LegacyLens

Building RAG Systems for Legacy Enterprise Codebases

Before You Start: Pre-Search (30 Minutes)

Before writing any code, complete the Pre-Search methodology at the end of this document. This structured process uses AI to explore vector database options, embedding strategies, and RAG architecture decisions. Your Pre-Search output becomes part of your final submission.

This week emphasizes RAG architecture selection and implementation. Pre-Search helps you choose the right vector database and retrieval strategy for your use case.

Background

Enterprise systems running on COBOL, Fortran, and other legacy languages power critical infrastructure: banking transactions, insurance claims, government services, and scientific computing. These codebases contain decades of business logic, but few engineers understand them.

This project requires you to build a RAG-powered system that makes large legacy codebases queryable and understandable. You will work with real open source enterprise projects, implementing retrieval pipelines that help developers navigate unfamiliar code through natural language.

The focus is on RAG architecture—vector databases, embedding strategies, chunking approaches, and retrieval pipelines that actually work on complex codebases.

Gate: Behavioral and technical interviews required for Austin admission.

Project Overview

One-week sprint with three deadlines:

Checkpoint	Deadline	Focus
MVP	Tuesday (24 hours)	Basic RAG pipeline working
Early Submission	Friday (4 days)	Full feature set
Final	Sunday (7 days)	Polish, documentation, deployment

MVP Requirements (24 Hours)

Hard gate. All items required to pass:

- ☐ Ingest at least one legacy codebase (COBOL, Fortran, or similar)
- ☐ Chunk code files with syntax-aware splitting
- ☐ Generate embeddings for all chunks
- ☐ Store embeddings in a vector database
- ☐ Implement semantic search across the codebase
- ☐ Natural language query interface (CLI or web)
- ☐ Return relevant code snippets with file/line references
- ☐ Basic answer generation using retrieved context
- ☐ Deployed and publicly accessible

A simple RAG pipeline with accurate retrieval beats a complex system with irrelevant results.

Target Codebases

Choose ONE primary codebase from this list (or propose an alternative of similar scope):

Project	Language	Description
GnuCOBOL	COBOL	Open source COBOL compiler
GNU Fortran (gfortran)	Fortran	Fortran compiler in GCC
LAPACK	Fortran	Linear algebra library
BLAS	Fortran	Basic linear algebra subprograms

OpenCOBOL Contrib	COBOL	Sample COBOL programs and utilities
Custom proposal	Any legacy	Get approval before starting

Minimum codebase size: 10,000+ lines of code across 50+ files.

Core RAG Infrastructure

Ingestion Pipeline

Component	Requirements
File Discovery	Recursively scan codebase, filter by file extension
Preprocessing	Handle encoding issues, normalize whitespace, extract comments
Chunking	Syntax-aware splitting (functions, paragraphs, sections)
Metadata Extraction	File path, line numbers, function names, dependencies
Embedding Generation	Generate vectors for each chunk with chosen model
Storage	Insert into vector database with metadata

Retrieval Pipeline

Component	Requirements
Query Processing	Parse natural language, extract intent and entities
Embedding	Convert query to vector using same model as ingestion
Similarity Search	Find top-k most similar chunks
Re-ranking	Optional: reorder results by relevance score
Context Assembly	Combine retrieved chunks with surrounding context
Answer Generation	LLM generates response using retrieved context

Chunking Strategies

Legacy code requires specialized chunking. Document your approach:

Strategy	Use Case
Function-level	Each function/subroutine as a chunk
Paragraph-level (COBOL)	COBOL PARAGRAPH as natural boundary
Fixed-size + overlap	Fallback for unstructured sections

Semantic splitting	Use LLM to identify logical boundaries
Hierarchical	Multiple granularities (file → section → function)

Testing Scenarios

We will test with queries like:

1. "Where is the main entry point of this program?"
2. "What functions modify the CUSTOMER-RECORD?"
3. "Explain what the CALCULATE-INTEREST paragraph does"
4. "Find all file I/O operations"
5. "What are the dependencies of MODULE-X?"
6. "Show me error handling patterns in this codebase"

