# AI Code Intelligence & Risk Analyzer

Comprehensive Project Documentation

Generated: February 05, 2026

Version 1.0.0

## Table of Contents

# 1. Executive Summary

The AI Code Intelligence & Risk Analyzer is a comprehensive code governance platform designed to help development teams understand, assess, and improve their codebase quality. By combining static code analysis with AI-powered insights, the platform provides actionable intelligence about security vulnerabilities, code maintainability, architectural patterns, and technical debt.

The platform addresses the growing need for automated code quality assessment in modern software development. With the increasing complexity of software systems and the pressure to deliver faster, teams need tools that can quickly identify potential issues and provide clear guidance on how to address them.

## Key Value Propositions:

- **Security First:** Identifies 20+ categories of security vulnerabilities with severity ratings and remediation guidance
- **Actionable Metrics:** Provides quantified scores (0-100) for security, maintainability, and architecture
- **AI-Powered Insights:** Leverages LLM technology to generate human-readable explanations and recommendations
- **Technical Debt Visibility:** Calculates and tracks technical debt with urgency classification
- **Professional Reporting:** Generates detailed PDF reports for stakeholder communication
- **User-Friendly Interface:** Modern React-based dashboard for easy interaction

## 2. Project Overview

### Project Information

| Attribute | Value |
|---|---|
| Project Name | AI Code Intelligence & Risk Analyzer |
| Version | 1.0.0 |
| License | MIT License |
| Primary Language | Python (Backend), TypeScript (Frontend) |
| Target Users | Developers, Tech Leads, Security Teams |
| Repository Type | Monorepo (Backend + Frontend) |

### Problem Statement

Modern software development faces several challenges that this platform addresses:

- Security vulnerabilities often go undetected until production deployment
- Technical debt accumulates silently, making future development harder
- Code quality metrics are often ignored due to lack of actionable insights
- Manual code reviews cannot scale with rapid development cycles
- Lack of standardized metrics makes it difficult to track improvement over time

### Solution Approach

The platform combines multiple analysis techniques to provide comprehensive insights:

- **Static Analysis:** Examines code without execution to find issues early
- **Pattern Matching:** Uses regex and AST analysis to detect anti-patterns
- **Metric Calculation:** Computes industry-standard metrics for objective assessment
- **AI Augmentation:** Uses LLM to translate technical findings into actionable recommendations
- **Visualization:** Presents data through intuitive dashboards and reports

# 3. Key Features & Capabilities

## 3.1 Security Analysis

The security module performs comprehensive vulnerability detection using multiple techniques:

| Feature | Description |
|---|---|
| Bandit Integration | Industry-standard Python security linter for vulnerability detection |
| Pattern-Based Detection | Custom regex patterns for common security anti-patterns |
| Severity Classification | Issues rated as CRITICAL, HIGH, MEDIUM, or LOW |
| CWE Mapping | Vulnerabilities mapped to Common Weakness Enumeration IDs |
| Remediation Guidance | Specific recommendations for fixing each issue |

## 3.2 Code Quality Analysis

The platform measures code quality through multiple dimensions:

| Metric | Description |
|---|---|
| Cyclomatic Complexity | Measures decision complexity with grades A-F |
| Cognitive Complexity | Measures how hard code is to understand |
| Maintainability Index | Composite score for code maintainability (0-100) |
| Lines of Code | LOC, SLOC, and logical LOC counts |
| Comment Ratio | Documentation coverage percentage |
| Halstead Metrics | Program length, difficulty, and effort calculations |

## 3.3 Architecture Analysis

| Analysis | Description |
|---|---|
| Dependency Graphs | Visualizes module dependencies and identifies circular references |
| Modularity Score | Measures coupling and cohesion between modules |

| | |
|---|---|
| Pattern Detection | Identifies architectural patterns (MVC, MVVM, layered) |
| Hub Analysis | Finds highly connected central modules |
| Layer Violations | Detects improper cross-layer dependencies |

# 4. Technology Stack

## 4.1 Backend Technologies

| Technology | Version | Purpose |
|---|---|---|
| FastAPI | 0.109.0 | Modern async web framework |
| SQLAlchemy | 2.0.25 | ORM for database operations |
| SQLite | - | Lightweight database storage |
| Radon | 6.0.1 | Code complexity analysis |
| Bandit | 1.7.7 | Security vulnerability scanning |
| NetworkX | 3.2.1 | Graph analysis for dependencies |
| Groq API | 0.4.2 | LLM integration for AI insights |
| ReportLab | 4.1.0 | PDF report generation |
| GitPython | 3.1.41 | Git repository operations |
| python-jose | - | JWT token handling |
| bcrypt | - | Password hashing |

