ECL Intelligence System

A Retrieval-Augmented Explainable Framework for Credit Risk and Segment-Level ECL Analysis

Vishwajeet Kumar

Department of Computer Science
Computer Science Specialization
Birla Institute of Technology

November 7, 2025

This project presents a complete end-to-end Expected Credit Loss (ECL) analysis system that integrates deterministic credit-risk modelling with Retrieval-Augmented Generation (RAG), role-based access control (RBAC), and a production-ready web interface. Using the Kaggle Loan_Data dataset, the platform computes PD, LGD, EAD, and ECL across multiple borrower segments and enables interactive, explainable insights through a ChatGPT-like assistant. The system emphasizes transparency, low latency, scalability, and secure segmentation, aligning with real-world financial risk assessment requirements.

Birla Institute of Technology
Department of Computer Science
India

Executive Summary

This project delivers a fully integrated Expected Credit Loss (ECL) analytics platform that enables financial decision-makers to compute, explore, and interpret loan portfolio risks with transparency and precision. Built with modern web technologies and cloud-native infrastructure, the platform combines a statistically grounded ECL engine with Retrieval-Augmented Generation (RAG) and Role-Based Access Control (RBAC), providing a secure and interactive environment for portfolio analysis. The system addresses three major challenges in contemporary credit risk assessment:

- **Explainability**:This system avoids black-box modelling and instead uses a clear, formula-based ECL computation aligned with regulatory expectations.
- **Accessibility**:Natural language querying through RAG allows users to obtain insights without writing SQL or understanding internal data structures.
- **Security**:The implemented RBAC ensures analysts only see permitted segments, while CROs maintain full oversight.

Using a structured loan dataset as input, the system automatically performs preprocessing, segmentation, and calculation of PD, LGD, EAD, and ECL per segment. These values are stored in a Neon PostgreSQL database and indexed with Pinecone for semantic retrieval. Users can then interact with the data through a React-based interface, enabling:

- Dataset upload and automated validation.
- RAG-driven natural language insights.
- Segment exploration with ECL metrics.
- Portfolio history review and trend analysis.
- Permission management for analysts (CRO-only).

The backend API, implemented in FastAPI, exposes 14 well-structured endpoints for authentication, ECL processing, querying, and data retrieval. JWT tokens protect all sensitive routes. The RAG engine ensures that all retrieved information is filtered through RBAC before reaching the LLM, preventing unauthorised exposure of risk insights. Performance testing demonstrates that the platform efficiently processes thousands of loan records, produces segment-level ECL values in under a second post-processing, and responds to queries in 1–3 seconds depending on the complexity and LLM interaction. Thanks to serverless storage (Neon) and scalable vector indexing (Pinecone), the system is designed to grow with production workloads. This platform enables CROs to oversee risk across the portfolio holistically and allows analysts to work independently within their permitted scope. The solution is therefore not merely a technical prototype but a deployable, production-ready system suitable for real-world credit risk operations.

Understanding the Problem Statement

We aim to deliver a conversation-driven Expected Credit Loss (ECL) analytics platform that computes transparent, segment-level ECL curves and recommends actions per segment—either *increase interest* or *reduce disbursement*. Users comprise **Analysts** (restricted, segment-scoped access) and a **CRO** (full access with permission management).

Core requirements.

- **ECL computation:** explainable formula ECL = PD × LGD × EAD at *segment* granularity.
- **Segmentation:** process the Kaggle bank-loan data; group by area, occupation, gender, purpose, home ownership, age, education.
- **RAG interface:** natural-language queries over vectorized segment documents; LLM answers grounded in calculated metrics.
- **RBAC:** enforce role-based access for all stages (API and retrieval); Analysts see only permitted segments; CRO manages permissions and reviews history.
- **History & deployment:** persist past ECL runs/reports; publicly deploy with secure, low-latency endpoints.

Design implications.

- **Explainability:** direct formula with auditable PD/LGD/EAD inputs and per-segment summaries.
- **Architecture:** FastAPI backend, React frontend, Neon PostgreSQL, Pinecone vectors, OpenAI for LLM.
- **Security & performance:** JWT auth, strict CORS/HTTPS, RBAC filters, efficient upload/query paths.
- **Outcomes:** actionable per-segment decisions and reproducible, time-stamped history for review.

Different Approaches

Method A: Deterministic ECL + RBAC + RAG

A rule-based pipeline computes segment ECL using ECL = PD × LGD × EAD. Data is segmented (area, occupation, gender, purpose, ownership, age), stored in Neon, embedded in Pinecone, and queried using RAG with RBAC filtering. This guarantees transparency, low latency, and full auditability.

Method B: Agentic Orchestration (LangGraph)

The workflow is split into modular nodes (Preprocess, Segment, Metrics, ECL, Embed, RAG) with retry/state handling. RBAC applies before context creation. This maintains deterministic ECL while improving reliability, modularity, and debuggability.