Performance Targets

Metric	Target
Query latency	<3 seconds end-to-end
Retrieval precision	>70% relevant chunks in top-5
Codebase coverage	100% of files indexed
Ingestion throughput	10,000+ LOC in <5 minutes
Answer accuracy	Correct file/line references

Required Features

Query Interface

Build a usable interface for querying the codebase:

- Natural language input for questions about the code
- Display retrieved code snippets with syntax highlighting
- Show file paths and line numbers for each result
- Confidence/relevance scores for retrieved chunks
- Generated explanation/answer from LLM
- Ability to drill down into full file context

Code Understanding Features

Implement at least 4 of the following:

Feature	Description
Code Explanation	Explain what a function/section does in plain English
Dependency Mapping	Show what calls what, data flow between modules
Pattern Detection	Find similar code patterns across the codebase
Impact Analysis	What would be affected if this code changes?
Documentation Gen	Generate documentation for undocumented code
Translation Hints	Suggest modern language equivalents
Bug Pattern Search	Find potential issues based on known patterns
Business Logic Extract	Identify and explain business rules in code

Vector Database Selection

Choose ONE vector database. Document your selection rationale:

Database	Hosting	Best For	Considerations
Pinecone	Managed cloud	Production scale	Free tier available, easy setup
Weaviate	Self-host or cloud	Hybrid search	GraphQL API, modules
Qdrant	Self-host or cloud	Filtering	Rust-based, fast
ChromaDB	Embedded/self-host	Prototyping	Simple API, local dev

pgvector	PostgreSQL extension	Existing Postgres	Familiar SQL interface
Milvus	Self-host or Zilliz	Large scale	GPU acceleration

Embedding Models

Choose an embedding model appropriate for code:

Model	Dimensions	Notes
OpenAI text-embedding-3-small	1536	Good balance of cost/quality
OpenAI text-embedding-3-large	3072	Higher quality, higher cost
Voyage Code 2	1536	Optimized for code
Cohere embed-english-v3	1024	Good for English text
sentence-transformers (local)	varies	Free, runs locally

AI Cost Analysis (Required)

Understanding AI costs is critical for production applications. Submit a cost analysis covering:

Development & Testing Costs

Track and report your actual spend during development:

- Embedding API costs (total tokens embedded)
- LLM API costs for answer generation
- Vector database hosting/usage costs
- Total development spend breakdown

Production Cost Projections

Estimate monthly costs at different user scales:

100 Users	1,000 Users	10,000 Users	100,000 Users
$___/month	$___/month	$___/month	$___/month

Include assumptions: queries per user per day, average tokens per query, embedding costs for new code additions, vector DB storage costs at scale.

RAG Architecture Documentation (Required)

Submit a 1-2 page document covering:

Section	Content
Vector DB Selection	Why you chose this database, tradeoffs considered
Embedding Strategy	Model choice, why it fits code understanding
Chunking Approach	How you split legacy code, boundary detection
Retrieval Pipeline	Query flow, re-ranking, context assembly
Failure Modes	What doesn't work well, edge cases discovered
Performance Results	Actual latency, precision metrics, examples

Technical Stack

Recommended Path

Layer	Technology
Vector Database	Pinecone, Weaviate, or Qdrant

Embeddings	OpenAI text-embedding-3-small or Voyage Code 2
LLM	GPT-4, Claude, or open source (Llama, Mistral)
Framework	LangChain, LlamaIndex, or custom pipeline
Backend	Node.js/Express, Python/FastAPI, or serverless
Frontend	React, Next.js, or CLI interface
Deployment	Vercel, Railway, or cloud provider

RAG Frameworks

Framework	Best For
LangChain	Flexible pipelines, many integrations, good docs
LlamaIndex	Document-focused RAG, structured data
Haystack	Production pipelines, evaluation tools
Custom	Full control, learning exercise

Use whatever stack helps you ship. Complete the Pre-Search process to make informed decisions.

