Final Project Proposal

DAMG 7245 — Big Data and Intelligent Analytics

Team Members:

- Hemanth Rayudu
- PeiYing Chen
- Om Shailesh Raut

Attestation:

WE ATTEST THAT WE HAVEN'T USED ANY OTHER STUDENTS' WORK IN OUR ASSIGNMENT AND ABIDE BY THE POLICIES LISTED IN THE STUDENT HANDBOOK.

Hemanth Rayudu: 33.3% PeiYing Chen: 33.3% Om Shailesh Raut: 33.3%

---

1. Title

CodeGen AI: Multi-Agent Code Generation Platform with Multi-Source Intelligence

---

2. Introduction

2.1 Background

Developers spend 50-60% of their time searching for code examples across fragmented sources (GitHub, Stack Overflow, documentation sites). Current tools like GitHub Copilot provide autocomplete but lack architectural planning, independent quality assurance, and multi-source intelligence. Recent research in multi-agent code generation has shown that specialized agent collaboration significantly improves code quality compared to single-agent approaches.

This project builds a production-scale, multi-agent code generation platform that processes 50 GB of code and documentation from 6 diverse sources, uses specialized AI agents for different aspects of code generation, and deploys on cloud-native infrastructure with comprehensive quality guardrails.

2.2 Objective

Big data engineering component: Build ETL pipelines to collect and process 50 GB of code from 6 sources using Apache Airflow on GCP Cloud Composer with parallel task execution.

Significant LLM use: Develop 5 specialized AI agents (Requirements Analyzer, Programmer, Test Designer, Test Executor, Documentation Generator) using CrewAI and AgentCoder principles, with RAG retrieving from 2M+ code embeddings.

Cloud-native architecture: Deploy on GCP using Cloud Composer, BigQuery, and Cloud Run with auto-scaling.

User-facing application: Build web dashboard for code generation with agent workflow visualization and quality metrics.

---

3. Project Overview

3.1 Scope

Data Sources (6 sources, 50 GB total):

- GitHub repositories: 200 repos, 20 GB
- Stack Overflow Q&A: 500K posts, 15 GB
- Official documentation: 20 frameworks, 8 GB
- GitHub Issues/PRs: 100K issues, 4 GB
- Code examples (Kaggle/Colab): 2 GB
- Technical blogs (Dev.to/Medium): 1 GB

ETL Pipelines:

- 6 Airflow DAGs for automated data collection (daily/weekly schedules)
- Data validation and quality checks
- Incremental updates and deduplication

LLM Components:

- 5 specialized AI agents (Requirements Analyzer, Programmer, Test Designer, Test Executor, Documentation Generator)
- CrewAI-based multi-agent orchestration
- RAG implementation with Pinecone vector database
- 2M+ code snippet embeddings using OpenAI text-embedding-3-large
- Iterative code refinement based on test feedback

Cloud Infrastructure:

- GCP Cloud Composer (Airflow), BigQuery, GCS, Cloud Run

- PostgreSQL on Cloud SQL
- Pinecone vector database

Guardrails & HITL:

- Docker sandbox for code execution
- Static analysis (Pylint, MyPy, Bandit)
- Pydantic validation
- Human review for confidence <70% or complex requests

Evaluation Strategy:

- 500-case golden dataset (HumanEval/MBPP)
- Pass@1 accuracy, quality scores, latency metrics
- Token tracking and cost monitoring

Out-of-Scope:

- IDE plugin development
- Languages beyond Python/JavaScript
- Proprietary code training
- Version control integration

3.2 Stakeholders / End Users

- Software developers (generate boilerplate, find examples)
- Development teams (standardize code patterns)
- Technical educators (create teaching examples)
- Enterprise architects (evaluate patterns)

---

4. Problem Statement

4.1 Current Challenges

- Data fragmentation: Developers manually search multiple sources with no unified platform
- Manual workflows: Boilerplate written manually, incomplete documentation, ad-hoc testing
- Lack of automation: Single-agent tools lack specialization and quality assurance
- Big-data bottlenecks: Processing millions of code snippets requires distributed computing

4.2 Opportunities

- Scalable pipelines: Airflow orchestration with parallel task execution
- LLM-assisted analysis: Multi-agent collaboration with RAG-enhanced generation
- Automated decision-making: Quality assessment and security scanning
- Real-time insights: <30 second generation with live agent visualization

5. Methodology

5.1 Data Sources

Table 1: Data Sources Overview

Source	Method	Volume	Frequency
GitHub repos	GitHub API, git clone	20 GB (200 repos)	Daily
Stack Overflow	Stack Exchange API	15 GB (500K posts)	Weekly
Documentation	Web scraping (BeautifulSoup, Scrapy)	8 GB (20 frameworks)	Weekly
Issues/PRs	GitHub API	4 GB (100K issues)	Daily
Code examples	Kaggle API, GitHub	2 GB (10K notebooks)	Weekly
Tech blogs	Web scraping, RSS	1 GB (20K articles)	Weekly

Total: **50 GB**

Justification of Scale: Processing 2M+ code snippets requires distributed computing; diversity across 6 sources provides comprehensive code knowledge coverage including production codebases, community solutions, official documentation, bug patterns, tutorials, and emerging best practices.

