Code Quality Analyzer - System Architecture

Executive Summary

The Code Quality Analyzer is built on a **pipeline-based architecture** with **parallel processing capabilities** and **AI-driven analysis**. The system follows a **multi-stage orchestration pattern** that transforms raw code input into actionable insights through structured analysis phases.

Architecture Philosophy

Core Principles

1. **Separation of Concerns**: Each component has a single, well-defined responsibility
2. **Scalable Processing**: Parallel execution for independent analysis tasks
3. **Extensible Design**: Easy to add new analysis types or supported languages
4. **Resilient Operations**: Graceful error handling and fallback mechanisms
5. **Resource Efficiency**: Optimized memory usage and API call management

Design Patterns Used

- **Orchestrator Pattern**: `CodeAnalysisOrchestrator` coordinates the entire pipeline
- **Factory Pattern**: Query engine creation for different file types
- **Strategy Pattern**: Different analysis strategies for various code issues
- **Observer Pattern**: Status updates throughout the analysis pipeline
- **Template Method**: Consistent analysis workflow across different modules

System Architecture Overview

```
┌─────────────────┐    ┌───────────────────┐    ┌─────────────────┐
│   Streamlit UI  │    │   File Handler    │    │  Temp Storage   │
│                 │────│                   │────│                 │
│  - Upload       │    │  - Validation     │    │  - Structure    │
│  - Display      │    │  - Extraction     │    │  - Cleanup      │
│  - Download     │    │  - Processing     │    │  - Persistence  │
└─────────────────┘    └───────────────────┘    └─────────────────┘
         │                       │                        │
         │                       │                        │
         v                       v                        v

┌ ──────────────────────────────────────────────────────────── ──┐
─ ──                                                              │
│                    ORCHESTRATION LAYER                          │
│                                                                 │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │            CodeAnalysisOrchestrator                      │  │
│  │                                                          │  │
│  │ • Pipeline Management                                    │  │
│  │ • Resource Coordination                                  │  │
│  │ • Error Handling                                         │  │
│  │ • Report Synthesis                                       │  │
│  └───────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────
─┘
```

```
                     │
                     │
                     v
┌──────────────────────────────────────────────────────────────────┐
│                   PROCESSING LAYER                              │
│                                                                 │
│ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐              │
│ │    Structure │ │     Query    │ │    Analysis │               │
│ │    Builder   │ │    Engine    │ │    Modules   │              │
│ │              │ │    Factory   │ │              │              │
│ │  • Parse     │ │ • Create     │ │ • Imports    │              │
│ │  • Map       │ │ • Index      │ │ • Issues     │              │
│ │  • Store     │ │ • Embed      │ │ • Duplicates│               │
│ └──────────────┘ └──────────────┘ └──────────────┘              │
└──────────────────────────────────────────────────────────────────┘
           │                 │                 │
           │                 │                 │
           v                 v                 v
┌──────────────────────────────────────────────────────────────────┐
│                    AI/LLM LAYER                                 │
│                                                                 │
│ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐              │
│ │    Gemini    │ │    OpenAI    │ │    LlamaIndex│              │
│ │    Models    │ │ Embeddings │ │      Framework │              │
│ │              │ │              │ │              │              │
│ │ • Analysis │ │ • Vector       │ │ • Query      │              │
│ │ • Summary    │ │ • Search     │ │ • Context    │              │
│ │ • Q&A        │ │ • Semantic │ │ • Retrieval     │             │
│ └──────────────┘ └──────────────┘ └──────────────┘              │
└──────────────────────────────────────────────────────────────────┘
```

Detailed Component Architecture

1. Input Processing Layer

File Handler

- **Purpose:** Manages file upload, validation, and temporary storage
- **Components:**
  - `upload_section()` : Handles UI interactions
  - `process_files()` : Individual file processing
  - `process_zip()` : Archive extraction and processing
- **Design Decision:** Temporary directory approach allows handling of large codebases without memory constraints

Structure Builder (`CodebaseStructurer`)

- **Purpose:** Creates hierarchical representation of codebase
- **Algorithm:** Recursive directory traversal with metadata extraction
- **Output:** JSON structure preserving file hierarchy
- **Design Decision:** Separation of content from structure enables efficient querying

2. Orchestration Layer

CodeAnalysisOrchestrator

- **Purpose:** Central coordinator for the entire analysis pipeline
- **Responsibilities:**
  - Resource lifecycle management
  - Pipeline stage coordination
  - Error handling and recovery
  - Report synthesis

3. Analysis Engine Layer

Query Engine Factory

- **Framework:** LlamaIndex for document processing
- **Components:**
  - Document indexing
  - Vector embeddings
  - Query processing
  - Context retrieval

Analysis Modules

Import Analyzer

Process Flow:

- Extract **import** statements **from** each file
- Categorize by type (standard, third-party, local)
- Parallel analysis across files
- Cross-reference dependencies
- Generate comprehensive report

Code Issues Analyzer

Process Flow:

- Security vulnerability scanning
- Performance bottleneck detection
- Code quality assessment
- Logic error identification
- Aggregated reporting

Duplication Analyzer

Process Flow:

- Combine all code files
- Create unified query engine
- Pattern recognition analysis

```
•  Similarity detection
•  Refactoring recommendations
```

4. AI/LLM Integration Layer

Model Selection Strategy

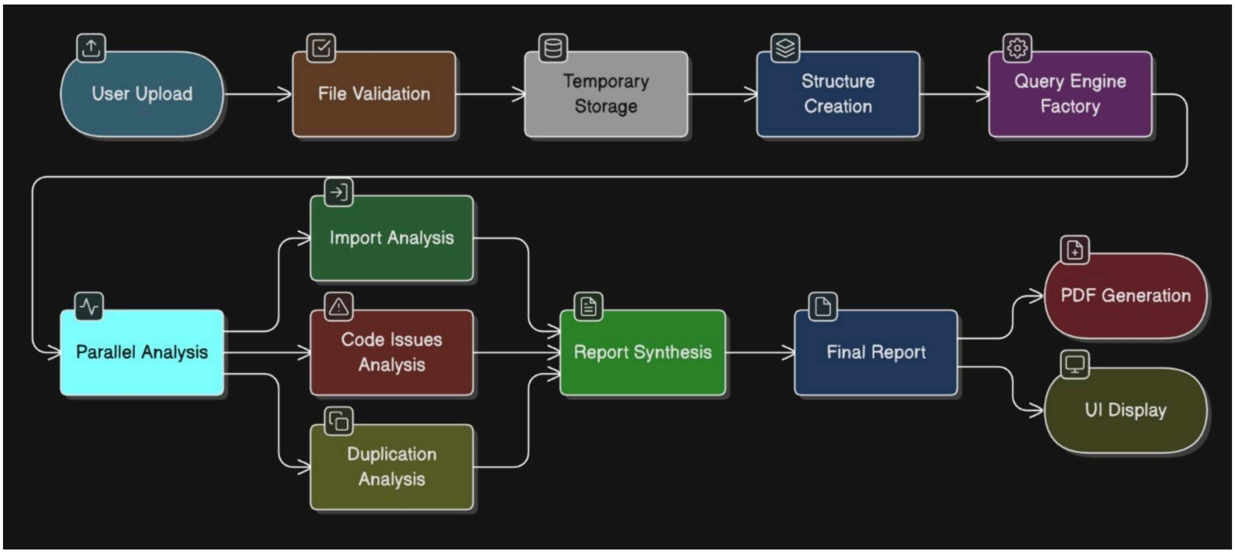
- **Primary LLM:** Google Gemini 2.0 Flash
  - **Rationale:** Optimized for code analysis with large context windows
  - **Usage:** Analysis prompts, report generation, Q&A
- **Embedding Model:** OpenAI text-embedding-3-small
  - **Rationale:** High-quality semantic representations
  - **Usage:** Vector search, document retrieval