Build Strategy

Priority Order

7. Codebase selection — Pick and download your target repository
8. Basic ingestion — Read files, simple chunking, generate embeddings
9. Vector storage — Insert chunks, verify retrieval works
10. Query interface — Accept questions, return raw results
11. Answer generation — Add LLM to synthesize responses
12. Chunking refinement — Improve syntax-aware splitting
13. Advanced features — Implement 4+ code understanding features
14. Evaluation — Measure precision, document failure modes

Critical Guidance

- Get basic retrieval working before optimizing chunking
- Test with real questions a developer would ask
- Measure retrieval quality before adding complexity
- Document failure cases—they inform architecture decisions
- Keep embedding dimensions consistent across ingestion and query

Submission Requirements

Deadline: Sunday 10:59 PM CT

Deliverable	Requirements
GitHub Repository	Setup guide, architecture overview, deployed link
Demo Video (3-5 min)	Show queries, retrieval results, answer generation
Pre-Search Document	Completed checklist from Phase 1-3
RAG Architecture Doc	1-2 page breakdown using template above
AI Cost Analysis	Dev spend + projections for 100/1K/10K/100K users
Deployed Application	Publicly accessible query interface
Social Post	Share on X or LinkedIn: description, features, demo/screenshots, tag @GauntletAI

Interview Preparation

This week includes interviews for Austin admission. Be prepared to discuss:

Technical Topics

- Why you chose your vector database
- Chunking strategy tradeoffs
- Embedding model selection rationale
- How you handle retrieval failures
- Performance optimization decisions

Mindset & Growth

- How you approached ambiguity in the project
- Times you pivoted based on failure
- What you learned about yourself this week
- How you handle pressure and uncertainty

Final Note

A simple RAG pipeline with accurate, relevant retrieval beats a complex system that returns garbage.

Interviews are required for Austin admission.

Appendix: Pre-Search Checklist

Complete this before writing code. Save your AI conversation as a reference document.

Phase 1: Define Your Constraints

1. Scale & Load Profile

- How large is your target codebase (LOC, file count)?
- Expected query volume?
- Batch ingestion or incremental updates?
- Latency requirements for queries?

2. Budget & Cost Ceiling

- Vector database hosting costs?
- Embedding API costs (per token)?
- LLM API costs for answer generation?
- Where will you trade money for time?

3. Time to Ship

- MVP timeline?
- Which features are must-have vs nice-to-have?
- Framework learning curve acceptable?

4. Data Sensitivity

- Is the codebase open source or proprietary?
- Can you send code to external APIs?
- Data residency requirements?

5. Team & Skill Constraints

- Familiarity with vector databases?
- Experience with RAG frameworks (LangChain, LlamaIndex)?
- Comfort with the target legacy language?

Phase 2: Architecture Discovery

6. Vector Database Selection

- Managed vs self-hosted?
- Filtering and metadata requirements?
- Hybrid search (vector + keyword) needed?
- Scaling characteristics?

7. Embedding Strategy

- Code-specific vs general-purpose model?
- Dimension size tradeoffs?
- Local vs API-based embedding?
- Batch processing approach?

8. Chunking Approach

- Syntax-aware vs fixed-size?
- Optimal chunk size for your embedding model?
- Overlap strategy?
- Metadata to preserve?

9. Retrieval Pipeline

- Top-k value for similarity search?
- Re-ranking approach?
- Context window management?
- Multi-query or query expansion?

10. Answer Generation

- Which LLM for synthesis?
- Prompt template design?
- Citation/reference formatting?
- Streaming vs batch response?

11. Framework Selection

- LangChain vs LlamaIndex vs custom?
- Evaluation and observability needs?
- Integration requirements?

Phase 3: Post-Stack Refinement

12. Failure Mode Analysis

- What happens when retrieval finds nothing relevant?
- How to handle ambiguous queries?
- Rate limiting and error handling?

13. Evaluation Strategy

- How to measure retrieval precision?
- Ground truth dataset for testing?
- User feedback collection?

14. Performance Optimization

- Caching strategy for embeddings?

- Index optimization?
- Query preprocessing?

15. Observability

- Logging for debugging retrieval issues?
- Metrics to track (latency, precision, usage)?
- Alerting needs?

16. Deployment & DevOps

- CI/CD for index updates?
- Environment management?
- Secrets handling for API keys?