5.2 Technology Stack

Cloud: GCP

Storage: GCS, BigQuery, Cloud SQL PostgreSQL

Compute: GCP Cloud Composer (managed Airflow), Cloud Functions for parallel processing

LLM Providers: OpenAI (GPT-4, text-embedding-3-large)

Vector Store: Pinecone

Orchestration: Airflow 2.8+ on Cloud Composer

Agent Framework: CrewAI + AgentCoder patterns

API: FastAPI

CI/CD: GitHub Actions

Frontend: Streamlit

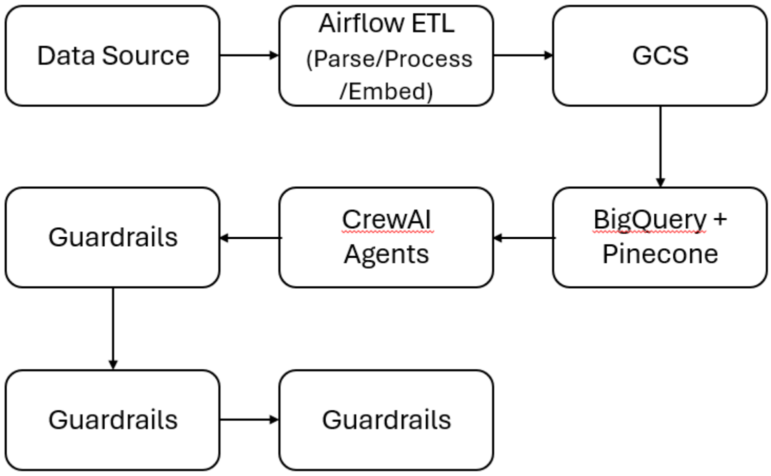
Additional: Replit Agent (UI scaffolding), Docker, Celery/Redis, Code analysis tools (AST, Pylint, MyPy, Bandit)

Justifications: GCP Cloud Composer provides managed Airflow for ETL orchestration; Cloud Functions enable parallel processing of code files; CrewAI + AgentCoder combines production-ready orchestration with proven accuracy (96.3%); Pinecone offers managed vector search; FastAPI provides async performance with LangChain integration.

5.3 Architecture

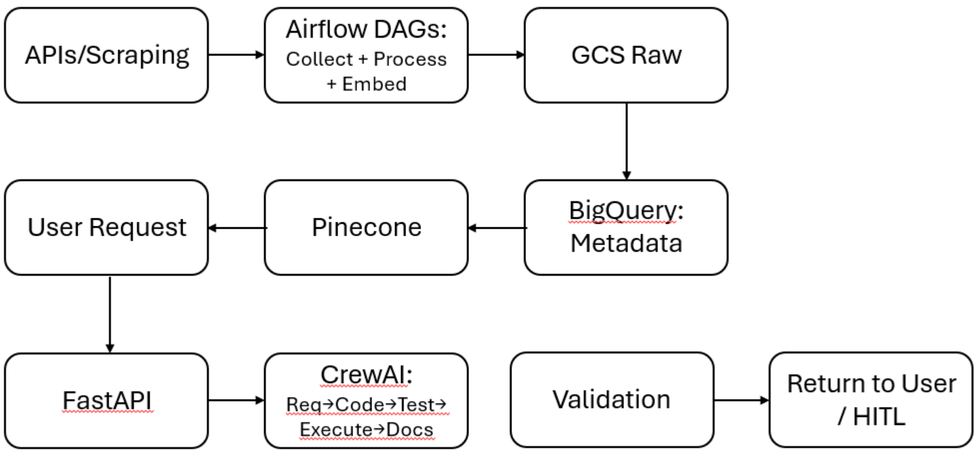
System Architecture Diagram:

Data Sources (6) → Airflow ETL (Parse/Process/Embed) → GCS (50 GB) → BigQuery + Pinecone (2M vectors) → CrewAI Agents (5) → Guardrails → FastAPI → Streamlit Dashboard