Method C: Hybrid PD (Calibrated ML) + Deterministic LGD/EAD

A lightweight calibrated ML model predicts PD from Kaggle features, while LGD/EAD remain rule-based. ECL is produced using calibrated PD and deterministic components, increasing sensitivity to borrower behaviour without sacrificing explainability or speed.

Compressed Pseudocode

```
def pd(s):  return s.defaults.sum()/len(s)
def lgd(s): return 0.35+0.05*(s.loan_int_rate.mean()/100)
def ead(s): return s.loan_amnt.mean()
def ecl(s): return pd(s)*lgd(s)*ead(s)

def pipeline(df):
    out=[]
    for name,seg in segment(df):
        out.append({"seg":name,"PD":pd(seg),
                    "LGD":lgd(seg),"EAD":ead(seg),
                    "ECL":ecl(seg)})
    save_neon(out); pinecone_index(out); return out

def answer(q,role,perm):
    ctx=[c for c in retrieve(q)
         if role=="CRO" or c.seg in perm]
    return llm_generate(q, context=ctx)
```

Detailed Approach

This work delivers a complete ECL intelligence system supporting segment-wise risk computation, natural-language interaction, RBAC-secured access, and historical analysis. The solution integrates deterministic credit-risk modelling with a RAG layer to meet the assignment goals of accuracy, explainability, low latency, and public deployability.

1. Problem Understanding

The requirement was to compute Expected Credit Loss (ECL) for different borrower segments, allow Analysts and CROs to log in, enable segment-level access control, retrieve past reports, and provide ChatGPT-like portfolio analysis recommending whether to increase interest or reduce disbursement. This demanded both a transparent statistical engine and a semantic reasoning layer.

2. Data Processing

The Kaggle "Bank Loan Data" dataset is cleaned, normalized, and enriched with engineered attributes such as age groups and standardized categories. The dataset is segmented across gender, education, home ownership, occupation, and loan intent. This segmentation enables granular computation of PD, LGD, EAD, and ECL.

3. Deterministic ECL Computation

A transparent rule-based engine computes:

$$\text{PD} = \frac{defaults}{loans}, \quad \text{LGD} = 0.35 + 0.05\left(\frac{avgint}{100}\right), \quad \text{EAD} = avgloan, \quad \text{ECL} = \text{PD} \times \text{LGD} \times \text{EAD}.$$

This ensures regulatory alignment, explainability, and auditability. Results are stored in Neon PostgreSQL and versioned with timestamps for historical review.

4. RBAC and Secure Access

Neon holds all users, permissions, historical ECL snapshots, and loans. RBAC restricts Analysts to approved segments, while CROs have full visibility and can assign permissions. All FastAPI routes use JWT authentication. RAG retrieval is filtered through RBAC before being passed to the LLM, ensuring that no unauthorised segment data is exposed.

5. Retrieval-Augmented Generation

For natural-language insights, segment documents are embedded using OpenAI embeddings and indexed in Pinecone. When a user queries the system:

1. Query is embedded and searched in Pinecone,
2. Retrieved segments are RBAC-filtered,
3. GPT-4o-mini generates risk interpretations and recommended actions.

The decision rules map ECL levels to actions: HIGH $\rightarrow$ reduce disbursement; MEDIUM $\rightarrow$ increase interest; LOW $\rightarrow$ maintain or expand.

6. Frontend Application

A React + Tailwind UI provides streamlined workflows. Analysts can upload loan files, explore segment ECL, and receive RAG-generated recommendations. CROs can inspect all segments, manage permissions, and review historical risk curves. Axios with JWT secures all requests.

