awesome—here’s a tight 4-day plan that maps exactly to what you listed, with hands-on labs, mini-codes, and concrete deliverables each day.

**Day 1 — ML Lifecycle + VS Code + GitHub + Python + .env + Kaggle + (Unstructured ➜ Structured)**

**Learning goals**

* Understand the ML lifecycle end-to-end.
* Set up a clean dev workflow (VS Code, venv, Git, GitHub).
* Pull a Kaggle dataset (COVID-19) and turn messy data into a tidy table.

**Topics**

* ML lifecycle: problem framing → data → features → model → eval → deploy → monitor → iterate
* VS Code essentials, Python venvs, project layout, .env secrets
* Git basics: repo init, branching, PRs
* Kaggle API quickstart
* Unstructured → structured (parsing, typing, validation)

**Repo scaffold (copy/paste)**

ml-bootcamp/

├─ data/ # raw & processed

├─ notebooks/ # exploration (optional)

├─ src/

│ ├─ data\_ingest.py

│ ├─ clean.py

│ └─ utils.py

├─ .env.example

├─ requirements.txt

└─ README.md

**Setup commands**

# VS Code terminal

python -m venv .venv

# Windows PowerShell:

.venv\Scripts\Activate.ps1

pip install --upgrade pip

pip install python-dotenv pandas matplotlib scikit-learn kaggle pyarrow

git init

git branch -M main

git remote add origin https://github.com/<you>/<repo>.git

**.env.example**

KAGGLE\_USERNAME=your\_kaggle\_user

KAGGLE\_KEY=your\_kaggle\_api\_key

**Lab 1: Download + clean COVID-19 (example “covid\_19\_clean\_complete.csv”)**

# src/data\_ingest.py

import os, zipfile, io, pandas as pd

from dotenv import load\_dotenv

load\_dotenv()

# If you already have the CSV locally, just read it:

df = pd.read\_csv("data/covid\_19\_clean\_complete.csv")

# Minimal normalize

keep = ["Date","Country/Region","Confirmed","Recovered","Deaths"]

df = df[keep].rename(columns={"Country/Region":"country","Date":"date"})

df["date"] = pd.to\_datetime(df["date"])

df = df.sort\_values(["country","date"]).reset\_index(drop=True)

df.to\_parquet("data/covid\_daily.parquet", index=False)

print(df.head())

**Deliverable (Day-1)**

* Repo on GitHub with data/covid\_daily.parquet committed (or in Git LFS).
* Short README describing lifecycle + what you cleaned.

**Day 2 — Coordinates, Vectors, Precision/Recall, Confusion Matrix, Chunking (all types)**

**Learning goals**

* Build geometric intuition: coordinate systems & vector spaces.
* Compute precision, recall, F1, confusion matrix.
* Compare chunking strategies; why “different systems → different chunking.”

**Topics**

* 2D/nd coordinates, cosine similarity vs Euclidean distance
* Classification metrics: TP/FP/FN/TN, precision/recall/F1, ROC-AUC (brief)
* Chunking: fixed-size, overlapping, sentence/semantic, heading-aware, page-based, hierarchical

**Metrics mini-lab**

from sklearn.metrics import precision\_score, recall\_score, f1\_score, confusion\_matrix

y\_true = [1,0,1,1,0,0,1,0]

y\_pred = [1,0,0,1,0,1,1,0]

print("Precision:", precision\_score(y\_true, y\_pred))

print("Recall :", recall\_score(y\_true, y\_pred))

print("F1 :", f1\_score(y\_true, y\_pred))

print("ConfMat:\n", confusion\_matrix(y\_true, y\_pred))

**Chunking cheat-sheet**

* **Fixed tokens** (fast, uniform): size=1000, overlap=100
* **Sentence/semantic** (quality ↑): split by sentences + merge till ~T tokens
* **Heading-aware** (DOCX/PDF with structure): group under headings → sub-split
* **Page-based** (PDF): per page then merge short pages
* **Hierarchical**: headings → paragraphs → sentences (fallback)
* **Rule of thumb**: choose chunk size by **retrieval granularity** and **model context**; overlap helps preserve cross-chunk context. Different content = different optimal chunking.

**Deliverable (Day-2)**

* A small Python module src/chunking.py that implements:
  + fixed\_chunk(text, tokens=1000, overlap=100)
  + semantic\_chunk(text, target\_tokens=1000, min\_sentences=2)

**Day 3 — Local PDFs ➜ Chunks ➜ Embeddings ➜ Clustering ➜ Vector DBs + RAG (incl. Fusion RAG)**

**Learning goals**

* Build a local pipeline: parse PDF → chunk → embed → store → cluster → retrieve.
* Understand RAG, its flow, and variants (plain, router, re-rank, **fusion RAG**).

