**Legal-AI Backend — Replit Quickstart & Build Plan (MVP)**

*Production-minded FastAPI backend for Company-Secretary researchers.*  
Focus today: **robust data gathering → high-quality, grounded summaries → reliable API**. Use Replit as the dev environment, but architecture is production-ready and portable.

**What you’ll get from this README**

1. Clear objectives and Monday-MVP checklist.
2. Ready project file structure to paste into Replit.
3. Exact env variables and commands to run locally / on Replit.
4. Concrete details for ingestion, parsing, summarisation (Gemini + LangChain agents).
5. API endpoints you’ll implement and quick test commands.
6. Quality gates, validation checks, and deployment hints.

**1 — MVP Goals (Today / Monday)**

* Ingest a small, curated corpus (e.g., last 5 years SC + NCLT judgments for company law).
* Produce **schema-enforced**, **grounded** summaries (CS-focused) using **LangChain + Gemini** agent.
* Expose APIs: /search, /documents/{id}, /documents/{id}/summary.
* Ensure every summary passes automated QA (grounding coverage threshold); failing ones are flagged for review.
* Demonstrate with a few sample documents and downloadable summary PDF.

Target users: Company Secretary researchers who need precise, citation-backed documents.

**2 — Project file structure (paste into your Replit workspace)**

legal-ai-backend/

├─ app/

│ ├─ main.py # FastAPI entrypoint

│ ├─ api/

│ │ └─ v1/

│ │ ├─ endpoints/

│ │ │ ├─ search.py

│ │ │ ├─ documents.py

│ │ │ └─ summaries.py

│ │ └─ deps.py

│ ├─ core/

│ │ ├─ config.py # env config (pydantic)

│ │ └─ logging.py

│ ├─ db/

│ │ ├─ base.py

│ │ ├─ models.py

│ │ ├─ session.py

│ │ └─ migrations/ # alembic (optional)

│ ├─ services/

│ │ ├─ storage.py # S3 / Supabase helper

│ │ ├─ search.py # Typesense wrapper

│ │ ├─ parser.py # PDF/HTML -> clean text

│ │ └─ summariser\_agent.py # LangChain + Gemini wrapper

│ ├─ tasks/

│ │ ├─ celery\_app.py

│ │ ├─ ingestion.py

│ │ └─ summarise.py

│ └─ utils/

│ ├─ text\_cleaner.py

│ └─ citation\_normalizer.py

├─ infra/

│ └─ docker-compose.yml

├─ tests/

│ └─ test\_sample\_ingest.py

├─ .env.example

├─ requirements.txt

└─ README.md

**3 — .env (example)**

Create .env or use Replit Secrets:

# DB

DATABASE\_URL=cockroachdb://<user>:<pass>@<host>:26257/<db>?sslmode=disable

# Redis / Celery

REDIS\_URL=redis://localhost:6379/0

# Search

TYPESENSE\_HOST=localhost

TYPESENSE\_PORT=8108

TYPESENSE\_API\_KEY=typesense-key

# Storage

S3\_ENDPOINT\_URL= # e.g., https://s3.amazonaws.com or Supabase URL

S3\_BUCKET=legal-ai-docs

S3\_ACCESS\_KEY=

S3\_SECRET\_KEY=

# Gemini / LangChain

GOOGLE\_API\_KEY=

LLM\_MODEL=gemini-1.5-pro # placeholder model name

# App

FASTAPI\_HOST=0.0.0.0

FASTAPI\_PORT=8000

**Replit tips:** Use the built-in Secrets UI to set sensitive env vars instead of a .env file.

**4 — Quick Start (Replit / Local)**

**Install**

requirements.txt must include (starter):

fastapi uvicorn[standard] sqlalchemy psycopg2-binary alembic

celery redis aiohttp pdfminer.six pypdf beautifulsoup4 langchain

typesense-client python-dotenv requests

In Replit shell:

pip install -r requirements.txt

**Run FastAPI (dev)**

uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload

**Run worker**

* On Replit you can either:
  + Run a second Repl for worker, or
  + Start worker in a background process (for quick dev):  
    celery -A app.tasks.celery\_app worker --loglevel=info
* For MVP on Replit, you can also run summariser synchronously via API (fallback) — **only for dev**.

**5 — Core models (SQLAlchemy simplified)**

Put in app/db/models.py (already provided earlier, but minimal here):

from sqlalchemy import Column, String, Date, Text, DateTime, ForeignKey, func

from sqlalchemy.dialects.postgresql import UUID, JSONB

from app.db.base import Base

import uuid

class Document(Base):

\_\_tablename\_\_ = "documents"

document\_id = Column(UUID(as\_uuid=True), primary\_key=True, default=uuid.uuid4)

title = Column(String, nullable=False)

court = Column(String)

decision\_date = Column(Date)

url = Column(String)

storage\_path = Column(String)

content\_hash = Column(String, unique=True, index=True)

raw\_text = Column(Text)

class Summary(Base):

\_\_tablename\_\_ = "summaries"

summary\_id = Column(UUID(as\_uuid=True), primary\_key=True, default=uuid.uuid4)

document\_id = Column(UUID(as\_uuid=True), ForeignKey("documents.document\_id"), nullable=False)

style = Column(String) # cs\_student | research | advocate

model\_id = Column(String)

prompt\_version = Column(String)

summary\_short = Column(Text)

summary\_detailed = Column(Text)

span\_citations = Column(JSONB)

quality\_score = Column(String)

human\_status = Column(String, default="pending")

created\_at = Column(DateTime(timezone=True), server\_default=func.now())

**6 — Ingestion pipeline (practical & idempotent)**

**Flow:** Prefect/Cron → Crawler connector(s) → fetch\_and\_store Celery task → parser → store raw PDF & text → index search → enqueue summariser.