## 4.2 Frontend Technologies

| Technology | Version | Purpose |
|---|---|---|
| React | 18.2.0 | UI component library |
| TypeScript | 5.2.2 | Type-safe JavaScript |
| Vite | 5.0.8 | Fast build tool and dev server |
| Tailwind CSS | 3.4.0 | Utility-first CSS framework |
| React Router | 6.21.0 | Client-side routing |
| Axios | 1.6.5 | HTTP client for API calls |
| Lucide React | 0.303.0 | Icon library |

# 5. System Architecture

## 5.1 High-Level Architecture

The system follows a modern three-tier architecture with clear separation of concerns:

**Presentation Layer (Frontend):** React-based single-page application that provides the user interface. Communicates with the backend via REST API calls. Handles authentication state, form validation, and result visualization.

**Application Layer (Backend):** FastAPI-based REST API server that handles business logic. Includes authentication, analysis orchestration, score calculation, LLM integration, and report generation.

**Data Layer:** SQLite database for persistent storage of user accounts and analysis results. Uses SQLAlchemy ORM for database operations.

## 5.2 Backend Module Structure

| Module | Responsibility |
|---|---|
| app/analyzers/ | Code analysis logic (security, complexity, architecture) |
| app/api/ | REST API endpoint definitions |
| app/auth/ | User authentication and authorization |
| app/core/ | Configuration and security utilities |
| app/db/ | Database models and session management |
| app/ingestion/ | GitHub repository cloning and loading |
| app/llm/ | Groq LLM integration and prompt building |
| app/reports/ | PDF report generation |
| app/scoring/ | Score calculation algorithms |
| app/utils/ | Shared utility functions |

## 6. Analysis Pipeline

The analysis pipeline processes repositories through a series of well-defined stages:

| Stage | Process | Output |
|---|---|---|
| 1. Authentication | Validate JWT token | User context |
| 2. Repository Ingestion | Clone GitHub repo via GitPython | Local repository files |
| 3. File Discovery | Scan for analyzable files | File list with metadata |
| 4. Structure Analysis | Parse AST for Python/JS files | Code structure data |
| 5. Complexity Analysis | Calculate via Radon | Complexity metrics |
| 6. Security Analysis | Run Bandit + pattern matching | Vulnerability list |
| 7. Maintainability Analysis | Compute quality metrics | Maintainability data |
| 8. Architecture Analysis | Build dependency graph | Architecture metrics |
| 9. Score Calculation | Apply weighted algorithms | Numeric scores |
| 10. AI Explanation | Generate via Groq LLM | Markdown summary |
| 11. Storage | Save to database | Analysis record |
| 12. Cleanup | Delete cloned repository | Clean temp directory |

### Pipeline Characteristics:

- **Deterministic Scoring:** All scores are calculated using rule-based algorithms, ensuring consistent and reproducible results
- **LLM for Explanation Only:** AI is used solely for generating human-readable explanations, never for decision-making
- **Graceful Degradation:** System works without LLM, using fallback explanations if API is unavailable
- **Automatic Cleanup:** Temporary files are always cleaned up, even on failure

# 7. Security Analysis Module

## 7.1 Vulnerability Categories

The security analyzer detects the following categories of vulnerabilities:

| Category | Severity | Example Pattern |
|---|---|---|
| Hardcoded Credentials | CRITICAL | credentials in source code |
| SQL Injection | CRITICAL | string concatenation in queries |
| Command Injection | CRITICAL | unsanitized shell commands |
| Unsafe Deserialization | HIGH | untrusted data deserialization |
| Dynamic Code Execution | HIGH | dynamic code execution |
| Weak Cryptography | MEDIUM | outdated hash algorithms |
| Debug Mode | MEDIUM | debug enabled in production |
| XSS Vulnerabilities | HIGH | unsanitized HTML output |
| Path Traversal | HIGH | unvalidated file paths |
| Insecure Random | MEDIUM | weak random for security |

## 7.2 Security Score Calculation

The security score (0-100) is calculated based on vulnerability severity and count:

```
Security Score = 100 - (CRITICAL × 25) - (HIGH × 15) - (MEDIUM × 8) - (LOW × 3)
```

The score is capped at 0 (minimum) and additional factors like vulnerability density (issues per KLOC) may further adjust the score.

