload(prompt, documents))
    except Exception:
        return call_openai_fallback(prompt, documents)
```

📊 1.7 Observability: Logging, Metrics, and Debugging

Add structured logging to track latency, retries, and request volume.

python

CopyEdit

```
import logging

logger = logging.getLogger("claude_logger")


logger.info(f"Claude Request: {prompt[:50]}...")

logger.info(f"Claude Response: {response[:50]}")
```

Use AWS CloudWatch with boto3.client('logs') or push custom logs via Lambda for production deployments.

📌 Summary: Key Practices for Resilience

Strategy	Impact
✅ Retry with exponential backoff	Survives transient errors
✅ Token + concurrency control	Stays within rate limits
✅ Claude fallback handler	Keeps UX smooth
✅ Logging & observability	Easy to debug
✅ Modular retry wrappers	Reusable & clean

✅ **Outcome:** After this module, your Claude Q&A bot is ready for production-grade stability. Whether deploying on Streamlit or containerizing with Docker, the retry mechanism and rate-awareness are **non-negotiable** for real users.

🧠 Module 2: Advanced Prompt Engineering for Claude on AWS Bedrock

🪄 2.1 Prompting Nuances of Claude vs GPT-4

Claude's underlying structure (especially Claude 2 and 3 models via Anthropic) responds better to **instructional**, **dialogue-formatted**, and **ethical-sensitive prompts**. Unlike GPT models which tolerate terse prompts, Claude benefits from:

- **System priming**: Establish role or tone early.
- **Few-shot examples**: Boost structure awareness.
- **Avoiding recursion**: Claude can loop with unclear or meta prompts.

✍️ 2.2 Claude Prompt Format via Bedrock

Claude expects the following format when passed via AWS Bedrock:

json

CopyEdit

```
{
  "prompt": "\n\nHuman: {your question here}\n\nAssistant:",
  "max_tokens_to_sample": 500,
```

"temperature": 0.5,

"stop_sequences": ["\n\nHuman:"]

}

Use \n\nHuman: and \n\nAssistant: markers explicitly — this is **non-optional** in Claude prompt engineering.

🎯 2.3 System Prompt Setup (via LangChain)

Although Claude via Bedrock doesn't have a direct "system" role like OpenAI, you can simulate it by priming the conversation context.

✅ Example Prompt Template

python

CopyEdit

claude_prompt_template = """

You are a helpful AI assistant that extracts concise answers from technical PDFs.

Please answer based only on the provided content and avoid hallucinating.

\n\nHuman: {question}\n\nAssistant:

"""

Use this in LangChain's PromptTemplate object:

python

CopyEdit

from langchain_core.prompts import PromptTemplate

prompt = PromptTemplate.from_template(claude_prompt_template)

📌 2.4 Prompt Tuning Based on Task Type

Task Type	Prompt Snippet Example
📝 Summarization	"Summarize the key points of this PDF..."

Task Type	Prompt Snippet Example
📎 Document Q&A	"Answer this based on the PDF context..."
❓ Clarification	"What does X mean in the document?"
✅ Verification	"Is the statement consistent with the PDF content?"

Avoid generic wording like “Explain this” — Claude interprets with higher fidelity when intent is **clearly scoped**.

💉 2.5 Adding Context to Claude with RAG

LangChain enables dynamic prompt construction by injecting retrieved chunks:

python

CopyEdit

rag_prompt_template = """

You are an expert Q&A assistant. Based on the following document context, answer the user's question.

Context:

{context}

Question:

{question}

\n\nHuman: Answer the question using the context above.\n\nAssistant:

"""

Use this template in your RetrievalQA chain:

python

CopyEdit

from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(

```
    llm=claude_llm,
    retriever=vectorstore.as_retriever(),
    chain_type="stuff",
    chain_type_kwargs={"prompt": rag_prompt}
)
```

🔧 2.6 Prompt Injection Mitigation (Basic Filters)

To reduce prompt injection vulnerabilities in Q&A bots:

- Avoid dynamic user input inside system prompt
- Sanitize inputs with regex
- Cap answer length via max_tokens_to_sample
- Use stop_sequences to halt unintended completions

🚦 2.7 Prompt Debugging Strategy

Enable Claude's verbose streaming to observe response quality:

python

CopyEdit