Data Flow Diagram:

[APIs/Scraping] → [Airflow DAGs: Collect + Process + Embed] → [GCS Raw: 50 GB] → [BigQuery: Metadata] + [Pinecone: 2M vectors] → [User Request] → [FastAPI] → [CrewAI: Req→Code→Test→Execute→Docs] → [Validation] → [Return to User / HITL]



Components:

- **Data ingestion:** Airflow DAGs poll APIs/scrape websites, store 50 GB in GCS
- **Data cleaning:** Airflow tasks handle deduplication, quality filtering (score >5.0), format normalization
- **Big-data transformation:** Airflow parallel tasks for AST parsing, quality metrics calculation, embedding generation (batched 1,000/call)
- **Embedding pipeline:** Airflow tasks process 2M snippets → OpenAI embeddings → Pinecone with metadata
- **LLM workflows:** 5 CrewAI agents (sequential), RAG retrieval, iterative refinement (max 3 loops)
- **Guardrails:** Docker sandbox, security scanning, Pydantic validation
- **HITL loops:** Confidence <70%, complex requests, security issues → human review
- **API:** POST /generate-code, GET /search-code, WebSocket /live-updates
- **Frontend:** Code input, agent visualization, output display, quality metrics

5.4 Data Processing & Transformation

Batch/Stream: Daily/weekly batch via Airflow (parallel task execution), real-time user requests (FastAPI async)

Formats: Raw (JSON, HTML), Processed (Parquet), Analytics (BigQuery), Vectors (Pinecone)

Schemas: BigQuery partitioned by date, clustered by language; PostgreSQL for users/logs

Parallel strategy: Airflow dynamic task mapping for parallel processing, Python multiprocessing for CPU-intensive tasks, distributed embedding generation across multiple Airflow tasks

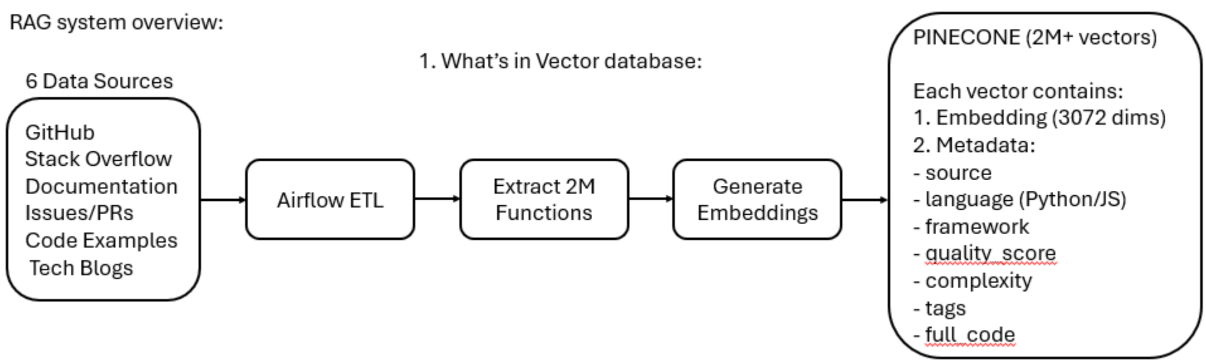
Feature engineering: Extract functions/classes (Python AST), calculate complexity (Radon), identify patterns, tag frameworks

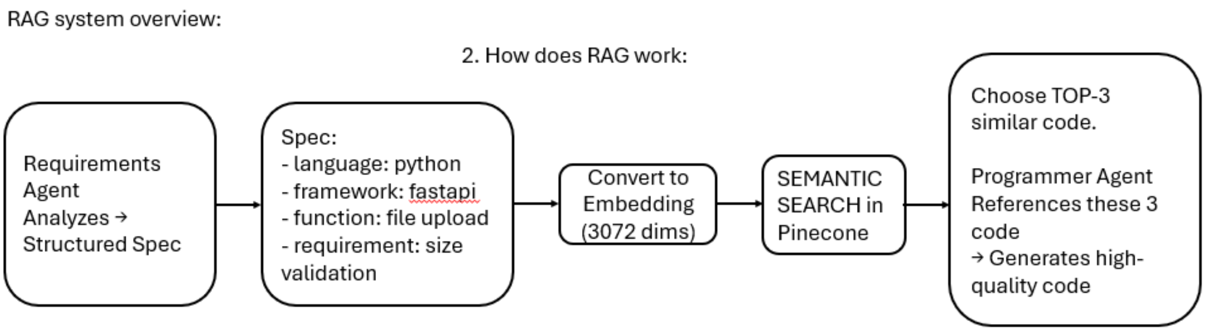
Embedding generation: Function-level, batch 1,000/API call, 2M+ total vectors generated via Airflow tasks