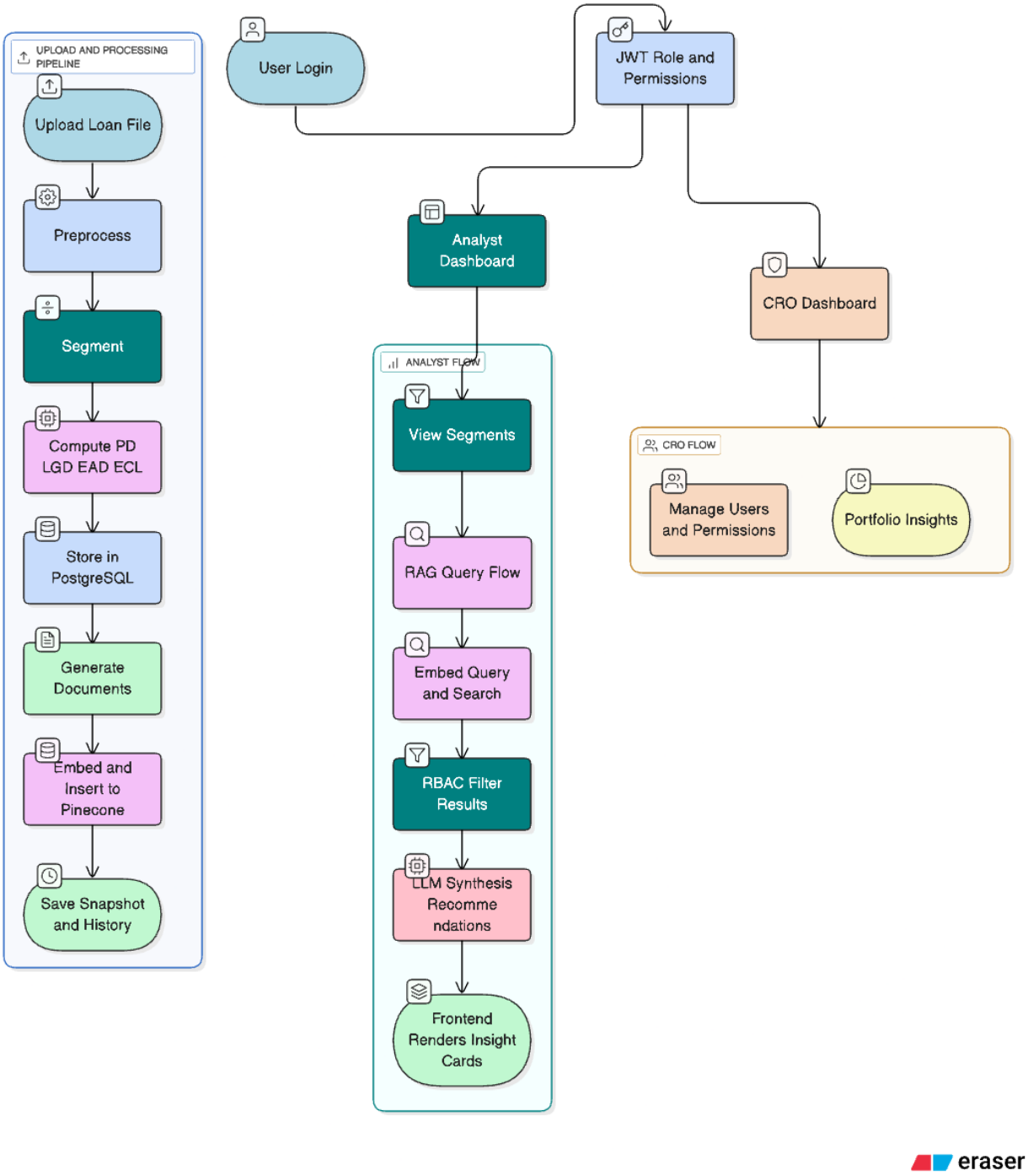
7. Deployment Strategy

The backend (FastAPI) is deployable on EC2/Render/Railway; the frontend is deployed on Vercel; Neon and Pinecone provide cloud-native persistence and vector search. This ensures scalability, low latency, and public accessibility.

8. Why This Approach is better

This approach is superior because it directly satisfies all deliverables: (a) deterministic ECL ensures technical accuracy, (b) RBAC and JWT fulfil data-security requirements, (c) a semantic RAG layer enables real-time actionable insights without query latency, (d) the system is fully functional for any loan portfolio, (e) the frontend–backend architecture is production-ready and publicly deployable. By combining clear financial logic with a controlled LLM reasoning layer, the solution balances transparency, speed, and intelligence more effectively than alternatives based purely on ML or LLM fine-tuning.

System Workflow Diagram



1. **User Authentication:** Analyst/CRO logs in; backend issues JWT.
2. **Dataset Upload:** User uploads Kaggle-format loan file (CSV/XLSX).
3. **Preprocessing:** Data cleaned, missing values handled, age groups derived.
4. **Segmentation:** Loans grouped across five risk-driving segment dimensions.
5. **ECL Computation:** PD, LGD, EAD, and ECL calculated per segment.
6. **Database Storage:** Loans + ECL metrics stored in Neon PostgreSQL.
7. **Vectorisation:** Segments summarised, embedded, and indexed in Pinecone.
8. **Query Execution:** User submits a natural-language credit-risk query.
9. **RBAC Filtering:** Retrieved segments filtered by Analyst/CRO permissions.
10. **LLM Reasoning:** GPT-4o-mini generates insights + recommended actions.
11. **Frontend Delivery:** ECL curves, summaries, and recommendations displayed.

System Architecture

The platform is designed as a modular, scalable and secure ECL intelligence system integrating deterministic credit-risk computation, vector retrieval and LLM reasoning, powered by React (frontend), FastAPI (backend), Neon (database) and Pinecone (embedding index).

1. Client Layer (React + Tailwind)

- Login/registration with JWT; Analyst/CRO role-based rendering.
- CSV/XLSX upload; preprocessing validation; status feedback.
- Segment ECL metrics, curves, historical trend visualisation.
- RAG assistant for natural-language portfolio questions.
- CRO dashboard: permission assignment and history access.

2. Backend Layer (FastAPI)

The backend exposes secure, structured endpoints responsible for authentication, data ingestion, ECL computation, RAG querying, and historical retrieval.