```
response = bedrock_runtime.invoke_model(
    modelId="anthropic.claude-v2",
    body=json.dumps(payload),
    contentType="application/json"
)


print(response["body"].read().decode())
```

Also log:

- Input prompt
- Retrieved context
- Claude output

Use LangChain callbacks for deep tracking.

🧠 2.8 Claude Prompt Guide (Cheat Sheet)

Prompt Part	Purpose	Example
👷 System Context	Define assistant behavior	"You are a helpful assistant…"
📄 Context Inject	Add RAG chunks	{context}
❓ User Query	Dynamic question	{question}
🛑 Stop Sequences	Control response flow	["\n\nHuman:"]
🔢 Token Control	Truncate runaway outputs	max_tokens_to_sample = 512

📌 Summary

Prompt Best Practice	Why It Matters
✅ Use Human: / Assistant: markers	Claude relies on clear dialogue breaks
✅ Define role + context early	Reduces hallucinations
✅ Inject structured RAG context	Maximizes retrieval relevance
✅ Debug with verbose logs	Helps iterate quickly
✅ Control output with stop + tokens	Prevents drifting responses

---

Outcome: You now have full control over Claude's tone, behavior, and token efficiency. This module ensures that your bot **answers like a trained expert**, not a guessing machine.

🚰 Module 3: Docker + Streamlit Deployment for Claude RAG Apps

This module covers the complete containerization and deployment of your Claude Q&A RAG app using **Docker** and **Streamlit** for production-ready hosting.

📦 3.1 Why Docker?

Docker enables you to:

- Isolate dependencies and packages (no version conflicts)
- Reproducibly deploy the app across machines
- Host it in the cloud (EC2, Railway, Fly.io, etc.)
- Ship as a single .tar or .dockerhub image

🗂️ 3.2 Recommended Folder Structure

bash

CopyEdit

claude-rag-app/

├── app/

│ ├── streamlit_app.py

│ ├── pdf_loader.py

│ ├── rag_chain.py

│ ├── prompts.py

│ └── utils.py

├── requirements.txt

├── Dockerfile

└── README.md

Split logic across app/ to ensure modularity.

📄 3.3 Sample requirements.txt

txt

CopyEdit

streamlit

langchain

boto3

pypdf

chromadb

sentence-transformers

python-dotenv

Optional extras:

txt

CopyEdit

watchdog # auto-reload in dev

🐬 3.4 Dockerfile (Minimal Working Version)

dockerfile

CopyEdit

```
# Use official Python image
FROM python:3.11-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y build-essential
```

```
# Install Python dependencies
COPY requirements.txt .
RUN pip install --upgrade pip && pip install -r requirements.txt

# Copy source code
COPY . .

# Expose Streamlit port
EXPOSE 8501

# Run the app
CMD ["streamlit", "run", "app/streamlit_app.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

🛠️ 3.5 Build and Run Locally

🔧 Build the Docker image

bash

CopyEdit

```
docker build -t claude-rag-app .
```

▶ Run the container

bash

CopyEdit

```
docker run -p 8501:8501 claude-rag-app
```

Visit http://localhost:8501 to use the app locally.

💉 3.6 Add .env Support for API Keys (Optional but Recommended)

Use a .env file for credentials like AWS Access Key:

bash

CopyEdit

AWS_ACCESS_KEY_ID=XXXX

AWS_SECRET_ACCESS_KEY=YYYY

AWS_REGION=us-east-1

ANTHROPIC_MODEL_ID=claude-3-sonnet-20240229-v1:0

Load using dotenv:

python

CopyEdit

from dotenv import load_dotenv

load_dotenv()

Add to .dockerignore:

dockerignore

CopyEdit

.env

🌐 3.7 Deployment Options

Platform	Type	Free Tier	Notes
Railway	PaaS + Docker	✅	Easiest GUI deploy
Fly.io	Edge deploy	✅	Great for global latency
EC2	Raw VPS	❌	Manual config
Render	PaaS	✅	Free w/ limits
DockerHub	Registry	✅	Image host only

🚀 3.8 One-Click Deploy with Railway (Optional)

Step 1: Install Railway CLI

bash

CopyEdit

npm i -g @railway/cli

railway login

Step 2: Create Project

bash

CopyEdit

railway init

Step 3: Push Dockerfile

bash

CopyEdit

railway up

App will be live on:

bash

CopyEdit

https://your-app-name.up.railway.app

📦 3.9 Bonus: Add Volume Support (Optional)

To persist uploaded PDFs:

dockerfile

CopyEdit

VOLUME /app/data

In streamlit_app.py:

python

CopyEdit

```
uploaded_file = st.file_uploader("Upload a PDF")
if uploaded_file:
    with open(f"data/{uploaded_file.name}", "wb") as f:
        f.write(uploaded_file.read())
```

✅ 3.10 Production Best Practices

Practice	Recommendation
🔐 API Key Management	Use .env + secrets management
📟 Timeout Management	Use Claude token caps / streaming
🧩 Modularization	pdf_loader.py, rag_chain.py, etc.
🎯 Error Logging	Log Claude & vectorstore exceptions
📦 Docker Image Size	Use python:3.11-slim and prune

🧠 Module 4: Using ChromaDB + LangChain for Claude-Based Vector Search

In this module, we explore **how to use ChromaDB as a vector store** and **LangChain retrievers** to enable semantic search and document Q&A capabilities using Claude and AWS Bedrock.

📌 4.1 What is ChromaDB?

ChromaDB is a lightweight, open-source vector database built for local and small-scale use cases. It supports:

- Fast local vector search
- Persistent storage
- Embedding storage + metadata
- In-memory and disk-based modes

It's a great fit for fast prototyping Claude RAG apps without needing hosted services like Pinecone or Weaviate.