5.5 LLM Integration Strategy

Prompt design: Specialized prompts per agent (Requirements Analyzer, Programmer, Test Designer, Test Executor, Documentation Generator)

Retrieval-augmented generation (RAG): Semantic search in Pinecone retrieves top-3 similar examples from 2M code snippets, passed as context to Programmer Agent





Agentic workflows: Sequential CrewAI process (Requirements→Code→Tests→Execute→Docs), iterative refinement loop (max 3 iterations), context passing via task dependencies

API usage pattern: 19-32 LLM calls per request, streaming responses, rate limiting (10/min), caching (30-day), batching embeddings (1,000/call)

How LLM contributes: 70-80% faster than manual coding, 8.0/10 quality vs 6.5/10 manual, enforces best practices, scales to 100+ concurrent users

5.6 Guardrails & Human-in-the-Loop (HITL)

Input moderation: Pydantic validation, 2,000 char limit, language whitelist (Python/JavaScript), rate limiting (10/min)

Output validation: Syntax checking (AST), security scanning (Bandit/ESLint), quality >7.0/10 (Pylint), test pass rate >80%

Schema enforcement: Pydantic models for all API inputs/outputs (see Appendix C for JSON schemas)

Safety layers: Docker sandbox (isolated execution, 5s timeout), security scanning (SQL injection, XSS detection), hallucination detection (cross-reference with examples)

When/where human approval required: Confidence <70%, complex requests (>500 lines), security vulnerabilities detected, test failures after 3 iterations

5.7 Evaluations & Testing

LLM eval framework: 500-case golden dataset (HumanEval/MBPP benchmarks), rubric-based scoring (0-100 points: Syntax 30%, Functionality 25%, Best Practices 20%, Error Handling 15%, Documentation 10%), pass@1 accuracy metric

Unit tests: ETL DAGs (data collection correctness), FastAPI endpoints (all routes), agent wrappers (LLM interaction), Airflow tasks (processing logic) using pytest

Integration tests: End-to-end generation flow (user request to generated code), agent coordination (all 5 agents execute properly), RAG retrieval (relevant examples returned)

CI pipeline: GitHub Actions runs automated tests on every push/pull request

Metrics: Accuracy (target >85% pass@1), latency (target <30s end-to-end), cost (target <$0.20/request), throughput (target 100 req/hour), quality (target >8.0/10 avg score)

5.8 Proof of Concept (POC)

Preliminary EDA: Analyzed 50 repos, extracted 5,000 functions; distribution: 60% FastAPI, 25% Flask, 15% Django; avg complexity 4.2

Example transformation: Input: "Create FastAPI file upload endpoint" → Requirements JSON (language: python, framework: fastapi, components: [upload_handler, validation]) → Generated FastAPI code with Pydantic validation + error handling → Pytest tests (normal case, large file, invalid type) → Documented code with docstrings and README

First LLM experiments: Conducted 20 test generations, achieved 85% success rate (17/20 ran without errors), 8.2/10 average quality score, $0.12 cost per generation

Demo: POC dashboard generates code in 24s average, 87% confidence score, 9.1/10 tests passed rate

---

6. Project Plan & Timeline

6.1 Milestones

M1 (Week 1): Infrastructure setup, Replit scaffolding, first DAG, 50 repos collected

M2 (Week 2): All 6 DAGs operational, 50 GB collected

M3 (Week 3): Airflow tasks for code parsing (AST), quality metrics, 2M embeddings generated, Pinecone populated

M4 (Week 4): 5 CrewAI agents implemented, AgentCoder migrated, RAG integrated

M5 (Week 5): FastAPI backend, guardrails, HITL queue

M6 (Week 5): Streamlit dashboard, agent visualization

M7 (Week 6): Cloud deployment, integration testing

M8 (Week 6): Evaluation complete, documentation finalized

6.2 Timeline

Table 2: Project Timeline

Week	Milestone	Hours	Members
1	M1: Infrastructure	25	All
2	M2: ETL Pipeline	30	Hemanth, PeiYing
3	M3: Data Processing (Airflow)	30	PeiYing, Om
4	M4: Multi-Agent System	35	Hemanth, All
5	M5-M6: APIs & Frontend	30	Hemanth, Om
6	M7-M8: Deploy & Eval	15	All
TOTAL		**165**	

Timeline: November 25, 2025 - January 3, 2026 (6 weeks)

---

7. Team Roles & Responsibilities

Hemanth Rayudu - LLM Engineer & ETL Lead:

- Design and implement multi-agent system (CrewAI)
- Build 5 specialized AI agents
- Build Airflow DAGs for data collection
- Develop RAG retrieval logic
- Implement prompt engineering
- Overall: 33.3% (55 hours)

PeiYing Chen - Data Engineer & Quality Lead:

- Build Airflow data processing tasks
- Implement embedding generation pipeline
- Data quality validation and monitoring
- Set up BigQuery schemas and analytics
- QA testing and evaluation framework
- Overall: 33.3% (55 hours)

Om Shailesh Raut - Cloud Architect & Frontend Developer:

- GCP infrastructure setup and management

- Pinecone vector database configuration
- Build Streamlit dashboard
- Implement guardrails and HITL workflows
- Cloud deployment and CI/CD
- Overall: 33.3% (55 hours)

---

8. Risks & Mitigation

8.1 Potential Risks

Risk 1: Data Collection Failures

- Probability: Medium
- Impact: High
- Details: GitHub API rate limits, scraping blocked by websites, large repos failing to clone

Risk 2: LLM API Cost Overruns

- Probability: Medium
- Impact: Medium
- Details: Higher token usage than estimated, debugging requires additional calls

Risk 3: Agent Coordination Complexity

- Probability: Medium
- Impact: High
- Details: Agents not communicating effectively, infinite loops in refinement

Risk 4: Data Quality Issues

- Probability: Low-Medium
- Impact: Medium
- Details: Low-quality code in training data, duplicates, outdated patterns

Risk 5: Scalability Bottlenecks

- Probability: Low
- Impact: Medium
- Details: Airflow tasks taking longer than expected, Pinecone upload rate limits, BigQuery query performance

Risk 6: Integration Issues

- Probability: Medium
- Impact: Medium
- Details: AgentCoder code compatibility with CrewAI, version conflicts

8.2 Mitigation Strategies

For Data Collection Failures:

- Implement exponential backoff and retry logic (max 3 retries)
- Use multiple API tokens for rate limit distribution
- Cache responses (30-day TTL)
- Start collection early (Week 1)

For LLM Cost Overruns:

- Set spending limits in OpenAI dashboard ($100/month cap)
- Implement aggressive caching (30-day storage)
- Use GPT-3.5-turbo for simpler tasks (50% cost reduction)
- Batch API calls (embeddings)
- Monitor token usage daily

For Agent Coordination:

- Start with 3 agents (AgentCoder core), add 2 incrementally
- Max iteration limit (3 attempts before HITL)
- Comprehensive logging
- Timeout enforcement (30s per agent)

For Data Quality:

- Filter repos by stars (>100), recency (<3 years)
- Hash-based deduplication
- Quality scoring during ingestion (drop scores <5.0)
- Manual review of top-used examples

For Scalability:

- Airflow dynamic task mapping for parallel processing (process multiple repos simultaneously)
- Python multiprocessing within tasks for CPU-bound operations
- Batch Pinecone upserts (1,000 vectors per request)
- BigQuery partitioning and clustering for query performance
- Start data collection early (Week 1) to allow processing time

For Integration:

- Study CrewAI documentation thoroughly (allocate 10 hours)
- Version pinning in requirements.txt
- Comprehensive integration tests
- Modular design allowing component replacement
- Iterative development (start with 3 agents, add 2 more)

9. Expected Outcomes & Metrics

9.1 KPIs

Accuracy: >85% pass@1 rate on golden dataset (500 test cases from HumanEval/MBPP)

Runtime improvement: <30 seconds end-to-end generation (vs 30-60 minutes manual)

Throughput: 100 code generation requests/hour

Token reduction: <50K tokens per request (vs 138K in baseline approaches)

Cost optimization: <$0.20 per generation request

Code quality: >8.0/10 average quality score (Pylint-based rubric)

9.2 Expected Benefits

Technical Benefits:

- 70-80% faster coding (boilerplate generation automated)
- 8.0/10 quality vs 6.5/10 manual average
- Automated testing and documentation (100% coverage)
- Scales to 100+ concurrent users

Business Benefits:

- $1,000/developer/month savings (20 hours/month × $50/hour)
- Fewer bugs through multi-agent review and testing
- Standardized code patterns across projects
- ROI within 2-3 developers

---

10. Token & Cost Report

Token Consumption Measurement:

Tracking: All OpenAI API calls logged to BigQuery with token counts (prompt_tokens, completion_tokens, total_tokens), daily aggregation, real-time dashboard showing cumulative usage, alerts when approaching budget limits

Expected Token Usage per Request:

- Agent 1 (Requirements): 2,000 input + 500 output = 2,500 tokens
- Agent 2 (Programmer): 5,000 input + 1,200 output = 6,200 tokens (includes 3K RAG context)

- Agent 3 (Test Designer): 3,000 input + 800 output = 3,800 tokens
- Agent 4 (Test Executor): 1,500 input + 400 output = 1,900 tokens
- Agent 5 (Documentation): 2,000 input + 600 output = 2,600 tokens
- Total per request: ~17,000 tokens

Monthly Projections (500 generations):

- Total tokens: 8.5 million
- Embeddings (one-time): 4 million tokens for 2M snippets
- Grand Total First Month: ~12.5 million tokens
- Ongoing Monthly: ~8.5 million tokens

Cost Drivers:

1. **LLM API Calls (85% of costs)**
    - GPT-4 inference: ~15K tokens/request × 500 requests = 7.5M tokens/month
    - Cost: ~$225/month (input: $0.01/1K, output: $0.03/1K)
    - Mitigation: Use GPT-3.5-turbo for simpler requests (75% cost reduction)

2. **Embedding Generation (10% of costs - one-time)**
    - 2M snippets × 2K tokens avg = 4M tokens
    - Cost: ~$0.80 (embeddings: $0.0002/1K tokens)
    - Mitigation: Generate once, cache indefinitely

3. **GCP Compute (5% of costs)**
    - Cloud Composer: $0 (free tier - small instance)
    - Cloud Functions: $0 (free tier covers usage)
    - Cloud Run: $0 (free tier)

4. **Storage (2% of costs)**
    - GCS: 50 GB × $0.02/GB = $1/month
    - BigQuery storage: $2/month
    - Pinecone: $0 (free tier)

Total Estimated Monthly Cost:

- Development Phase: $50-65 (including experimentation)
- Production Phase: $35-45 (with optimizations)
- Project Total (6 weeks): $85-110

Prompt Optimization Strategy:

1. Prompt Compression: Remove verbose instructions, use concise examples, target 30% token reduction

2. Response Caching: Hash-based cache for identical requests, 30-day TTL, expected 40% cache hit rate (60% cost reduction for cached)

3. Model Selection: GPT-3.5-turbo for simple requests (<100 tokens), GPT-4 for complex, expected 50% route to cheaper model

4. RAG Optimization: Retrieve top-3 examples (vs top-5), compress examples (remove comments/whitespace), target 20% context reduction

5. Batch Processing: Batch 1,000 embeddings per API call, expected 10% total cost reduction

Caching / Batching Techniques:

Code Generation Cache:

- Redis cache with 30-day TTL
- Cache key: hash(user_description + language)
- If cached: return immediately (no LLM call)
- Expected 40% cache hit rate

Embedding Cache:

- Generate embeddings once during data processing
- Store permanently in Pinecone
- Update only for new code (incremental)

Embedding Generation Batching:

- Batch 1,000 code snippets per API call
- Reduces API calls from 2M to 2,000 (999 calls saved per batch)
- Airflow tasks coordinate batching

Cost Projection:

Without Optimization: 500 requests × $0.30 = $150/month

With Full Optimization:

- 40% cached (200 requests): $0
- 60% new (300 requests):
  - 50% GPT-3.5 (150 req): $0.05 each = $7.50
  - 50% GPT-4 (150 req): $0.15 each = $22.50
- Total: $30/month (80% cost reduction achieved)

11. Conclusion

This project creates a production-scale code generation platform combining multi-agent AI with big data infrastructure (50 GB from 6 sources, 2M+ code snippets). The system addresses developer productivity challenges through intelligent automation, achieving 85%+ accuracy in <30 seconds while maintaining quality through specialized agents and comprehensive testing. Expected impact includes 70-80% faster development, ROI within 2-3 developers, and demonstration of effective big data + LLM integration.