**Vector DB options**

* **Local**: Chroma, FAISS
* **Managed**: Pinecone, Qdrant Cloud
* **General DBs**: MongoDB Atlas + Vector Search

**RAG flow (standard)**

1. Ingest (parse → chunk) → 2) Embed → 3) Upsert to vector store
2. Query: (a) embed query (b) retrieve top-k (c) build context → 5) LLM answer

**RAG variants**

* **Basic**: single retriever
* **Re-ranker**: post-retrieval reranking (e.g., cross-encoder)
* **Router**: multiple indices by domain; route per query
* **Citation-first**: retrieve then cite with sources
* **Fusion RAG**: retrieve from **multiple sources** (local + Google Drive + AWS S3) and *merge-and-rank* results before answering

**Minimal ingestion & store (local Chroma example)**

# src/ingest\_pdf.py

import fitz, re

from pathlib import Path

from chromadb import Client

from chromadb.config import Settings

from openai import OpenAI # pip install openai>=1.0.0

import tiktoken, uuid

def extract\_pdf\_text(path:str)->str:

doc = fitz.open(path)

pages = [p.get\_text("text") for p in doc]

doc.close()

return "\n\n".join(pages)

def fixed\_chunk(text, max\_tokens=1000, overlap=100, enc="cl100k\_base"):

enc = tiktoken.get\_encoding(enc)

toks = enc.encode(text)

out, i = [], 0

while i < len(toks):

j = min(i+max\_tokens, len(toks))

out.append(enc.decode(toks[i:j]))

i = j - overlap

if i < 0: i = 0

return out

def embed\_texts(texts):

client = OpenAI()

# Choose your embedding model (example name; set in .env as OPENAI\_API\_KEY)

model = "text-embedding-3-large"

return [client.embeddings.create(model=model, input=t).data[0].embedding for t in texts]

def upsert\_chroma(chunks, collection="docs"):

db = Client(Settings(anonymized\_telemetry=False))

col = db.get\_or\_create\_collection(collection)

embs = embed\_texts(chunks)

ids = [str(uuid.uuid4()) for \_ in chunks]

col.add(ids=ids, documents=chunks, embeddings=embs)

return len(ids)

if \_\_name\_\_ == "\_\_main\_\_":

pdf\_path = r"C:\path\to\your.pdf"

text = extract\_pdf\_text(pdf\_path)

chunks = fixed\_chunk(text, max\_tokens=1000, overlap=120)

n = upsert\_chroma(chunks, collection="local\_pdf")

print("Upserted:", n)

**Clustering (automatic) to inspect corpus**

# src/cluster\_view.py

from sklearn.cluster import KMeans

import numpy as np

# embs = (N, D) numpy array of your embeddings

k = 8

labels = KMeans(n\_clusters=k, random\_state=0).fit\_predict(embs) # use elbow/silhouette to tune

**Fusion retrieval sketch**

# src/fusion\_retriever.py

# Pseudocode: retrieve top\_k from each source → score-normalize → merge → top\_k global

def fuse(results\_by\_source, k=8):

# results\_by\_source: dict{name: [(doc, score), ...]}

pool = []

for name, items in results\_by\_source.items():

for doc, score in items:

pool.append((doc, score, name))

# normalize scores per source then sort desc

# ... implement min-max per source ...

pool.sort(key=lambda x: x[1], reverse=True)

return pool[:k]

**Deliverable (Day-3)**

* Scripts that: parse a **local PDF**, chunk, embed, upsert into **one** vector DB (Chroma or Qdrant/Pinecone).
* A short note explaining **What is RAG**, **Flow**, and **Which RAG variant** you’ll use tomorrow (recommend: Fusion RAG if you truly need multi-source).

**Day 4 — Chatbot (RAG UI in Streamlit) + Fusion RAG (Local + Google Drive + AWS)**

**Learning goals**

* Ship a minimal, production-style RAG chatbot in Streamlit.
* Wire multi-source retrieval (local + Drive + S3) into a single fused context.

**.env (extend)**

OPENAI\_API\_KEY=...

# If using Google Drive + S3 in your own utilities:

GOOGLE\_DRIVE\_FILE\_IDS=1abc...,7xyz...

AWS\_ACCESS\_KEY\_ID=...

AWS\_SECRET\_ACCESS\_KEY=...

S3\_BUCKET=...

