Code-to-Doc: Legacy Code Documentation Agent

Complete Implementation Guide

Phase 1: Foundation Setup

Step 1: Repository Cloning & Initial Setup

Objective: Safely clone and prepare the repository for analysis

Implementation:

```
- Accept GitHub URL from user
- Validate URL format
- Clone repository to local temp directory
- Extract repo metadata (name, description, primary language)
- Calculate repo size and file count
- Create working directory structure
```

Key Considerations:

- Use gitpython library for safe cloning
- Handle private repos with GitHub tokens
- Implement size limits to prevent memory issues
- Create backup before any modifications

Step 2: Code Scanning & File Analysis

Objective: Identify and categorize all code files

Implementation:

```
- Traverse directory tree
- Detect programming language for each file
- Filter out:
  * Binary files
  * Node modules, venv, .git directories
  * Test files (optional, can document separately)
  * Build artifacts
- Create inventory of code files
- Calculate complexity metrics
```

Tools:

- `pathlib` for file operations
- `chardet` for encoding detection
- `python-magic` for file type detection
- File extension mapping dictionary

Phase 2: Code Understanding

Step 3: Dependency Graph Construction

Objective: Map how modules depend on each other

Implementation:

```
For Python:
- Parse import statements (import X, from X import Y)
- Use AST module to extract dependency info
- Handle relative imports
- Resolve circular dependencies
- Create dependency adjacency matrix

For Java:
- Parse import statements
- Extract class hierarchies
- Map package dependencies
- Identify public interfaces
```

Output Structure:

```
dependency_graph = {
  'module_a.py': {
    'imports': ['module_b', 'module_c'],
    'imported_by': ['module_d'],
    'external_deps': ['requests', 'numpy']
  }
}
```

Step 4: Architecture Understanding

Objective: Understand high-level system design

Implementation:

```
- Identify core modules (entry points, main logic)
- Detect design patterns:
  * Factory patterns
  * Singleton patterns
  * Observer patterns
- Analyze data flow between modules
- Identify configuration files
- Find database schemas or models
- Map API endpoints (if applicable)
```

Prompt for LLM: "Analyze this code structure and identify: 1) Core architectural components 2) Main design patterns 3) Data flow 4) Dependencies"

Phase 3: Agent Setup & Configuration

Step 5: LangChain Agent Configuration

Objective: Set up AI agent with appropriate tools

Implementation:

```
python
```

```
# Initialize components
- LLM: Claude (Sonnet 4) or CodeLlama via API
- Memory: Conversation buffer for context
- Tools:
  * read_file(path) - Read code file
  * analyze_function(code) - Analyze single function
  * search_codebase(query) - Search for patterns
  * get_dependency_info(file) - Retrieve dependency data
  * generate_docstring(function_code) - Create docstring

# Set up agent configuration
- Max iterations: 10-15
- Temperature: 0.3 (low for consistency)
- Context window: Use 90% of available tokens
- Timeout: 5 minutes per file
```

Agent Personality Prompt: "You are a Senior Software Engineer reviewing legacy code. Generate clear, concise documentation that a junior developer could understand. Include: purpose, parameters, return values, and examples where helpful."

Phase 4: Core Documentation Generation

Step 6: Function-Level Analysis & Docstring Generation

Objective: Generate docstrings for all functions

Implementation:

```
For each code file:
1. Parse AST to extract all functions/methods
2. For each function:
   - Extract signature
   - Analyze code logic (up to 4KB context)
   - Call LLM agent to generate docstring
   - Format according to language standards:
     * Python: Google/NumPy style
     * Java: JavaDoc format
3. Inject docstring into AST
4. Regenerate code with docstrings
5. Validate syntax
```

Example Docstring Prompt:

```
Analyze this Python function and generate a Google-style docstring.
Include: brief description, Args, Returns, Raises (if applicable)

Function:
[function code here]

Generate ONLY the docstring, properly formatted.
```

Output Format (Python):

```
python

def function_name(param1, param2):
    """
    Brief description of what function does.

    More detailed explanation if needed. Explain the algorithm
    or approach if it's non-obvious.

    Args:
        param1 (str): Description of param1
        param2 (int): Description of param2

    Returns:
        bool: Description of return value

    Raises:
        ValueError: When input validation fails
        KeyError: When required key is missing

    Example:
        >>> result = function_name('test', 42)
        >>> print(result)
        True
    """
```

Step 7: README Generation

Objective: Create comprehensive architecture documentation

Implementation:

```
README structure:
1. Project Overview
  - Purpose and context
  - Why this code exists

2. Architecture
  - High-level system design (ASCII diagram)
  - Core components and their responsibilities
  - Design patterns used

3. Dependencies
  - External library list
  - Version requirements
  - Internal module relationships (dependency graph)

4. Getting Started
  - Prerequisites
  - Installation steps
  - Basic usage example

5. Module Guide
  - Detailed description of each core module
  - Key classes and functions
  - Data structures used

6. API Reference (if applicable)
  - Endpoint descriptions
  - Request/response formats

7. Known Limitations
  - Technical debt identified
  - Performance considerations

8. Development Notes
  - Testing approach
  - Debugging tips
```

Prompt for LLM:

```
Generate a professional README.md for this legacy codebase.
Architecture overview:
[dependency graph]

Code statistics:
[file counts, complexity metrics]

Create sections for: Overview, Architecture, Dependencies, Getting Started, and Module Guide.
```