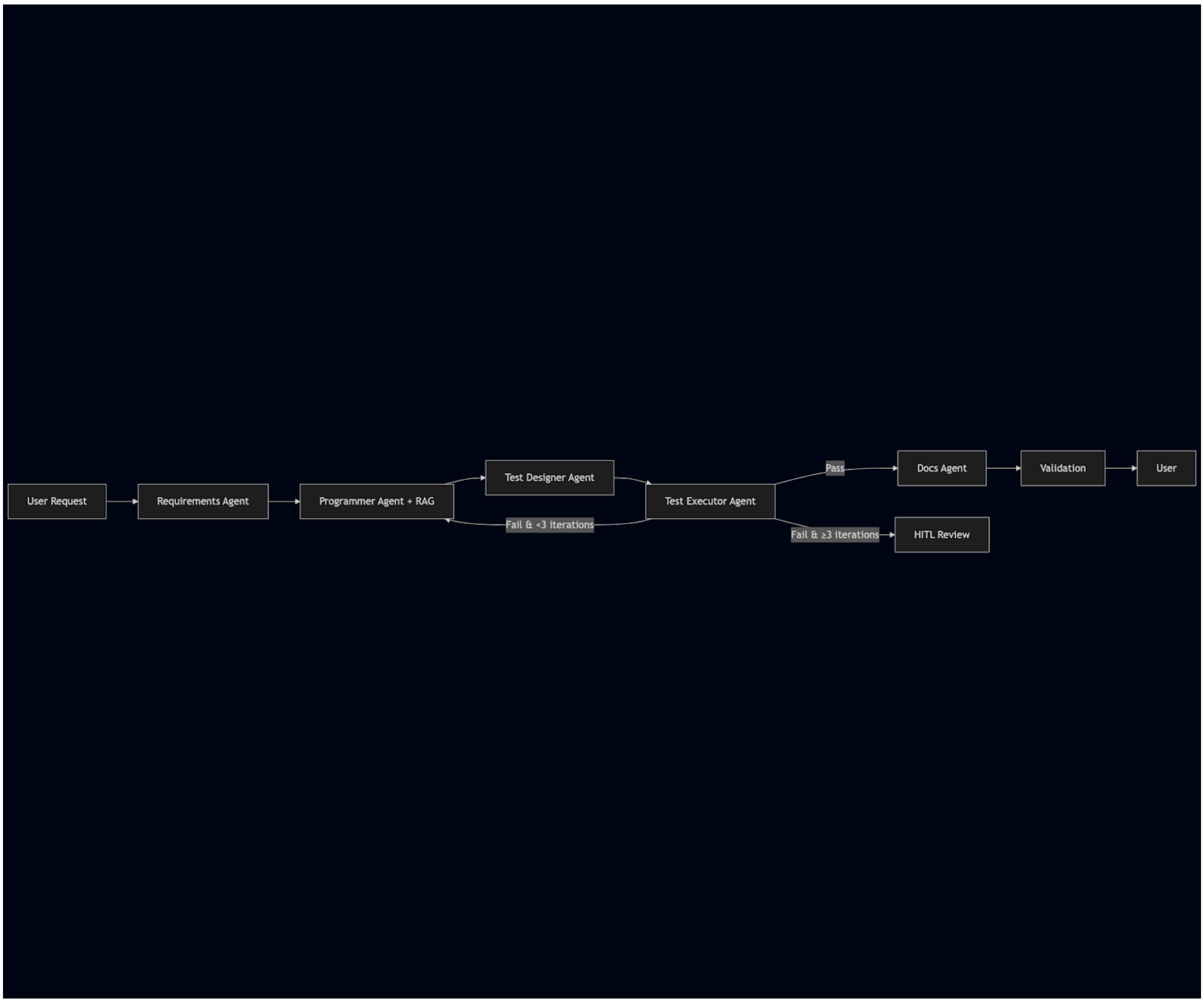
12. References

12. Huang, D., et al. (2023). "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing." arXiv:2312.13010
13. Chen, M., et al. (2021). "Evaluating Large Language Models Trained on Code." arXiv:2107.03374
14. Apache Airflow Documentation. https://airflow.apache.org/docs/
15. Google Cloud Composer Documentation. https://cloud.google.com/composer/docs
16. CrewAI Documentation. https://docs.crewai.com/
17. Pinecone Documentation. https://docs.pinecone.io/
18. Stack Exchange Data Dump. https://archive.org/details/stackexchange