**Minimal Streamlit RAG app (single file)**

# rag\_chat.py

import streamlit as st

from openai import OpenAI

from chromadb import Client

from chromadb.config import Settings

st.set\_page\_config(page\_title="RAG Chat", layout="wide")

st.title("RAG Chatbot")

@st.cache\_resource

def get\_db():

return Client(Settings(anonymized\_telemetry=False)).get\_or\_create\_collection("local\_pdf")

def retrieve(query, k=6):

db = get\_db()

results = db.query(query\_texts=[query], n\_results=k, include=["documents","distances"])

docs = results["documents"][0]

dists= results["distances"][0]

# convert distance to similarity-ish score (1/(1+dist))

pairs = list(zip(docs, [1/(1+d) for d in dists]))

return pairs

def fuse\_multi\_source(query, k=8):

# TODO: add Google Drive / S3 retrievers; here we demo local only

local = retrieve(query, k=8)

# drive = retrieve\_from\_drive(...)

# s3 = retrieve\_from\_s3(...)

merged = sorted(local, key=lambda x: x[1], reverse=True)[:k]

return [d for d,\_ in merged]

def answer\_with\_context(query, context\_docs):

system = "Answer with citations. If unsure, say so."

context = "\n\n---\n".join(context\_docs[:6])

prompt = f"{system}\n\nContext:\n{context}\n\nUser: {query}\nAssistant:"

client = OpenAI()

resp = client.chat.completions.create(

model="gpt-4o-mini",

messages=[{"role":"user","content":prompt}],

temperature=0.2

)

return resp.choices[0].message.content

with st.form("chat"):

q = st.text\_input("Ask a question about your documents")

submitted = st.form\_submit\_button("Ask")

if submitted and q.strip():

with st.spinner("Retrieving..."):

ctx = fuse\_multi\_source(q)

with st.spinner("Generating..."):

ans = answer\_with\_context(q, ctx)

st.markdown("### Answer")

st.write(ans)

with st.expander("Context chunks"):

for i, d in enumerate(ctx, 1):

st.markdown(f"\*\*{i}.\*\* {d[:500]}…")

**Where to plug Fusion sources**

* **Google Drive**: download file(s) by ID (Drive API), parse → index in a separate collection, add a retrieve\_from\_drive(query) that queries that collection.
* **AWS S3**: list objects (e.g., PDFs/txt), fetch/parse/index into its own collection, retrieve\_from\_s3(query).
* In fuse\_multi\_source, call all retrievers, normalize scores per source, then merge + top-k.

**Quality tips**

* Add a **re-rank** step with a cross-encoder (optional) to refine the final top-k.
* Store metadata (source, page, heading) so your UI can show citations.
* Cache frequent Q&A (Redis) between retrieval and LLM to save tokens.

**Deliverable (Day-4)**

* rag\_chat.py running locally with your indexed local PDF.
* (Stretch) Drive + S3 sources integrated and fused.

**Quick checklist**

* **Day-1**
  + venv + requirements installed
  + GitHub repo pushed
  + Kaggle dataset cleaned to parquet
* **Day-2**
  + Metrics script (precision/recall/F1 + confusion matrix)
  + chunking.py with fixed + semantic
* **Day-3**
  + PDF ➜ chunks ➜ embeddings ➜ vector DB upsert
  + Note on RAG types + chosen variant
* **Day-4**
  + Streamlit chatbot answering with citations
  + Fusion hooks ready (Drive/S3 collections + merger)

if you want, I can drop in a **Drive retriever** and **Redis cache** snippet next, plugged into the same structure.

Got it! I’ve pulled out clean, ready-to-paste documentation for the three advanced RAG variants you highlighted. If you want a PPT afterward, I can turn this into slides with a simple flow diagram per type.

**Corrective RAG**

**Goal**  
Improve factual alignment to the *retrieved context* by adding a self-checking loop that verifies and corrects the draft answer.

**When to use**

* Compliance/accuracy is critical (policy, medical guidelines, contracts).
* You can afford extra latency for higher precision.

**Architecture (flow)**

1. Retrieve top-k chunks for the query.
2. Generate a draft answer (LLM).
3. **Judge**: A verifier agent checks the draft strictly against retrieved chunks.
4. If misaligned/low-confidence → propose corrections or ask for re-retrieve → regenerate.
5. Return corrected answer + citations.

**Key components**

* Retriever (BM25 / dense / hybrid).
* Generator LLM.
* Verifier LLM (can be same model with different prompt).
* Optional: rule checks (regex, policy constraints).

**Evaluation**

* Context-faithfulness (does the answer quote/align with provided chunks?).
* Precision@k (retrieval), Answer EM/F1 on grounded QA sets, Human spot checks.

**Common pitfalls**

* Infinite or redundant loops → cap iterations.
* Over-strict verifier rejecting good paraphrases → calibrate thresholds.