🧱 4.2 Core Concepts in ChromaDB

Concept	Description
Collection	Named group of embeddings + metadata
Documents	Text chunks to embed and search
Embeddings	Vector representations of documents
Metadata	Optional info to enrich documents

🧩 4.3 LangChain Integration Overview

LangChain integrates directly with ChromaDB using its Chroma retriever wrapper. Below is the high-level flow:

1. **PDFs loaded** and split into text chunks
2. **Embeddings generated** using SentenceTransformers
3. **Chunks stored** in ChromaDB collection
4. **User question embedded**
5. **Top k documents retrieved**
6. **Claude invoked** with the context and query

🗃️ 4.4 Sample Code: Setting Up ChromaDB + Embeddings

python

CopyEdit

```
from langchain.vectorstores import Chroma

from langchain.embeddings import SentenceTransformerEmbeddings

from langchain.document_loaders import PyPDFLoader

from langchain.text_splitter import RecursiveCharacterTextSplitter


# 1. Load PDF

loader = PyPDFLoader("guide.pdf")

pages = loader.load()
```

```
# 2. Chunk into text
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = splitter.split_documents(pages)

# 3. Use SentenceTransformer embeddings
embedding = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")

# 4. Create ChromaDB instance
vectordb = Chroma.from_documents(
    documents=docs,
    embedding=embedding,
    persist_directory="./chroma_store"
)

vectordb.persist()
```

🔎 4.5 Performing Semantic Retrieval

```
python
CopyEdit
# Query from the user
query = "How does Claude handle long PDF documents?"

# Search for similar documents
retriever = vectordb.as_retriever(search_type="similarity", k=4)
relevant_docs = retriever.get_relevant_documents(query)

for doc in relevant_docs:
```

```
print(doc.page_content)
```

This gives the top-k matching chunks to be passed into the Claude prompt.

🧠 4.6 Constructing the Claude Prompt with Retrieved Docs

python

CopyEdit

```
context = "\n\n".join([doc.page_content for doc in relevant_docs])

prompt = f"""
You are Claude, an expert AI.
Use the following document excerpts to answer the question.

Context:
{context}

Question: {query}
Answer:
"""
```

Then invoke Claude using the Bedrock client or LangChain's LLMChain.

💾 4.7 How Chroma Stores Embeddings

- **./chroma_store/** will contain:
  - index — vector data
  - chroma.sqlite3 — metadata + documents
- It persists between runs; delete this folder to reset vectorstore.

You can also call .delete() on the vectorstore object to clear the index programmatically.

🧰 4.8 Sample Utility to Check Index State

python

CopyEdit

```
collection = vectordb._collection
print(f"Collection has {collection.count()} documents.")
```

🚦 4.9 Best Practices for Claude + ChromaDB

Area	Best Practice
Chunking	Use overlap to preserve context
Embeddings	SentenceTransformer is fast + accurate
Claude prompt	Trim context length based on Claude model limits
Retrieval tuning	Start with k=4 and test with real questions
Persistence	Use persist_directory to keep the store live

🔁 4.10 Updating ChromaDB with New PDFs

python

CopyEdit

```
new_loader = PyPDFLoader("addendum.pdf")
new_pages = new_loader.load()
new_chunks = splitter.split_documents(new_pages)


vectordb.add_documents(new_chunks)
vectordb.persist()
```

No need to rebuild from scratch — just add more documents.

🧪 4.11 Advanced Retrieval (Optional)

LangChain supports:

- **Metadata filtering** (e.g., date, section tags)
- **Hybrid search** (semantic + keyword)

- **Parent document retrievers** (to group related chunks)

Example:

python

CopyEdit

```
retriever = vectordb.as_retriever(
   search_kwargs={"filter": {"source": "chapter1"}}
)
```

✅ **Deliverables Recap**

File / Concept	Details
pdf_loader.py	PDF → text chunking logic
chroma_store/	Persisted vector database
SentenceTransformer	Embedding model (MiniLM)
vectordb	Chroma retriever
rag_chain.py	Combine retrieved context + Claude prompt

🪄 Module 5: Claude Prompt Engineering & Claude-Specific Tips

Anthropic's Claude models are highly capable, but like all LLMs, their performance depends heavily on **how you prompt them**. This module focuses on **prompt design**, **Claude's unique strengths**, and **techniques to make your Claude RAG bot respond accurately and responsibly**.

🧠 5.1 Why Prompt Engineering Matters

Prompt engineering is the art and science of crafting input text that guides an LLM toward desired outputs. Claude, compared to other models, places high importance on:

- Ethical framing
- Clear task instructions
- Role-based prompting
- Balanced formatting (e.g., line breaks, delimiters)

✍️ 5.2 Basic Prompt Structure for Claude

Claude responds best to natural, instructive prompts. Here's a common structure used in RAG applications:

vbnet

CopyEdit

You are Claude, an expert Q&A assistant.