Phase 5: Code Modification & Integration

Step 8: Docstring Injection

Objective: Safely inject docstrings into original files

Implementation:

```
For Python (using AST):
1. Parse file with ast.parse()
2. Use ast.NodeVisitor to traverse
3. For each FunctionDef/ClassDef node:
   - Check if docstring exists
   - Insert generated docstring
   - Maintain indentation
   - Preserve comments
4. Use astor or unparse() to regenerate code
5. Write back to file
6. Validate with ast.parse()

For Java (using regex/AST):
1. Parse with JavaParser library
2. Navigate to method declarations
3. Insert JavaDoc comments above method signature
4. Format according to JavaDoc standards
5. Write back to file
6. Compile check with javac
```

Safety Measures:

- Create backup of original file
- Use atomic writes (write to temp file, then rename)
- Track all changes for diff report

- Validate syntax after modification
- Handle edge cases: multi-line signatures, decorators, etc.

Step 9: Quality Assurance

Objective: Ensure documentation quality and correctness

Implementation:

```
Validation checks:
1. Syntax Validation
  - Python: ast.parse() on modified files
  - Java: javac compilation check

2. Docstring Quality Checks
  - Minimum length check (avoid empty descriptions)
  - Parameter documentation completeness
  - Check for placeholder text
  - Verify proper formatting

3. Consistency Checks
  - Parameter names match function signature
  - Return type mentioned if function returns
  - Raised exceptions documented

4. LLM Re-review (optional)
  - Prompt LLM to rate docstring quality
  - Flag low-quality docstrings for regeneration

5. Conflict Resolution
  - Handle existing docstrings (append, replace, or skip)
  - Resolve ambiguous parameters
```

Phase 6: Delivery

Step 10: Output Generation

Objective: Package deliverables for user

Implementation:

```
Generate outputs:
1. Modified source files with docstrings
2. README.md with architecture documentation
3. DOCUMENTATION.md with detailed module guide
4. dependency_graph.json (machine-readable)
5. changes.diff - unified diff of all changes
6. documentation_report.json:
   {
     "total_files": 45,
     "functions_documented": 234,
     "coverage": "98%",
     "quality_score": 8.5,
     "processing_time": "5m 23s"
   }
7. Index of all documented functions (searchable)

Directory structure:
```

```
output/
├── README.md
├── DOCUMENTATION.md
├── src/  (modified source files)
├── docs/
│   ├── dependency_graph.json
│   ├── architecture_diagram.md
│   └── module_index.html
├── reports/
│   ├── changes.diff
│   └── documentation_report.json
```

```


---


### Step 11: Git Integration (Optional)
**Objective:** Automate PR creation for documentation

**Implementation:**
```

1. Create feature branch:
   - Branch name: docs/auto-documentation-{timestamp}
2. Stage changes:
   - git add -A

3. Create commit:

    - Message: "docs: auto-generate docstrings and README"

    - Include documentation report in commit body

4. Push to remote:

    - git push origin feature-branch

5. Create Pull Request:

    - Title: "Documentation: Auto-generated docs for legacy code"

    - Body: Include statistics and review checklist

    - Request review from code owners

6. Handle responses:

    - Track merge status

    - Update on additional feedback

```

---

## Implementation Order & Priorities

### Priority 1 (MVP): Steps 1-3, 6
- Clone repo
- Scan files
- Map dependencies
- Generate docstrings

### Priority 2 (Core): Steps 4-5, 7
- Understand architecture
- Set up agent
- Generate README

### Priority 3 (Polish): Steps 8-11
- Inject docstrings
- QA checks
- Package outputs
- Git integration

---

## Technology Stack Recommendations

### Core Libraries
```

langchain==0.1.0+
llama-index==0.9.0+
gitpython==3.1.0+
anthropic==0.7.0+ # or use Claude API

```
### Language Parsing
```

ast (Python, built-in)
javaparser (Java)
tree-sitter (Multiple languages)

```
### Code Analysis
```

radon (Python complexity)
lizard (Multi-language metrics)
networkx (Dependency graphs)

```
### Additional Tools
```

pygments (Syntax highlighting)
astor (Python AST to code)
python-dotenv (Configuration)
rich (Terminal output)

```
---

## Key Error Handling Patterns
```

- Timeout handling for large files
- Memory management for deep recursion
- API rate limiting with retry logic
- Graceful degradation for unsupported languages
- Rollback on critical failures
- Detailed logging for debugging

```
---

## Performance Optimization

- Process files in parallel (ThreadPoolExecutor for I/O)
- Cache dependency graphs and AST parses
- Batch LLM calls when possible (process multiple functions per request)
- Use streaming for large files
- Implement incremental processing (skip documented files)
- Monitor token usage and implement cost controls

---

## Testing Strategy
```

1. Unit tests: Individual components (file parser, AST handler)
2. Integration tests: End-to-end documentation for sample repos
3. Quality tests: Docstring content validation
4. Performance tests: Handle large codebases
5. Safety tests: No file corruption, reversible changes

```
---

## Next Steps After Implementation

1. Support multiple languages (Go, Rust, JavaScript)
2. Add configuration file for customization
3. Web UI for easier usage
4. Integration with popular code hosting platforms
5. Custom docstring style templates
6. Team collaboration features (comments, suggestions)
7. Continuous documentation updates on code changes
```