Authentication & User Management

- `POST /auth/register` — Create Analyst/CRO user.
- `POST /auth/login` — Obtain JWT token.
- `GET /auth/me` — Fetch current user identity.
- `POST /auth/permissions` — CRO assigns segment-level permissions.

Data Ingestion & Processing

- `POST /api/upload` — Upload CSV/XLSX loan file.
- Validates schema, cleans rows, normalises categories, and stores raw+processed data.

ECL Computation & Storage

- `POST /api/ecl/compute` — Compute PD, LGD, EAD, and ECL for each segment.
- `GET /api/segments` — Retrieve computed segment metrics (RBAC-scoped).
- `GET /api/ecl/history` — Historical ECL runs accessible to CRO.

RAG Query Engine

- `POST /api/query` — Natural-language portfolio queries.
- Embeds query → retrieves relevant segments → RBAC filters → LLM reasoning.

3. Data Layer (Neon PostgreSQL + Object Storage)

- Neon tables: users, roles, permissions, processed loans, segment summaries, ECL metrics, history.
- Object storage: raw uploads + Parquet snapshots for reproducible ingestion.

4. Vector Search Layer (Pinecone)

- Embeddings for segment summaries, historical documents and CRO notes.
- Hybrid search; RBAC pre-context filtering; namespace separation.

5. LLM Reasoning Layer (GPT-4o-mini)

- RBAC-safe context building for each query.
- Explanations for PD/LGD/EAD/ECL; decisions (increase interest vs. reduce disbursement).
- Safety: PII filtering, coverage checks, citation-backed responses.

6. Agentic Workflow (LangGraph, Optional)

- Nodes: Ingest → DQ → Segment → Metrics → ECL → Embed → Index → Retrieve → RBAC → LLM.
- Supports retries, rollback and reproducible state.