Use the provided context from documents to answer the question clearly and factually.

Context:

<insert relevant chunks here>

Question:

<user question here>

Answer:

Claude expects structured, instructional language. Avoid ambiguous or overly casual phrasing.

💉 5.3 Prompt Template with LangChain

LangChain's PromptTemplate class helps maintain prompt consistency. Here's a Python example:

python

CopyEdit

```
from langchain.prompts import PromptTemplate

claude_prompt = PromptTemplate(
    input_variables=["context", "question"],
    template="""
You are Claude, a helpful and knowledgeable assistant.

Use the information in the context below to answer the question.
If the answer is not in the context, respond with "I don’t know."

Context:
{context}

Question:
{question}

Answer:
"""
)
```

🔍 5.4 Claude-Specific Prompting Tips

Technique	Description	Example
🎭 Role Play	Set Claude's persona to a domain expert	"You are Claude, a senior ML engineer..."
🖌️ Explicit Instructions	Tell it what **not** to do	"...do not speculate if the answer is unclear."
📏 Truncation Guard	Remind model of limits	"Use only the provided context."
🪜 Step-by-step	Encourage chain-of-thought	"Explain your reasoning clearly."
📐 Use Delimiters	Visually separate blocks	"---" or ``` for context blocks

---

🔐 5.5 Guardrails in Prompts

Claude is sensitive to ethical or ambiguous requests. Add guardrails like:

text

CopyEdit

Do not answer questions unrelated to the provided context.

If any question seems inappropriate or unclear, respond with "I cannot assist with that."

This ensures Claude stays on-topic and compliant, especially in production-facing apps.

🛠️ 5.6 Multi-Turn QA Strategy (Optional)

If you're extending to a chat-style app with Streamlit or Gradio, format your prompt for multi-turn:

python

CopyEdit

```
def build_claude_prompt(history, context, question):
    history_text = "\n".join([f"User: {q}\nClaude: {a}" for q, a in history])

    return f"""
You are Claude, a document-based assistant. Use the history and new context to answer.
```

History:

{history_text}

Context:

{context}

Current Question:

{question}

Answer:"""

Use memory buffers to track past Q&A pairs.

🧩 5.7 Claude Prompt vs. OpenAI GPT

Feature	Claude	GPT (OpenAI)
Context window	100k tokens (Claude 2.1)	32k tokens (GPT-4 Turbo)
Tone	Friendly, safety-aligned	Neutral, data-driven
Error Sensitivity	May decline unclear prompts	Tends to respond even if unsure
Formatting Sensitivity	Higher	More flexible
Guardrails Required	Yes (for commercial use)	Optional (user-enforced)

Claude shines in tasks needing nuance, tone, and strong safety alignment.

📋 5.8 Claude Prompt Debugging Checklist

- ✅ Prompt is **instructional**
- ✅ Context is **clearly labeled**
- ✅ Input chunks are **clean, factual**
- ✅ Token limit is **respected** (e.g., trim large docs)
- ✅ No overlapping/conflicting instructions
- ✅ Use **fallback text** like “I don’t know”

📌 5.9 Prompt Tuning Tips

- 💡 **Use simpler language** — Claude prefers clarity over cleverness
- 🪄 **Include examples** — show it how to answer
- 🔁 **Test edge cases** — ask confusing, vague, or misleading questions
- ✂️ **Trim irrelevant context** — more signal, less noise
- 📈 **Track prompt outputs** — use logging to monitor variations

---

📚 5.10 Sample Claude Prompt from Your Guide

Here's one used in your Streamlit Claude Q&A app:

python

CopyEdit

claude_prompt = f"""

You are Claude, an expert PDF assistant.

Use ONLY the information from the following document segments to answer the question.

Context:

{context}

Question:

{user_question}

Answer:

"""

This form was tested for accuracy, coherence, and tone. Use it as your base template.

🖥️ Module 6: Full Claude RAG App with Streamlit

This module walks you through **building a complete Retrieval-Augmented Generation (RAG) web app** using Claude, AWS Bedrock, LangChain, and Streamlit. You'll learn to connect document loading, embedding, retrieval, prompt chaining, and front-end interactions in a unified architecture.

🔧 6.1 App Architecture Overview

Components:

- 📄 **PDF Loader** → Extracts content from uploaded files
- 🔍 **Vector Store** → FAISS for local semantic search
- 🧠 **Claude Chain** → Handles prompt template + response via Bedrock
- 🌐 **Streamlit UI** → Clean, interactive web interface
- 🛠️ **Modular Python Scripts** → Separate logic for better reuse

High-Level Flow:

css

CopyEdit

[PDF Upload]

↓

[Text Extraction]

↓

[Chunking + Embedding]

↓

[FAISS Vector Index]

↓

[User Query → Claude Prompt]

↓

[Claude Response (via Bedrock)]

↓

[Streamlit Output]

🧩 6.2 Project Folder Structure

graphql

CopyEdit

```
claude_pdf_qa_bot/
|
├── app.py              # Streamlit UI
├── pdf_loader.py       # PDF loading and chunking
├── vectorstore.py      # FAISS vector store
├── rag_chain.py        # Claude prompt + QA logic
├── prompts.py          # Custom templates
├── utils.py            # Misc helpers
├── requirements.txt    # Dependencies
└── assets/             # Images, logos, static files
```

📦 6.3 Requirements File (requirements.txt)

txt

CopyEdit

streamlit

boto3

langchain

pypdf

faiss-cpu

tiktoken

python-dotenv

Add openai, unstructured, or langchain-community modules as needed.