**Pseudocode sketch**

ctx = retrieve(query, k=8)

draft = llm.generate(query, context=ctx)

verdict = llm.verify(draft, context=ctx, policy="must be grounded")

if verdict.needs\_fix:

ctx2 = maybe\_retrieve\_more(query, hints=verdict.gaps)

final = llm.generate(query, context=ctx+ctx2, guidance=verdict.corrections)

else:

final = draft

return final, cite(ctx)

**Speculative RAG**

**Goal**  
Reduce latency by pre-drafting likely responses/intent before full retrieval, then confirming/refining once evidence arrives.

**When to use**

* Interactive apps needing low perceived latency (chat, assistive UIs).
* Predictable domains (FAQ, product support) where intent priors help.

**Architecture (flow)**

1. **Speculate**: Light agent predicts likely intent/answer skeleton instantly.
2. In parallel, start retrieval on the actual query (and on speculative intents).
3. Merge: compare speculative draft with retrieved evidence; keep, refine, or correct.
4. Return answer; cache popular Q/A.

**Key components**

* Fast speculative head (small LLM/classifier/templates).
* Parallel retrievers (pre-fetch on predicted intents).
* Reconciliation step to ensure grounding.

**Evaluation**

* Latency (p50/p95), correction rate (how often speculation changes), final quality.

**Common pitfalls**

* Speculation drift (confident but wrong) → always reconcile with retrieved context.
* Wasted cycles on bad predictions → track and prune poor priors.

**Pseudocode sketch**

spec = small\_llm.predict\_intent\_and\_draft(query) # fast

ret\_main = retrieve\_async(query, k=6)

ret\_spec = retrieve\_async(spec.intent, k=4)

ctx = await merge(ret\_main, ret\_spec)

answer = big\_llm.generate(query, context=ctx, draft=spec.draft)

return answer, cite(ctx)

**Fusion RAG**

**Goal**  
Aggregate evidence from **multiple repositories/silos** (e.g., OneDrive, Google Drive, Azure Blob, DBs) and rank them together.

**When to use**

* Knowledge is spread across tools/tenants.
* You want broad recall without losing precision.

**Architecture (flow)**

1. **Per-source retrieval**: run the query against each source/index.
2. **Score normalization**: normalize distances/similarities per source.
3. **Reciprocal Rank Fusion (RRF)** or similar: combine lists into a global top-k.
4. Generate answer grounded in fused context; show per-source citations.

**RRF (typical)**

* For document with rank *r* in a list: score = 1/(k + r) (k≈60 by convention).
* Sum across sources → sort desc → choose top-k.

**Key components**

* Connectors (Drive/Blob/DB).
* Per-source retrievers + embeddings (consistent models or per-domain).
* Rank fusion (RRF, Borda, weighted variants).

**Evaluation**

* Recall@k across all sources, blended MRR/NDCG, answer quality with mixed-source citations.

**Common pitfalls**

* Score scales differ across stores → always normalize.
* Duplicates across sources → dedupe by content hash/URL.
* Slowest source bottlenecks latency → use timeouts and partial fusion.

**Pseudocode sketch**

results = {

"onedrive": retrieve(q, src="onedrive"),

"gdrive": retrieve(q, src="gdrive"),

"blob": retrieve(q, src="azure\_blob"),

}

fused = rrf\_fuse(results, k\_global=8)

ctx = [d.doc for d in fused]

answer = llm.generate(q, context=ctx, style="cite\_sources")

return answer, cite(fused)

**One-glance visual flows (ASCII)**

**Corrective RAG**

Query → Retrieve → Draft Answer → Judge vs Context → {Fix? → Re-retrieve/Regenerate} → Final + Citations

**Speculative RAG**

┌─> Speculative Draft (fast)

Query ────┤

└─> Full Retrieval ─┐

└─> Reconcile → Final + Citations

**Fusion RAG**

→ Source A Retrieve ─┐

Query → fan-out → Source B Retrieve ─┼→ Rank Fusion (RRF) → Context → Answer

→ Source C Retrieve ─┘

**Practical testing checklist**

* **Datasets**: sports, food, politics → then multi-source same-domain.
* **Metrics**: retrieval Recall@k, groundedness score, latency p95, human spot checks.
* **Pipelines**: confirm top results include items from multiple repos in fusion.
* **Prompts**: enforce grounding (“only use provided context; cite sources”).
* **GitHub**: commit working pipelines separately per variant; tag runs/results.

If you want this packaged as slides, say the word and I’ll generate a tidy PPT with a comparison table and the three flow diagrams baked in.