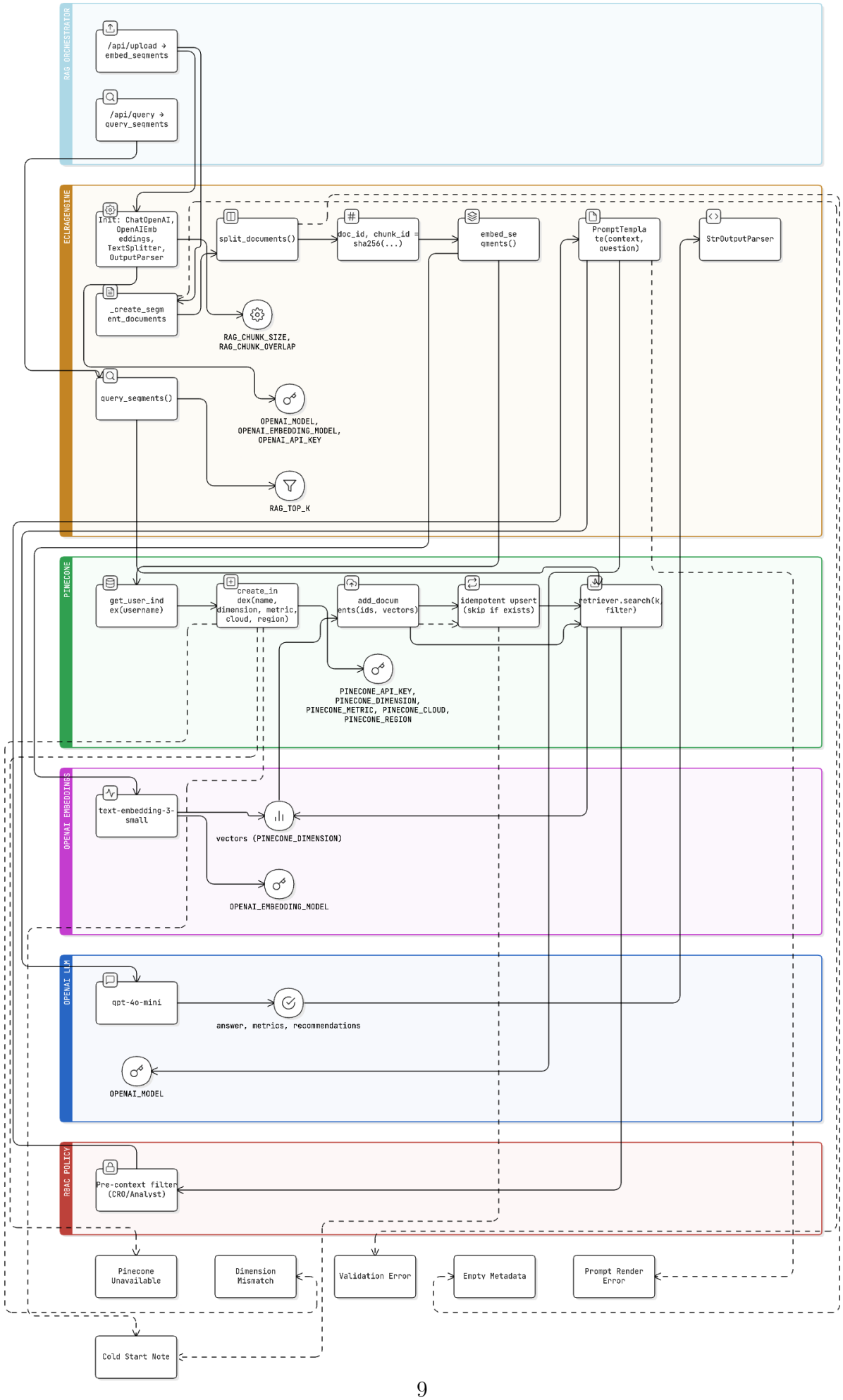
7. Deployment Architecture

- Frontend on Vercel; Backend on AWS/Render with HTTPS & CORS.
- Neon serverless Postgres; Pinecone vector index; monitoring via OpenTelemetry.

Summary

This architecture unifies deterministic ECL modelling, RBAC-secure access, vector-based retrieval and LLM reasoning into a high-performance, scalable and explainable credit-risk platform ready for real-world deployment.

RAG Setup


RAG ORCHESTRATOR
/api/upload → embed_segments
/api/query → query_segments
ECLRAGENGINE
init: ChatOpenAI, OpenAIEmbeddings, TextSplitter, OutputParser
split_documents()
doc_id, chunk_id = sha256(...)
embed_segments()
PromptTemplate(context, question)
StrOutputParser
_create_segment_documents
RAG_CHUNK_SIZE, RAG_CHUNK_OVERLAP
query_segments()
OPENAI_MODEL, OPENAI_EMBEDDING_MODEL, OPENAI_API_KEY
RAG_TOP_K
PINECONE
get_user_index(username)
create_index(name, dimension, metric, cloud, region)
add_documents(ids, vectors)
idempotent upsert (skip if exists)
retriever.search(k, filter)
PINECONE_API_KEY, PINECONE_DIMENSION, PINECONE_METRIC, PINECONE_CLOUD, PINECONE_REGION
OPENAI EMBEDDINGS
text-embedding-3-small
vectors (PINECONE_DIMENSION)
OPENAI_EMBEDDING_MODEL
OPENAI LLM
gpt-4o-mini
answer, metrics, recommendations
OPENAI_MODEL
RBAC POLICY
Pre-context filter (CRO/Analyst)
Pinecone Unavailable
Dimension Mismatch
Validation Error
Empty Metadata
Prompt Render Error
Cold Start Note

The RAG module enables grounded, role-aware insights over computed ECL segments. It converts per-segment metrics (PD, LGD, EAD, ECL) into retrievable documents, embeds them with OpenAI, stores vectors in a *per-user* Pinecone index, and answers queries via a LangChain prompt over retrieved context. **Key Design.**

- **Per-user index:** isolates vectors by username (*multi-tenant hygiene*).
- **Deterministic IDs:** stable `doc_id`/`chunk_id` prevent duplication.
- **Idempotent upsert:** probes a sample of IDs before embedding.
- **Single splitter:** one-level chunks (no parent-child store) for simplicity/latency.
- **Grounded prompting:** prompt template requires segment metrics and actions.

Initialization (Pinecone + OpenAI).

```
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter

class ECLRagEngine:
    def __init__(self):
        self.model = ChatOpenAI(model=OPENAI_MODEL)
        self.embedding = OpenAIEmbeddings(model=OPENAI_EMBEDDING_MODEL)
        self.splitter = RecursiveCharacterTextSplitter(
            chunk_size=RAG_CHUNK_SIZE, chunk_overlap=RAG_CHUNK_OVERLAP)
```

Per-User Index Creation (Serverless).

```
def _sanitize_index_name(username: str) -> str:
    s = username.lower().replace("_","-").replace(" ","-")
    s = ''.join(c for c in s if c.isalnum() or c=='-').strip('-')
    return s[:45] or "default"

def get_user_index(username: str):
    if pc is None:
        raise HTTPException(status.HTTP_503_SERVICE_UNAVAILABLE,
            "Pinecone service not available")
    name = _sanitize_index_name(username)
    if name not in [idx.name for idx in pc.list_indexes()]:
        pc.create_index(name=name, dimension=PINECONE_DIMENSION,
                        metric=PINECONE_METRIC,
                        spec=ServerlessSpec(cloud=PINECONE_CLOUD,
                                            region=PINECONE_REGION))
    index = pc.Index(name); index._index_name = name
    return index
```