📜 6.4 app.py – Streamlit App UI

python

CopyEdit

```
import streamlit as st
from pdf_loader import load_pdf_chunks
from vectorstore import get_vectorstore
from rag_chain import get_claude_response

st.set_page_config(page_title="Claude PDF Q&A Bot")

st.title(" 📘 Claude PDF Q&A Bot")

uploaded_file = st.file_uploader("Upload a PDF", type=["pdf"])
query = st.text_input("Ask a question")

if uploaded_file and query:
```

```
    chunks = load_pdf_chunks(uploaded_file)
    vs = get_vectorstore(chunks)
    answer = get_claude_response(query, vs)
    st.write("🧠 Claude's Answer:")
    st.success(answer)
```

📄 6.5 pdf_loader.py – Load and Chunk PDFs

```
python
CopyEdit
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter

def load_pdf_chunks(file):
    pdf = PdfReader(file)
    text = "".join([page.extract_text() for page in pdf.pages])
    splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
    return splitter.split_text(text)
```

🔍 6.6 vectorstore.py – FAISS Embeddings

```
python
CopyEdit
from langchain.vectorstores import FAISS
from langchain.embeddings import BedrockEmbeddings

def get_vectorstore(chunks):
    embeddings = BedrockEmbeddings()
    return FAISS.from_texts(chunks, embedding=embeddings)
```

You may also use OpenAIEmbeddings() for local testing.

🔁 6.7 rag_chain.py – Claude Chain

python

CopyEdit

```
from langchain.llms import Bedrock
from langchain.chains import RetrievalQA
from prompts import claude_prompt_template

def get_claude_response(query, vectorstore):
    llm = Bedrock(model_id="anthropic.claude-v2")
    chain = RetrievalQA.from_chain_type(
        llm=llm,
        retriever=vectorstore.as_retriever(),
        chain_type="stuff",
        chain_type_kwargs={"prompt": claude_prompt_template}
    )
    return chain.run(query)
```

✍️ 6.8 prompts.py – Claude Prompt Template

python

CopyEdit

```
from langchain.prompts import PromptTemplate

claude_prompt_template = PromptTemplate(
    input_variables=["context", "question"],
    template="""
You are Claude, a helpful assistant.
```

Use ONLY the information below to answer.

Context:

{context}

Question:

{question}

Answer:

"""

)

🌐 6.9 Streamlit UX Enhancements

Add these Streamlit improvements:

- st.spinner("Thinking...") → better UX
- st.markdown() → use for styled sections
- st.sidebar → add about info, contact
- Add session state for multi-turn Q&A

Example:

python

CopyEdit

```
with st.spinner("Claude is thinking..."):
    answer = get_claude_response(query, vs)
```

🍼 6.10 Deploy Locally

bash

CopyEdit

```
pip install -r requirements.txt
```

streamlit run app.py

Make sure .env includes your AWS credentials and Bedrock region settings:

ini

CopyEdit

AWS_ACCESS_KEY_ID=your_key

AWS_SECRET_ACCESS_KEY=your_secret

AWS_DEFAULT_REGION=us-east-1

🚀 6.11 Optional: Deploy to Hugging Face or Streamlit Cloud

To host the bot:

- Add secrets.toml (for Streamlit Cloud)
- Make repo public on GitHub
- Push your code and PDF assets
- Create an app on Streamlit Cloud

🛡️ Module 7: Claude Error Handling & Retry Wrappers

When deploying Claude-powered apps using AWS Bedrock, handling transient errors, latency, or throttling gracefully is essential for a production-grade experience. This module teaches how to implement **retry wrappers**, **exception handling**, and **graceful fallbacks** around the Claude LLM calls using LangChain.

⚠️ 7.1 Common Errors with Claude + Bedrock

Error Type	Cause	Resolution
ThrottlingException	Too many requests/sec to Bedrock API	Use exponential backoff & retries
InternalServerError	AWS infrastructure failure	Retry with delay
TimeoutError or ReadTimeout	Claude LLM response takes too long	Increase timeout, use retry logic
ValidationException	Malformed prompt, missing input parameters	Add parameter validation in prompt
ModelNotReadyException	Model loading delay	Retry with wait

🔁 7.2 Retry Wrapper with tenacity

Install the package:

bash

CopyEdit

pip install tenacity

Example Wrapper

python

CopyEdit

from tenacity import retry, wait_exponential, stop_after_attempt, retry_if_exception_type

```
import botocore.exceptions

@retry(
    wait=wait_exponential(multiplier=1, min=2, max=30),
    stop=stop_after_attempt(5),
    retry=retry_if_exception_type((botocore.exceptions.ClientError, TimeoutError))
)
def call_claude(prompt, llm):
    return llm(prompt)
```

This code retries failed Claude calls up to 5 times with exponential backoff.