Prompt Engineering Architecture

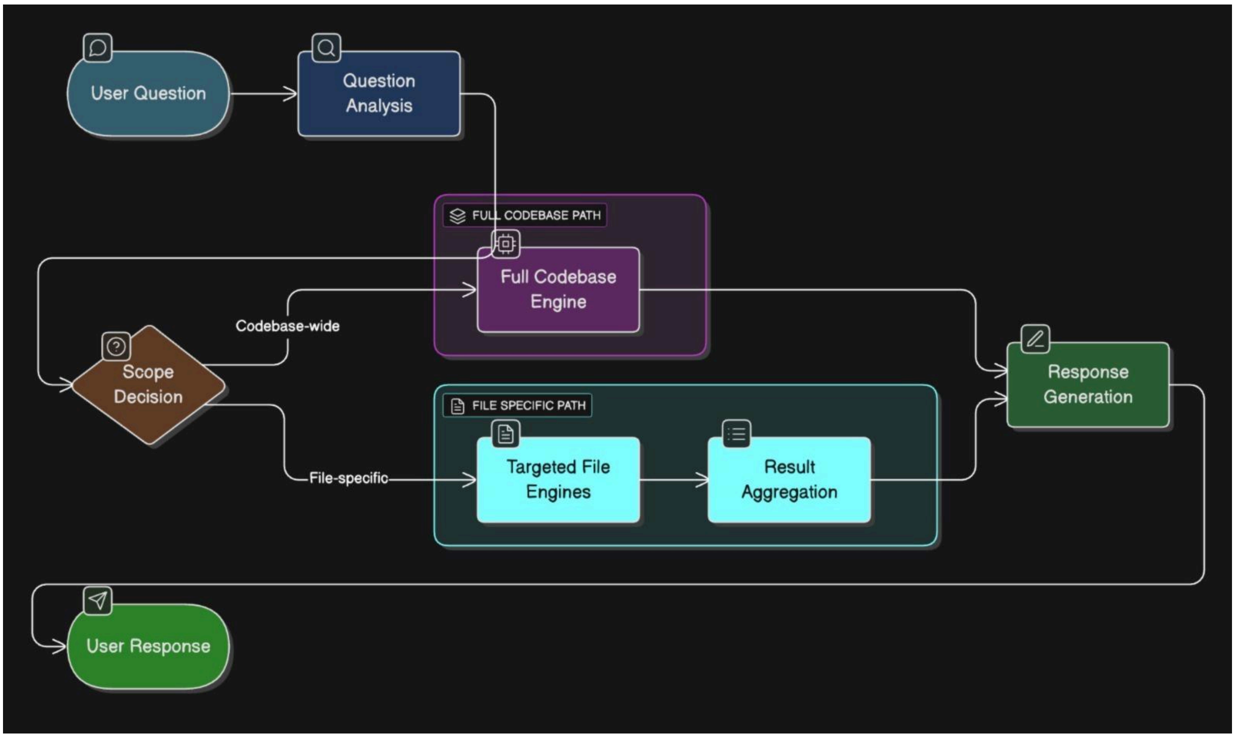
```
AnalysisPrompts Class:
├── Import Analysis Prompts
├── Code Issues Prompts
├── Duplication Analysis Prompts
└── Summary Generation Prompts

ResponseCleaner Class:
├── JSON Response Parsing
├── Markdown Cleaning
└── Error Recovery
```

Data Flow Architecture

Primary Analysis Flow



Q&A Flow



Scalability Architecture

Parallel Processing Strategy

1. **File-Level Parallelism:** Independent analysis of each code file
2. **Analysis-Type Parallelism:** Concurrent execution of different analysis types
3. **Thread Pool Management:** Configurable concurrency limits

Memory Management

1. **Streaming Processing:** Files processed individually to limit memory usage
2. **Temporary Storage:** Disk-based storage for large codebases
3. **Cleanup Mechanisms:** Automatic resource deallocation

API Rate Limiting

1. **Request Batching:** Minimize API calls through intelligent batching
2. **Concurrent Limits:** Configurable thread pools prevent API throttling
3. **Error Recovery:** Exponential backoff for failed requests

Security Architecture

Input Validation

- File type verification
- Content sanitization
- Size limit enforcement
- Malicious content detection

API Security

- Environment variable management
- Secure API key handling
- Request/response sanitization

Data Privacy

- Temporary storage with automatic cleanup
- No persistent storage of user code
- Secure processing pipeline

Extension Points

Adding New Analysis Types

```
class NewAnalysisModule:
    def analyze(self, query_engines):
        # Custom analysis logic
        pass

    def generate_report(self, results):
        # Report generation
        pass
```

Adding New File Types

```
SUPPORTED_EXTENSIONS = {
    '.py', '.js', '.tsx',
    '.new_extension' # Add here
}
```

Custom LLM Integration

```
class CustomLLMProvider:
    def _init_(self, api_key):
        self.configure_model()

    def generate_response(self, prompt):
        # Custom LLM logic
        pass
```

Performance Characteristics

Analysis Complexity

- **Time Complexity:** O(n × m) where n = files, m = average file size

- **Space Complexity:** O(k) where k = concurrent processing limit
- **API Calls:** Linear with file count plus constant summary calls

Bottleneck Analysis

1. **Primary Bottleneck:** LLM API response times
2. **Secondary Bottleneck:** Embedding generation for large files
3. **Mitigation:** Parallel processing and request batching

Error Handling Strategy

Graceful Degradation

- Individual file failures don't stop pipeline
- Partial results returned when possible
- Clear error reporting for user action

Recovery Mechanisms

- Automatic retry with exponential backoff
- Alternative analysis paths for failures
- Comprehensive logging for debugging

Future Architecture Considerations

Planned Enhancements

1. **GitHub Integration:** Direct repository analysis
2. **Real-time Analysis:** Incremental processing for code changes
3. **Custom Rule Engine:** User-defined analysis rules
4. **Multi-repository Analysis:** Cross-project dependency analysis

Scalability Improvements

1. **Distributed Processing:** Multi-node analysis capability
2. **Caching Layer:** Results caching for repeated analyses
3. **Background Processing:** Asynchronous analysis queue
4. **Database Integration:** Persistent analysis history

This architecture provides a solid foundation for comprehensive code analysis while maintaining flexibility for future enhancements and scalability requirements.