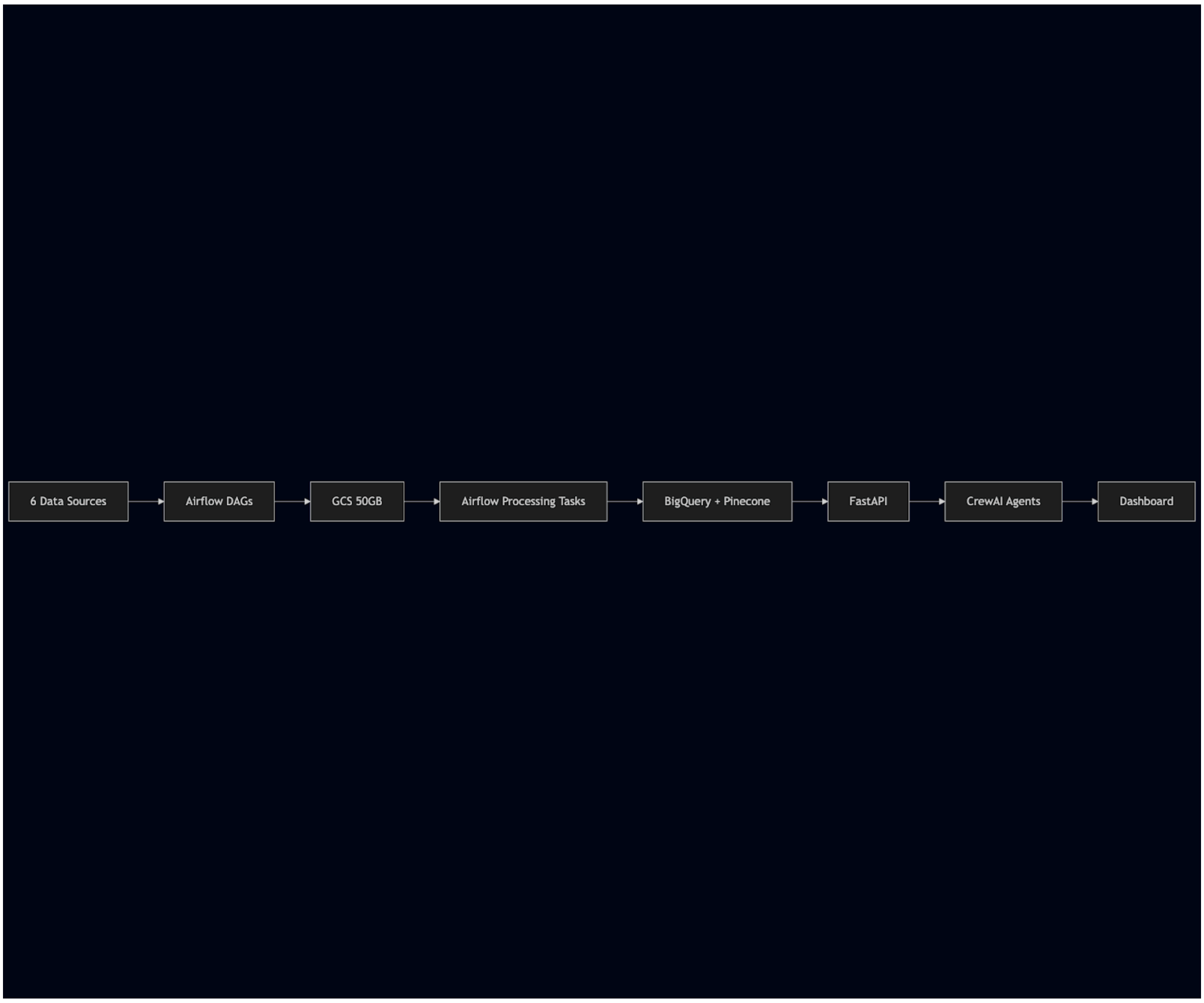
Appendix

A. **Mermaid Diagrams**

Multi-Agent Workflow Diagram:



Data Pipeline Diagram:



CrewAI Agents → Dashboard

B. Pseudocode

Programmer Agent Prompt:

You are an Expert Programmer. Generate production-ready code.

Requirements: {requirements_spec} Examples (RAG): {similar_code} Feedback: {test_feedback}

Generate code following best practices with error handling and type hints.

Test Designer Agent Prompt:

Create comprehensive tests based ONLY on requirements (not code). Cover: normal cases, edge cases, error cases. Output: pytest/Jest test suite.

C. JSON Schema

CodeGenerationRequest Schema: { "description": "string (10-2000 chars)", "language": "python | javascript", "complexity": "simple | medium | complex" }

CodeGenerationResponse Schema: { "code": "string", "tests": "string", "documentation": "string", "quality_score": "float (0-10)", "confidence": "float (0-1)" }

D. Pseudocode

Main Code Generation Flow:

FUNCTION generate_code(description): requirements = requirements_agent.analyze(description) examples = pinecone.search(requirements.embedding, top_k=3) code = programmer_agent.generate(requirements, examples) tests = test_designer_agent.generate(requirements)

```
FOR iteration IN 1 to 3:
    result = test_executor.execute(code, tests)
    IF result.passed: BREAK
    code = programmer_agent.refine(code, result.feedback)

IF NOT result.passed: RETURN to_hitl_queue()

docs = docs_agent.generate(code)
quality = validate(code)
IF quality < 7.0: RETURN to_hitl_queue()

RETURN {code, tests, docs, metadata}
```