Key rules:

* Respect robots.txt & ToS. If a source disallows scraping, only store metadata + link.
* **Dedupe by content\_hash (SHA256)** — compute on normalized text and skip if exists.
* Store original PDF snapshot in object storage and record storage\_path & checksum.
* Keep parser version in DB for provenance.

**Sample Celery task (ingestion.py)**

@celery.task

def fetch\_and\_store(url, source\_id):

raw = http\_get(url)

pdf\_path = storage.store\_raw\_pdf(raw, filename=...)

text = parser.extract\_text\_from\_pdf(raw)

norm = text\_cleaner.normalize(text)

content\_hash = sha256(norm)

if db.exists\_document(content\_hash): return

doc = db.create\_document(title=..., url=url, storage\_path=pdf\_path, raw\_text=norm, content\_hash=content\_hash)

typesense.index\_document(doc)

summarise\_document.delay(str(doc.document\_id))

**7 — Summariser agent (LangChain + Gemini) — structure & sample**

**Design**: Use a Structured Output schema; deterministic LLM params (temp=0). Each summary must produce JSON with issues, holding, reasoning, disposition, citations and span\_offsets.

**High-level pseudo code (summariser\_agent.py)**

from langchain import LLMChain

from langchain.output\_parsers import StructuredOutputParser, ResponseSchema

from langchain.chat\_models import ChatGoogleGenerativeAI # placeholder

schemas = [

ResponseSchema(name="issues", description="List of issues"),

ResponseSchema(name="holding", description="Court's holding"),

ResponseSchema(name="reasoning", description="Short reasoning"),

ResponseSchema(name="citations", description="List citations"),

ResponseSchema(name="span\_offsets", description="offsets in raw text for claims"),

]

parser = StructuredOutputParser.from\_response\_schemas(schemas)

prompt = """

You are a legal summariser for Company Secretary users. Output ONLY JSON in the schema:

{schema}

Document:

{text}

""".format(schema=parser.get\_format\_instructions(), text="{text}")

model = ChatGoogleGenerativeAI(model="gemini-1.5-pro", temperature=0)

chain = LLMChain(llm=model, prompt=prompt)

def summarise\_text(raw\_text):

response = chain.run({"text": raw\_text})

parsed = parser.parse(response)

return parsed

**Important:** After generation, always run a **grounding check** that every claim is backed by a span\_offset pointing into raw\_text (see QA section below).

**8 — Grounding & QA (non-negotiable)**

* **Grounding Coverage**: Target ≥ **95%** of summary claims must be linked to at least one offset/span in the raw\_text.
* **Citation Resolve Rate**: ≥ **90%** of cited cases/sections must be resolvable to our documents or to a known act section.
* **NLI Consistency Check**: Run a cheap NLI (or heuristic) to detect contradictions. Threshold ≤ **1% contradictions**.
* **If any check fails** → mark human\_status='pending' and push to reviewer queue.

These checks are applied automatically by summarise\_document task before writing Summary with approved status.

**9 — API endpoints (minimal for MVP)**

**1. Search**

GET /api/v1/search?q=companies+act+sec+241&page=1&per\_page=20

Response: paginated list (document\_id, title, court, date, snippet)

**2. Get document**

GET /api/v1/documents/{document\_id}

Response: metadata + signed URL to stored PDF + available summaries

**3. Get summary**

GET /api/v1/documents/{document\_id}/summary?style=cs\_student

Response: summary JSON (only if approved) + quality metrics + provenance (source url, checksum, spans)

**4. Manual ingestion trigger (auth required)**

POST /api/v1/ingest/manual

Body: { "url": "...", "source": "supreme\_court" }

Returns: task\_id

**cURL example**

curl "http://localhost:8000/api/v1/documents/<id>/summary?style=cs\_student"

**10 — Replit-specific notes & caveats**

* Replit is great for prototyping; however:
  + Long-running background workers are tricky on a single Repl. For workers, run a separate Repl instance or use a hosted Redis/Celery and a separate worker Repl.
  + Persistent disk space on free Replit is limited. Use Supabase Storage or S3 for PDFs.
  + Use Replit Secrets for credentials; don’t commit .env.
* For demo on Replit: you can run summariser synchronously within the HTTP request for small docs (only for demo). But **do not** do this in production.

**11 — Minimal checklist (concrete tasks — order matters)**

1. Create Repl and push the project structure above.
2. Set Replit Secrets: DB, Redis (or hosted), Typesense key, Google API key, S3 creds.
3. Implement DB models + run migrations (Alembic or simple create\_all for MVP).
4. Implement fetch\_and\_store Celery task and a simple driver script to enqueue 3 sample SC/NCLT URLs.
5. Implement parser.extract\_text\_from\_pdf using pdfminer.six / pypdf. Test on 3 sample PDFs.
6. Implement summariser\_agent.summarise\_text (LangChain wrapper) and test on one long judgment. Ensure JSON parse successful.
7. Implement grounding QA and only write Summary if pass; otherwise mark pending.
8. Implement GET /documents/{id} and GET /documents/{id}/summary endpoints.
9. Add Typesense indexing for the documents (title + snippet + facets).
10. Demo: ingest 3 docs → see saved summaries → fetch summary via API → download the PDF & summary.

**12 — Production notes & scaling (short)**

* **Database**: CockroachDB or managed Postgres with pgvector later.
* **Search**: Typesense or any free search application
* **Storage**: Neon db for now.
* **LLM**: Start with Gemini (reasonable cost / quality). Add caching of summary outputs and conservative rate-limits.