# 8. Code Quality Metrics

## 8.1 Cyclomatic Complexity

Cyclomatic complexity measures the number of independent paths through code. It's calculated by analyzing decision points (if, for, while, etc.).

| Grade | Complexity Range | Risk Level |
|---|---|---|
| A | 1-5 | Low - simple, well-structured |
| B | 6-10 | Low - reasonable complexity |
| C | 11-20 | Moderate - more complex |
| D | 21-30 | High - difficult to test |
| E | 31-40 | Very High - error prone |
| F | 41+ | Critical - untestable |

## 8.2 Maintainability Index

The Maintainability Index (MI) is a composite metric that considers volume, complexity, and lines of code:

```
MI = 171 - 5.2 × ln(V) - 0.23 × G - 16.2 × ln(LOC) Where: V = Halstead Volume G =
Cyclomatic Complexity LOC = Lines of Code
```

| MI Range | Interpretation |
|---|---|
| 85-100 | Highly maintainable |
| 65-84 | Moderately maintainable |
| 0-64 | Difficult to maintain |

# 9. Technical Debt Calculation

## 9.1 Debt Index Formula

Technical debt is calculated as a weighted combination of multiple factors:

```
Tech Debt Index = (0.35 × Security Debt) + (0.30 × Maintainability Debt) + (0.25 ×
Architecture Debt) + (0.10 × Code Smell Debt) Where each debt component = 100 - respective
score
```

## 9.2 Refactoring Urgency Levels

| Level | Debt Range | Recommended Action |
|---|---|---|
| LOW | 0-25 | Continue normal development, address issues opportunistically |
| MEDIUM | 26-50 | Plan dedicated refactoring time in upcoming sprints |
| HIGH | 51-75 | Prioritize debt reduction before new features |
| CRITICAL | 76-100 | Immediate action required, consider feature freeze |

# 10. AI Integration

## 10.1 Groq LLM Integration

The platform integrates with Groq's fast LLM inference API to generate human-readable analysis explanations. Key characteristics:

- **Model:** Uses llama3-8b-8192 by default (configurable)
- **Purpose:** Explanation generation only - never used for scoring decisions
- **Temperature:** Low (0.2) for consistent, focused outputs
- **Max Tokens:** 600 tokens for concise summaries
- **Fallback:** Template-based explanations when LLM unavailable

## 10.2 Prompt Engineering

The LLM receives structured prompts containing:

- Repository name and analysis context
- Calculated scores (security, maintainability, architecture)
- Top security vulnerabilities with severity
- Complexity hotspots and code quality issues
- Technical debt index and urgency level
- Instructions to provide actionable recommendations

# 11. API Reference

## 11.1 Authentication Endpoints

| Method | Endpoint | Description |
|---|---|---|
| POST | /api/auth/signup | Register new user account |
| POST | /api/auth/login | Authenticate and get JWT token |
| POST | /api/auth/verify | Verify JWT token validity |
| GET | /api/auth/me | Get current user information |

## 11.2 Analysis Endpoints

| Method | Endpoint | Description |
|---|---|---|
| POST | /api/analyze/ | Analyze a GitHub repository |
| GET | /api/analyze/ | List user's analysis history |
| GET | /api/analyze/{id} | Get specific analysis details |
| DELETE | /api/analyze/{id} | Delete an analysis |

## 11.3 Report Endpoints

| Method | Endpoint | Description |
|---|---|---|
| GET | /api/reports/{id}/pdf | Download PDF report |
| GET | /api/reports/{id}/summary | Get analysis summary |

## 12. Database Schema

### 12.1 Users Table

| Column | Type | Description |
|---|---|---|
| id | INTEGER (PK) | Auto-incrementing primary key |
| email | VARCHAR (unique) | User email address |
| username | VARCHAR (unique) | Username for display |
| hashed_password | VARCHAR | bcrypt hashed password |
| is_active | BOOLEAN | Account active status |
| is_admin | BOOLEAN | Admin privileges flag |
| created_at | DATETIME | Account creation timestamp |
| updated_at | DATETIME | Last update timestamp |