🧰 7.3 Claude LLM Retry with LangChain Wrapper

```
python
CopyEdit
from langchain.llms import Bedrock
from tenacity import retry, stop_after_attempt, wait_random_exponential

@retry(wait=wait_random_exponential(min=1, max=10), stop=stop_after_attempt(5))
def run_claude_chain(chain, query):
    return chain.run(query)
```

Use run_claude_chain instead of chain.run() directly.

✅ 7.4 Fallback Logic

In mission-critical apps, consider **fallbacks**:

```
python
CopyEdit
def fallback_response():
    return "Sorry, I couldn't process your request at the moment. Please try again later."
```

```
try:
    answer = run_claude_chain(chain, query)
except Exception as e:
    print("Claude error:", e)
    answer = fallback_response()
```

This prevents app crashes and improves UX during outages.

💉 7.5 Observability: Logging Claude Calls

Add structured logs to trace Claude interactions:

```
python
CopyEdit
import logging
logging.basicConfig(level=logging.INFO)

def run_with_logging(query):
    logging.info(f"User Query: {query}")
    response = run_claude_chain(chain, query)
    logging.info(f"Claude Response: {response}")
    return response
```

Optional: Push logs to S3, CloudWatch, or Datadog.

⚙️ 7.6 Graceful Timeout with concurrent.futures

To limit max response time per user request:

```
python
CopyEdit
import concurrent.futures
```

```
def safe_claude_run(query):
    with concurrent.futures.ThreadPoolExecutor(max_workers=1) as executor:
        future = executor.submit(run_claude_chain, query)
        try:
            return future.result(timeout=15)  # 15 seconds max
        except concurrent.futures.TimeoutError:
            return "Claude is taking too long. Try again later."
```

📝 7.7 Retry UX in Streamlit

Show retry progress to users:

python

CopyEdit

```
with st.spinner("Claude is thinking... (with retry logic)"):
    try:
        answer = run_claude_chain(chain, query)
        st.success(answer)
    except:
        st.error("Oops! Something went wrong. Try again later.")
```

📌 Summary

Feature	Implementation
Retry Logic	tenacity decorators
Exponential Backoff	wait_exponential()
Timeout Handling	concurrent.futures
Fallbacks	try-except + default msg
Logging	logging.info() + Claude I/O

🧠 Module 8: Advanced Prompt Engineering for Claude on AWS Bedrock

Claude's effectiveness as a Q&A bot in RAG pipelines depends heavily on how prompts are structured, contextualized, and grounded with retrieved data. In this module, we'll cover advanced prompt design techniques tailored for Claude's API on AWS Bedrock.

📌 8.1 Prompt Anatomy for Claude

Claude responds best to structured, instruction-style prompts. Here's a proven structure:

text

CopyEdit

Human: You are an expert assistant. Answer the question based on the document context below.

Document:

"""

<INSERT_RETRIEVED_TEXT_HERE>

"""

Question: <INSERT_USER_QUESTION_HERE>

Assistant:

Notice the key parts:

- **Instructional tone**: Clear task definition.
- **Context window**: Grounding the response using RAG.
- **Delimiters**: Use """ or --- to separate context.

✨ 8.2 Claude-Specific Prompt Best Practices

Guideline	Reason
Use "Human:" and "Assistant:"	Claude prefers Anthropic-style formatting
Include delimiters like """	Helps distinguish doc from instructions
Avoid overly long context (>8k tokens)	Claude's context window on Bedrock may truncate input
State the role explicitly	"You are a helpful assistant…" gives more reliable completions
Ask for step-by-step answers	Improves accuracy on reasoning questions

---

⚙️ 8.3 Prompt Templates in LangChain

python

CopyEdit

```
from langchain.prompts import PromptTemplate


prompt_template = PromptTemplate(
    input_variables=["context", "question"],
    template="""
Human: You are a helpful AI assistant. Use only the information provided to answer.
Context:
```

\"\"\"{context}\"\"\"

Question: {question}

Assistant:

"""

)

Use this template with LLMChain.

🔁 8.4 Few-shot Examples (Optional)

Claude can handle **few-shot learning**. Example:

text

CopyEdit

Human: You're an expert in medical policy compliance.

Context:

"""

In India, Form 10D is required for pension withdrawal.

"""

Question: What form is needed for pension withdrawal?

Assistant: Form 10D

Context:

"""

<INSERT_DYNAMIC_DOCUMENT_HERE>

"""

Question: <INSERT_QUESTION_HERE>

Assistant:

Use judiciously to avoid context overflow.