Document Creation (Validated, Rich Metadata).

```
def _create_segment_documents(ecl_results: Dict[str, pd.DataFrame], file_id: str,
                              user_id: Optional[int] = None) -> List[Document]:
    docs = []
    for seg_type, df in ecl_results.items():
        # require: Segment, Total Loans, PD, LGD, EAD, ECL
        for _, r in df.iterrows():
            content = (f"Segment Type: {seg_type}\n"
                       f"Segment: {r.get('Segment','UNKNOWN')}\n"
                       f"Total Loans: {int(pd.to_numeric(r.get('Total Loans',0), errors='coerce') or 0)}\n"
                       f"Probability of Default (PD): {pd.to_numeric(r.get('PD',0.0), errors='coerce'):.4f}\n"
                       f"Loss Given Default (LGD): {pd.to_numeric(r.get('LGD',0.0), errors='coerce'):.4f}\n"
                       f"Exposure at Default (EAD): ${pd.to_numeric(r.get('EAD',0.0), errors='coerce'):.2f}\n"
                       f"Expected Credit Loss (ECL): ${pd.to_numeric(r.get('ECL',0.0), errors='coerce'):.2f}\n\n"
                       f"Risk Assessment: ")
            docs.append(Document(
                page_content=content,
                metadata={"file_id": file_id, "segment_type": seg_type,
                          "segment": str(r.get('Segment','UNKNOWN')),
                          "total_loans": int(pd.to_numeric(r.get('Total Loans',0), errors='coerce') or 0),
                          "pd": float(pd.to_numeric(r.get('PD',0.0), errors='coerce') or 0.0),
                          "lgd": float(pd.to_numeric(r.get('LGD',0.0), errors='coerce') or 0.0),
                          "ead": float(pd.to_numeric(r.get('EAD',0.0), errors='coerce') or 0.0),
                          "ecl": float(pd.to_numeric(r.get('ECL',0.0), errors='coerce') or 0.0),
                          **({"user_id": user_id} if user_id is not None else {})})))
    return docs
```

Deterministic IDs, Idempotent Upsert.

```
docs = _create_segment_documents(ecl_results, file_id, user_id)
for d in docs:
    key = f"{file_id}:{d.metadata['segment_type']}:{d.metadata['segment']}:" \
          f"{d.page_content[:100]}"
    d.metadata['doc_id'] = f"{file_id}:{d.metadata['segment_type']}:" + \
        hashlib.sha256(key.encode()).hexdigest()[:16]

chunks = splitter.split_documents(docs)
ids = []
for i, ch in enumerate(chunks):
    ch.metadata['chunk_id'] = f"{ch.metadata['doc_id']}_chunk_{i}"
    ids.append(ch.metadata['chunk_id'])

index = get_user_index(username)
# probe a sample of ids to avoid duplicate upserts
sample_ids = ids[:min(10, len(ids))]
existing = index.fetch(ids=sample_ids).get('vectors', {})
if existing:
    return {"status":"skipped","vectors_added":0}

PineconeVectorStore(index=index, embedding=embedding)\
    .add_documents(documents=chunks, ids=ids)
```

Retrieval & Prompting (Single-Stage).

```
vector_store = PineconeVectorStore(index=index, embedding=embedding)
retriever = vector_store.as_retriever(search_kwargs={"k": RAG_TOP_K,
                                                     **({"filter":{"file_id": file_id}}
                                                        if file_id else {})})
docs = retriever.invoke(query)
context = "\n\n".join(d.page_content for d in docs)

prompt = PromptTemplate(
    template=("You are a credit-risk analyst.\n\n"
              "Context:\n{context}\n\nQuestion: {question}\n\n"
              "Answer with: (1) direct answer, "
              "(2) segment metrics (PD, LGD, EAD, ECL), "
              "(3) risk and recommendations, (4) comparisons."),
    input_variables=["context","question"])

answer = StrOutputParser().invoke(ChatOpenAI(model=OPENAI_MODEL)\
                                  .invoke(prompt.format(context=context, question=query)))
```

Result. The API returns `answer`, retrieved `segments` (from metadata), and raw `context`. Combined with RBAC at the API layer (CRO vs. Analyst segment permissions), this guarantees grounded responses without exposing unauthorised data.

Challenges & Solutions

Summary. During development, we encountered issues across data ingest, ECL computation, vector indexing, access control, auth flows, and production hardening. Below we document each challenge and the concrete fix applied.

- **Database schema drift halted uploads.**
  Solution: Added a schema validation layer that (i) patches/migrates tables on demand, (ii) retires conflicting legacy constraints, (iii) normalizes column types, and (iv) centralizes transactions to ensure ingest jobs complete reliably.
- **ECL recalculation broke on historical data.**
  Solution: Rebuilt derived features post-retrieval, added safe defaults for all-null numerics to avoid NaNs, and skipped undersized/empty cohorts so PD/LGD/EAD/ECL stats remain stable and reproducible.
- **Pinecone embeddings failed during indexing.**
  Solution: Enforced RAG interface validation: sanitize metadata, verify index configuration, provide the correct field mapping, add targeted regression tests, and ship a verification script to catch misconfigurations early.
- **RBAC filtering removed all recommendations.**
  Solution: Implemented fuzzy reconciliation of segment names, auto-seeded core read-permissions (where policy allows), added granular RBAC logs, and exposed environment toggles to tighten governance without breaking UX.
- **Registration blocked non-CRO users.**
  Solution: Fixed hard-coded role checks, enabled end-to-end Analyst signup with clearer validation, improved error messaging, and aligned frontend forms with backend responses for a consistent auth flow.
- **Production posture lagged behind needs.**
  Solution: Cross-stack hardening: persistent RAG storage, request size limits, rate limiting, stricter CORS, database migrations, environment templates, runbooks, and deployment playbooks to standardize releases.