### 12.2 AnalysisReports Table

| Column | Type | Description |
|---|---|---|
| id | INTEGER (PK) | Auto-incrementing primary key |
| user_id | INTEGER (FK) | Reference to users table |
| repo_url | VARCHAR | GitHub repository URL |
| repo_name | VARCHAR | Repository name |
| branch | VARCHAR | Analyzed branch |
| metrics | JSON | Detailed analysis metrics |
| security_score | FLOAT | Security score (0-100) |
| maintainability_score | FLOAT | Maintainability score (0-100) |
| architecture_score | FLOAT | Architecture score (0-100) |
| tech_debt_index | FLOAT | Technical debt index (0-100) |

| | | |
|---|---|---|
| refactor_urgency | VARCHAR | LOW/MEDIUM/HIGH/CRITICAL |
| llm_explanation | TEXT | AI-generated explanation |
| files_analyzed | INTEGER | Number of files analyzed |
| total_lines | INTEGER | Total lines of code |
| analysis_duration | FLOAT | Time taken in seconds |
| created_at | DATETIME | Analysis timestamp |

# 13. Installation Guide

## 13.1 Prerequisites

- Python 3.9 or higher
- Node.js 16 or higher with npm
- Git (for repository cloning functionality)

## 13.2 Backend Installation

```
# Clone repository git clone <repository-url> cd "AI Code Intelligence & Risk Analyzer" #
Navigate to backend cd backend # Create virtual environment python -m venv venv # Activate
virtual environment # Windows: venv\Scripts\activate # Linux/Mac: source
venv/bin/activate # Install dependencies pip install -r requirements.txt # Create
environment file cp .env.example .env # Edit .env with your configuration # Run server
python main.py
```

## 13.3 Frontend Installation

```
# Navigate to frontend cd frontend # Install dependencies npm install # Run development
server npm run dev
```

# 14. Configuration

## 14.1 Environment Variables

| Variable | Required | Default | Description |
|---|---|---|---|
| JWT_SECRET_KEY | Yes | - | Secret for JWT token generation |
| GROQ_API_KEY | No | - | Groq API key for AI explanations |
| GROQ_MODEL | No | llama3-8b-8192 | LLM model to use |
| GROQ_MAX_TOKENS | No | 600 | Max tokens for response |
| GROQ_TEMPERATURE | No | 0.2 | LLM temperature |
| DEBUG | No | false | Enable debug mode |

## 14.2 Getting Groq API Key

1. Visit https://console.groq.com
2. Create a free account
3. Navigate to API Keys section
4. Generate a new API key
5. Add key to .env file as GROQ_API_KEY

# 15. Usage Guide

## 15.1 Getting Started

1. **Create Account:** Navigate to the signup page and create a new account with email, username, and password
2. **Login:** Use your credentials to log in and receive a JWT token
3. **Submit Repository:** Enter a public GitHub repository URL (e.g., https://github.com/owner/repo) and optionally specify a branch
4. **Wait for Analysis:** The system will clone the repository, analyze the code, and calculate scores
5. **View Results:** Review security, maintainability, and architecture scores on the dashboard
6. **Read AI Insights:** Check the AI-generated explanation for actionable recommendations
7. **Download Report:** Generate a PDF report for documentation or stakeholder communication

## 15.2 Interpreting Results

**Security Score:** Higher is better. Focus on CRITICAL and HIGH severity issues first. A score below 60 indicates significant security concerns that should be addressed before deployment.

**Maintainability Score:** Higher is better. Scores below 65 suggest the codebase may be difficult to maintain. Focus on reducing complexity in flagged modules.

**Architecture Score:** Higher is better. Low scores may indicate circular dependencies, poor modularity, or architectural anti-patterns. Consider refactoring highly-coupled modules.

**Technical Debt:** Lower is better (it's debt!). Use the refactoring urgency level to prioritize remediation efforts.

## 16. Future Enhancements

The following enhancements are planned or under consideration for future releases:

- **Language Support Expansion:** Add support for Java, Go, Rust, and other languages
- **CI/CD Integration:** GitHub Actions, GitLab CI, and Jenkins plugins
- **Trend Analysis:** Track code quality metrics over time across multiple analyses
- **Team Features:** Organization accounts, shared dashboards, and role-based access
- **Custom Rules:** Allow users to define custom security patterns and thresholds
- **Private Repository Support:** Authentication for private GitHub repositories
- **Real-time Monitoring:** Webhook-based analysis on push events
- **IDE Extensions:** VS Code and JetBrains extensions for in-editor analysis
- **Automated Fix Suggestions:** AI-powered code fix recommendations
- **Docker Support:** Containerized deployment option

## Document Information

| Attribute | Value |
|---|---|
| Document Title | AI Code Intelligence & Risk Analyzer - Documentation |
| Version | 1.0.0 |
| Generated Date | February 05, 2026 at 13:33 |
| Total Pages | 16+ |
| Author | Auto-generated Documentation |