🧪 8.5 Prompt Engineering Sandbox

Use promptlayer or LangChain Playground to experiment:

bash

CopyEdit

pip install promptlayer

Use Claude as your LLM and run rapid iterations:

python

CopyEdit

from langchain.llms import Bedrock

llm = Bedrock(model_id="anthropic.claude-v2", region_name="us-east-1")

prompt = prompt_template.format(context=docs, question="What is the compliance deadline?")

llm(prompt)

Test, tweak, refine, repeat.

🔬 8.6 Improving Accuracy with Context Splitting

Use chunk overlap + refined retriever settings:

python

CopyEdit

from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(

  chunk_size=750,

  chunk_overlap=100

)

chunks = splitter.split_documents(docs)

Prevent cutoff of semantic boundaries.

🧠 8.7 Instruction Fine-tuning vs Prompting

Claude (via Bedrock) does **not support fine-tuning**, so all optimization comes from:

- Prompt wording
- Context structuring
- Chain logic

Use tools like **LangSmith** to analyze performance.

🔐 8.8 Claude as Safe Completion Agent

Claude includes built-in safety measures. Still, avoid ambiguous prompts:

✅ Do:

text

CopyEdit

Human: Based on the context, summarize the key environmental policy.

🚫 Avoid:

text

CopyEdit

Human: Write anything you want about this topic.

Too open-ended prompts may trigger refusals or generic completions.

💬 8.9 UI-Level Prompt Controls

In your Streamlit frontend, allow users to:

- Set “tone” (Formal / Friendly)
- Choose output format (Bullet / Paragraph)
- Enable summarization vs verbatim answer

Example:

python

CopyEdit

```
tone = st.selectbox("Select response tone", ["Formal", "Friendly"])
if tone == "Formal":
    instruction = "Use a formal tone while answering."
```

Then append instruction to base prompt dynamically.

✅ **Summary Table**

Technique	Purpose
Prompt Templates	Maintain consistency across queries
Few-Shot Prompting	Boosts specificity
Instruction Layering	Guides tone and structure
Claude-specific formatting	Ensures response reliability
Prompt UX in Streamlit	Customizes experience

🚀 Module 9: Deploying Claude PDF Q&A Bot to Hugging Face Spaces or Render

Once your Claude-powered Streamlit app is working locally, it's time to deploy it publicly so others can access it. In this module, you'll learn two reliable and free (or low-cost) deployment paths:

- Hugging Face Spaces (using Streamlit)
- Render (using Docker)

🧱 9.1 Deployment Prerequisites

Files Required:

✅ app.py (Streamlit app)

✅ requirements.txt (Python dependencies)

✅ .streamlit/config.toml (optional: for UI tweaks)

✅ claude_api_wrapper.py (Claude invocation logic)

✅ .env (use for secrets, excluded via .gitignore)

🛰️ 9.2 Deploying to Hugging Face Spaces

Why HF Spaces?

- 100% free for public repos
- Supports Streamlit, Gradio, or static apps
- Easy integration with GitHub

Step-by-Step:

1. Create a New Space

Go to: https://huggingface.co/spaces

- Click + Create new Space
- Choose Streamlit
- Name it e.g., claude-pdf-qna-bot
- Choose **public** (or private for HF Pro)

2. Push Your Code

You can use Git or web upload.

Structure:

arduino

CopyEdit

claude-pdf-qna-bot/

├── app.py

├── claude_api_wrapper.py

├── requirements.txt

└── .streamlit/

  └── config.toml

Sample requirements.txt:

txt

CopyEdit

streamlit

boto3

langchain

pypdf

python-dotenv

3. Configure Secrets (Claude API Key)

HF Spaces doesn't support .env files directly.

Instead, set environment variables using the **HF secrets interface**.

- Go to your Space → Settings → Secrets
- Add:
    - AWS_ACCESS_KEY_ID
    - AWS_SECRET_ACCESS_KEY
    - BEDROCK_REGION

4. Run It!

Once code and secrets are in place, the app auto-deploys.

🎉 Done! Your public URL will be:

arduino

CopyEdit

https://<your-username>.huggingface.co/spaces/claude-pdf-qna-bot

---

🚢 9.3 Deploying to Render with Docker

Why Render?