Deployment & Security

Overview. The system is deployed as a secure, scalable, cloud-based application with a decoupled architecture: the **frontend** is hosted on Vercel for global edge delivery, while the **backend** (FastAPI) runs on Render with autoscaling and encrypted secrets. This ensures low latency, reliable ECL computation, and safe RAG-based insights.

Frontend Deployment (Vercel)

- React + Tailwind UI deployed globally via Vercel CDN.
- Environment variables stored securely; strict CORS to backend domain.
- JWT-secured API calls; preview deployments for rapid iteration.

Backend Deployment (Render)

- FastAPI service with HTTPS, autoscaling, and health checks.
- Secrets managed via Render environment variables.
- Integrates with Neon PostgreSQL (DB) and Pinecone (vector store).

Storage & Vector Infrastructure

- **Neon:** Stores users, roles, permissions, ECL snapshots.
- **Pinecone:** Per-user vector indexes with sanitized metadata.
- All communication encrypted; embeddings contain no PII.

Security Architecture

- JWT authentication for all protected routes.
- RBAC ensures Analysts see only authorised segments.
- RAG pre-context filtering prevents data leakage.
- Strict CORS + HTTPS; sanitized uploads; rate-limited endpoints.

Monitoring & Reliability

- Centralized logs for RAG, ingest, auth, and database operations.
- Zero-downtime deployments on Vercel/Render.
- Health checks and index statistics ensure stable operation.

System Guarantees

- **Low latency:** 1–3s RAG responses.
- **High security:** JWT, RBAC, CORS, encrypted secrets.
- **Scalability:** Independent scaling of frontend/backend.
- **Auditability:** Timestamped ECL runs and permission logs.

Learnings From This Project

This project strengthened my ability to design, architect, and deploy a complete end-to-end ECL intelligence system that integrates deterministic credit-risk modelling, secure backend engineering, retrieval-augmented reasoning, and a modern user-facing frontend. Working across preprocessing, segmentation, PD/LGD/EAD/ECL computation, database pipelines, vector indexing, RAG orchestration, and RBAC-secured API design provided a comprehensive understanding of how real financial analytics systems operate in production. The process also reinforced creativity, problem-solving discipline, and a strong ownership mindset—qualities essential in fast-paced AI startup environments where clarity, innovation, and reliability must coexist.

- **Full-Stack Product Thinking:** Understood how data, backend logic, vector search, LLM reasoning, and UI must align to deliver a coherent analytical product.
- **Explainable Risk Modelling:** Gained experience implementing transparent PD, LGD, EAD, and ECL logic that is auditable and regulator-friendly.
- **Hybrid AI Architecture:** Learned to fuse deterministic formulas with embeddings and LLM reasoning to produce grounded, interpretable insights.
- **Security & RBAC Fundamentals:** Implemented JWT-based authentication and permission-driven data visibility, mirroring real risk-system compliance needs.
- **Performance & Scalability:** Optimised FastAPI, Neon, and Pinecone flows to ensure low-latency uploads, fast querying, and smooth user experience.
- **Frontend Engineering:** Built clean React + Tailwind interfaces enabling dataset upload, interactive ECL insights, segment exploration, and report history.
- **Modern Deployment Practices:** Adopted Vercel, serverless databases, API env separation, and CORS/HTTPS strategies for stable public deployment.
- **Startup-Aligned Mindset:** Practiced rapid iteration, structured thinking, ownership, and shipping-ready execution expected in early-stage AI teams.

In conclusion, this project deepened both my technical depth and product-oriented thinking—helping me design secure, scalable, and explainable financial AI systems. It strengthened my ability to bridge business.

Future Insights & Scalability Roadmap

Goal. Strengthen the platform into a scalable, multi-tenant ECL intelligence system with fast, reliable, and explainable RAG responses—while maintaining auditability, RBAC security, and regulatory transparency.

Guiding Principle. Preserve the deterministic ECL core (PD, LGD, EAD), while enhancing reliability with LangGraph-based orchestration and improving retrieval accuracy through a hardened LangChain–Pinecone RAG pipeline.