- Great for private deployments
- Supports background workers + autoscaling
- Can handle secrets via dashboard

Step-by-Step:

1. Add Dockerfile to Your Project

dockerfile

CopyEdit

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["streamlit", "run", "app.py", "--server.port=8080", "--server.enableCORS=false"]
```

2. Push Code to GitHub

Create a GitHub repo with your full app structure and Dockerfile.

3. Connect Render

- Go to https://render.com
- Click **New → Web Service**
- Choose **Deploy from GitHub**
- Select your repo

Settings:

- Runtime → Docker
- Name → claude-qna
- Region → closest to users
- Port → 8080

4. Set Secrets

Go to Environment → Add Environment Variables

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- BEDROCK_REGION

🎯 You can also add:

- CLAUDE_MODEL_ID=anthropic.claude-v2

🔧 9.4 Tips for Scaling

Provider	Public Access	Cost	Notes
Hugging Face	✅ Yes	Free (Public)	No custom domain support
Render	✅ Yes	Free/paid	Private repos, custom domains
AWS EC2	✅ Yes	Pay-per-use	Full control, but higher setup

🍼 9.5 Testing the Deployment

Once deployed:

- Visit the app URL
- Upload a sample PDF

- Ask a few questions
- Check console logs (HF → Logs, Render → Dashboard)

If errors appear:

- Ensure Claude API keys are correct
- Validate AWS Bedrock region and permissions

💡 9.6 Bonus: Adding a Custom Domain (Render)

Want your own branded domain like:

arduino

CopyEdit

https://qna.yourstartup.ai

Render supports custom domains.

1. Go to Render → your app → Settings
2. Scroll to "Custom Domains"
3. Add your domain
4. Update DNS settings from your domain provider

🔁 Module 10: GitHub Actions for Continuous Deployment + Claude Testing Pipelines

Once your Claude PDF Q&A Bot is deployed, automating your updates is the next step toward a production-grade workflow. GitHub Actions allows you to set up CI/CD pipelines, Claude API testing routines, and automated deployments to Hugging Face or Render.

🧩 10.1 What Is GitHub Actions?

GitHub Actions is a workflow automation tool built into GitHub. It lets you:

- Automate testing of Claude prompt output.
- Deploy automatically when you push changes.
- Notify yourself of build failures or missing secrets.
- Test AWS Bedrock API keys in CI.

🧱 10.2 Basic Folder Structure for CI

markdown

CopyEdit

.github/

└── workflows/

  └── deploy.yml

  └── test_claude.yml

Make sure these files are inside .github/workflows/.

🚀 10.3 Deploy to Hugging Face Spaces via GitHub Push

Hugging Face supports deployment via Git push.

Sample Workflow: deploy.yml

yaml

CopyEdit

```
name: Deploy to Hugging Face

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repo
        uses: actions/checkout@v3
```

```
    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'

    - name: Install Dependencies
      run: |
        pip install -r requirements.txt

    - name: Deploy to HF (Git Push)
      run: |
        git config --global user.email "you@example.com"
        git config --global user.name "Your Name"
        git remote set-url origin https://<HF_TOKEN>@huggingface.co/spaces/yourname/claude-pdf-bot
        git push origin main
      env:
        HF_TOKEN: ${{ secrets.HF_TOKEN }}
```

🔐 Set your Hugging Face token in GitHub → Settings → Secrets → HF_TOKEN.

✅ 10.4 Claude Prompt Testing Workflow

You can validate Claude responses automatically using a test script.

Create: tests/test_claude_output.py

```
python
CopyEdit
from claude_api_wrapper import ask_claude

def test_answer_not_empty():
```

```
  prompt = "Summarize this PDF section in one line."
  answer = ask_claude(prompt)
  assert answer is not None and len(answer.strip()) > 10
```

Sample Workflow: test_claude.yml

yaml

CopyEdit

```
name: Claude API Prompt Test

on:
 push:
  branches: [main]
 pull_request:

jobs:
 test-claude:
  runs-on: ubuntu-latest

  steps:
   - name: Checkout Code
     uses: actions/checkout@v3

   - name: Set up Python
     uses: actions/setup-python@v4
     with:
       python-version: '3.10'

   - name: Install dependencies
```

```
    run: |
      pip install -r requirements.txt
      pip install pytest

  - name: Run Claude Tests
    run: pytest tests/

    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      BEDROCK_REGION: ${{ secrets.BEDROCK_REGION }}
```

🔐 Don't forget to set the AWS secrets under GitHub Secrets.

🛠️ 10.5 Real-World Use Cases

Task	GitHub Actions Workflow
Claude response validation	test_claude.yml
Auto-deploy to HF or Render	deploy.yml
Code formatting + lint checks	black, flake8 workflows
Auto Docker build	Push to DockerHub or GHCR
Notifications via Slack	Webhook step on failure

📈 10.6 Optional Enhancements

- **Use pre-commit** hooks to enforce formatting before commit
- **Slack notifications** using GitHub Actions + Slack API
- **Email reports** for failed Claude generations
- **Streamlit cache invalidation** on deploy

🔐 10.7 GitHub Secrets Required

Set these under:

GitHub → Your Repo → Settings → Secrets and Variables → Actions

Name	Value (example)
AWS_ACCESS_KEY_ID	Your Bedrock AWS Key
AWS_SECRET_ACCESS_KEY	Your AWS secret
BEDROCK_REGION	e.g. us-east-1
HF_TOKEN	Hugging Face token (for auto-push)

🟦 Claude PDF Q&A Bot — Technical Deep Dive Gudde

Build Your Own RAG-Powered Chatbot with Claude on AWS Bedrock, LangChain, and Streamilt