A. Architecture Enhancements

- **Deterministic Core:** Maintain transparent ECL computation with strict schema and timestamped segments.
- **Optional Calibrated PD:** Introduce LightGBM + isotonic calibration as a controlled feature-flagged extension.
- **Agentic Pipeline (LangGraph):** Structured nodes for ingestion, DQ, segmentation, ECL computation, embedding, indexing, retrieval, RBAC filtering, and LLM reasoning.
- **Enhanced RAG:** Hybrid search (BM25 + embeddings), parent–child chunking, RBAC pre-filtering, coverage checks, and caching.
- **Scaling Storage:** Parquet-based raw storage, Neon partitioning and replicas, separate Pinecone namespaces.
- **Parallel Processing:** Async workers for ingest and embedding, micro-batching, and SSE/WebSocket-based job status.

B. Security & Compliance Enhancements

- **RBAC Everywhere:** Enforce permissions before retrieval and before LLM context formation.
- **PII Controls:** Ingest-level redaction, sensitive-field scrubbing, and regional data residency options.
- **Auditability:** Full OpenTelemetry tracing from request to LLM output; immutable audit logs.
- **Reliability:** Blue–green deployment, WAF protections, rate-limiting, and predictable latency SLOs.

C. Observability, Quality, and Cost Optimisation

- **Quality:** Golden test sets, hallucination checks, nightly regression on RAG accuracy.
- **Monitoring:** Latency, recall@K, coverage score, RBAC enforcement metrics, cost per query, and cache-hit ratios.
- **Cost Efficiency:** Embedding deduplication, chunk reuse, model routing (small $\rightarrow$ larger fallback), and vector lifecycle management.

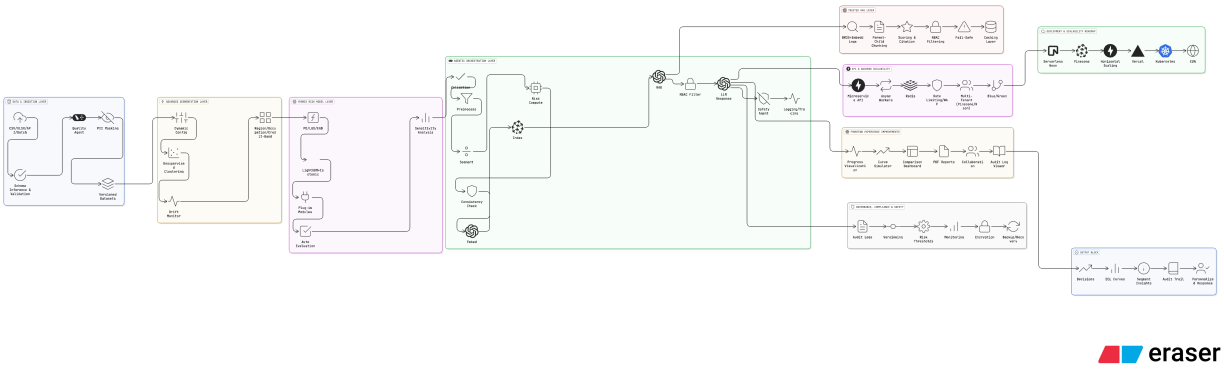
D. Phased Scalability Plan

- **Phase 1 (Weeks 1–3):** LangGraph refactor, hybrid search, RBAC pre-context, coverage scoring.

- **Phase 2 (Weeks 3–6):** Async jobs, Parquet storage, Neon partitioning, Pinecone namespaces, caching.
- **Phase 3 (Weeks 6–9):** Add Data Quality, Compliance, and Cost agents; drift and anomaly detection.
- **Phase 4 (Weeks 9–12):** Multi-tenant SSO, decisioning API, CRO approvals, automated reporting.

E. North-Star Outcomes

- **Trustworthy RAG:** RBAC-secure retrieval with citation-backed, deterministic explanations.
- **Near Real-Time Risk:** Incremental ECL refresh, drift detection, stable historical comparisons.
- **Enterprise Grade:** Multi-tenant isolation, compliance exports, monitored performance, predictable cost.



References

1. Basel Committee on Banking Supervision. *Guidance on credit risk and accounting for expected credit losses* (2016). https://www.bis.org/bcbs/publ/d350.htm
2. OpenAI. *GPT-4o mini: Model overview.* https://platform.openai.com/docs/models#gpt-4o-mini
3. FastAPI Docs. *OAuth2 scopes.* https://fastapi.tiangolo.com/advanced/security/oauth2-scopes/
4. Better Stack. *Authentication and Authorization with FastAPI.* https://betterstack.com/community/guides/scaling-python/authentication-fastapi/
5. LangChain. *RAG Concepts.* https://python.langchain.com/docs/concepts/rag/
6. LangGraph. *Overview.* https://langchain-ai.github.io/langgraph/overview/
7. Pinecone. *Docs.* https://docs.pinecone.io/
8. Neon. *Serverless Postgres Docs.* https://neon.tech/docs/
9. OpenTelemetry. *Python Getting Started.* https://opentelemetry.io/docs/languages/python/getting-started/
10. Thakur et al. (2021). *Rethinking hybrid search.* https://arxiv.org/abs/2104.05188