SMARTqA

Autonomous QA Agent

Intelligent Test Case & Selenium Script Generation

from Documentation

Technical Documentation

Ayush Pandey

@AyushPandey003

November 26, 2025

Contents

1 Executive Summary

Project Overview

SMARTqA is an intelligent, autonomous QA agent that revolutionizes test automation by constructing a "testing brain" from project documentation. It leverages Retrieval-Augmented Generation (RAG) to generate comprehensive test cases and executable Selenium scripts with zero hallucinations.

1.1 Key Capabilities

- **Documentation-Grounded Testing**: Zero hallucination test plans based on actual project specs
- **Multi-Format Support**: Ingests PDF, Markdown, TXT, JSON, and HTML documents
- **Intelligent Code Generation**: Production-ready Selenium scripts with explicit waits and error handling
- **Semantic Search**: FAISS-powered vector database for context retrieval
- **User-Friendly Interface**: Streamlit-based dashboard with real-time feedback

2 System Architecture

2.1 High-Level Architecture

The SMARTqA system follows a layered architecture with four primary components:

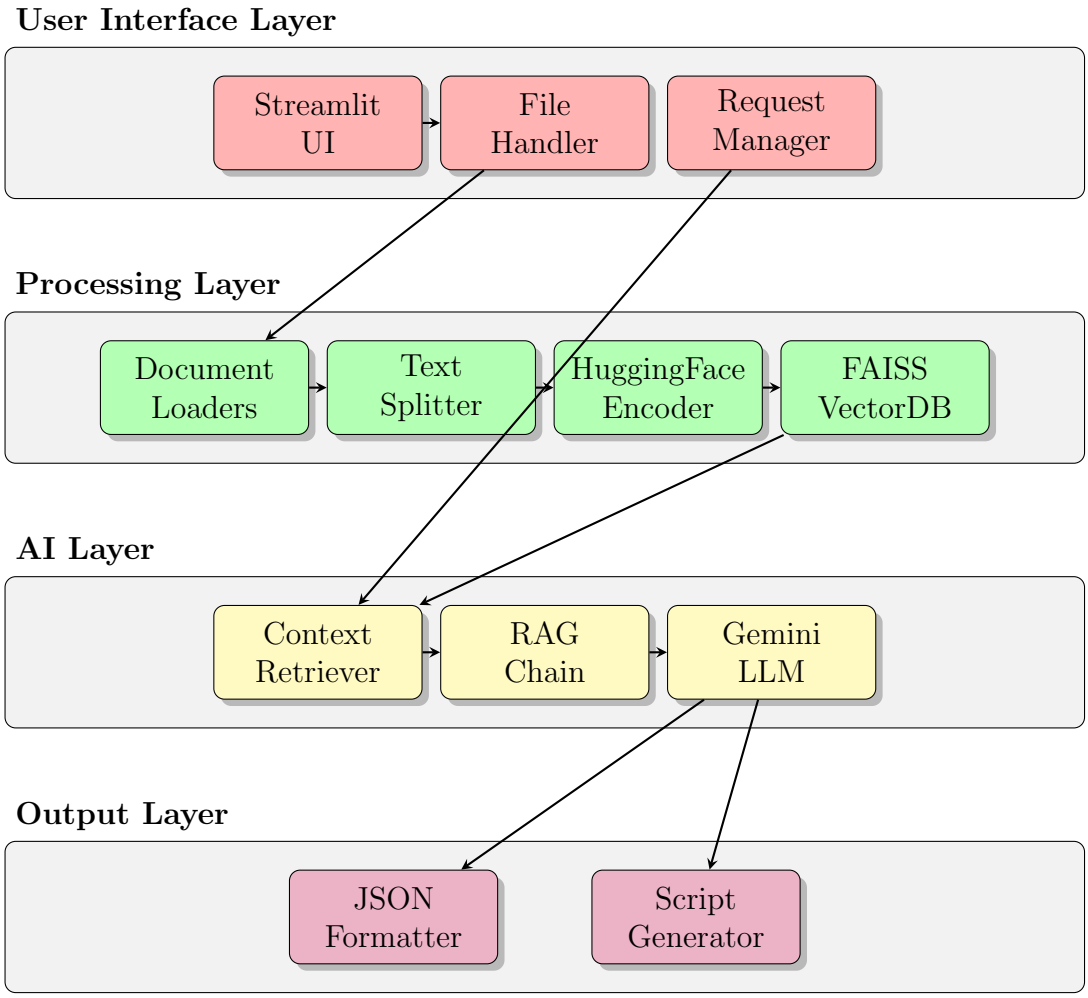


Figure 1: SMARTqA Layered Architecture

2.2 Component Interaction Diagram

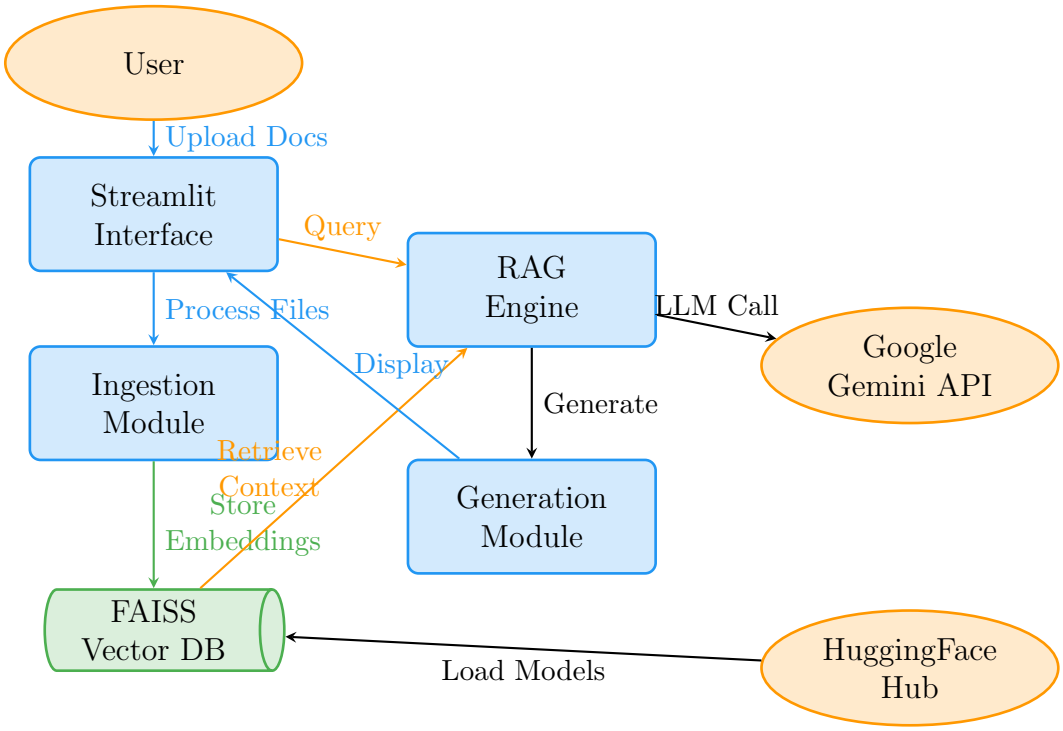


Figure 2: Component Interaction Flow

3 Data Flow Architecture

3.1 Document Ingestion Pipeline

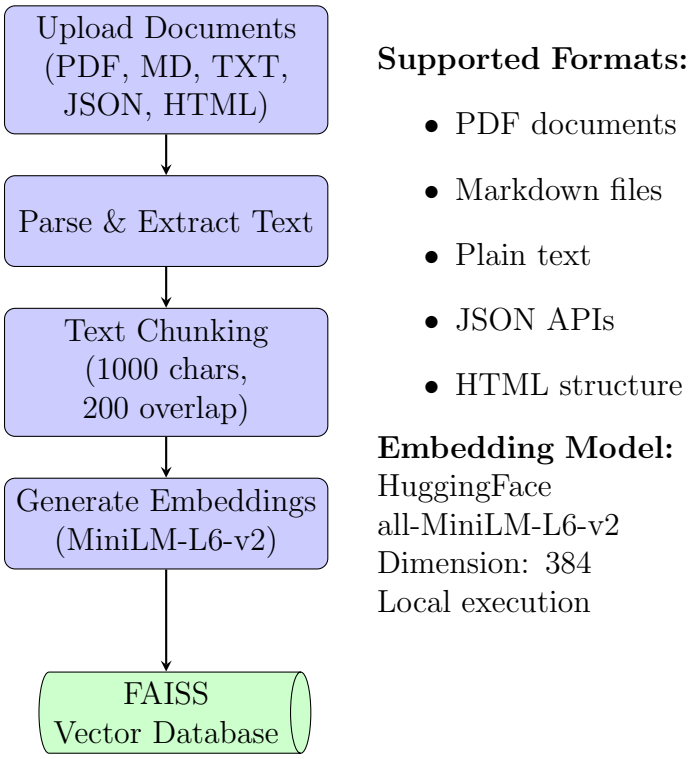


Figure 3: Document Ingestion Data Flow

3.2 Test Case Generation Flow

4 Core Components

4.1 Module Breakdown

Module	Responsibility
`app.py`	Main Streamlit interface, user interaction, workflow orchestration
`ingestion.py`	Document loading, text splitting, vector database creation
`rag.py`	LLM initialization and configuration
`generation.py`	Test case and Selenium script generation logic

Table 1: Core Module Responsibilities

4.2 Backend Module Architecture

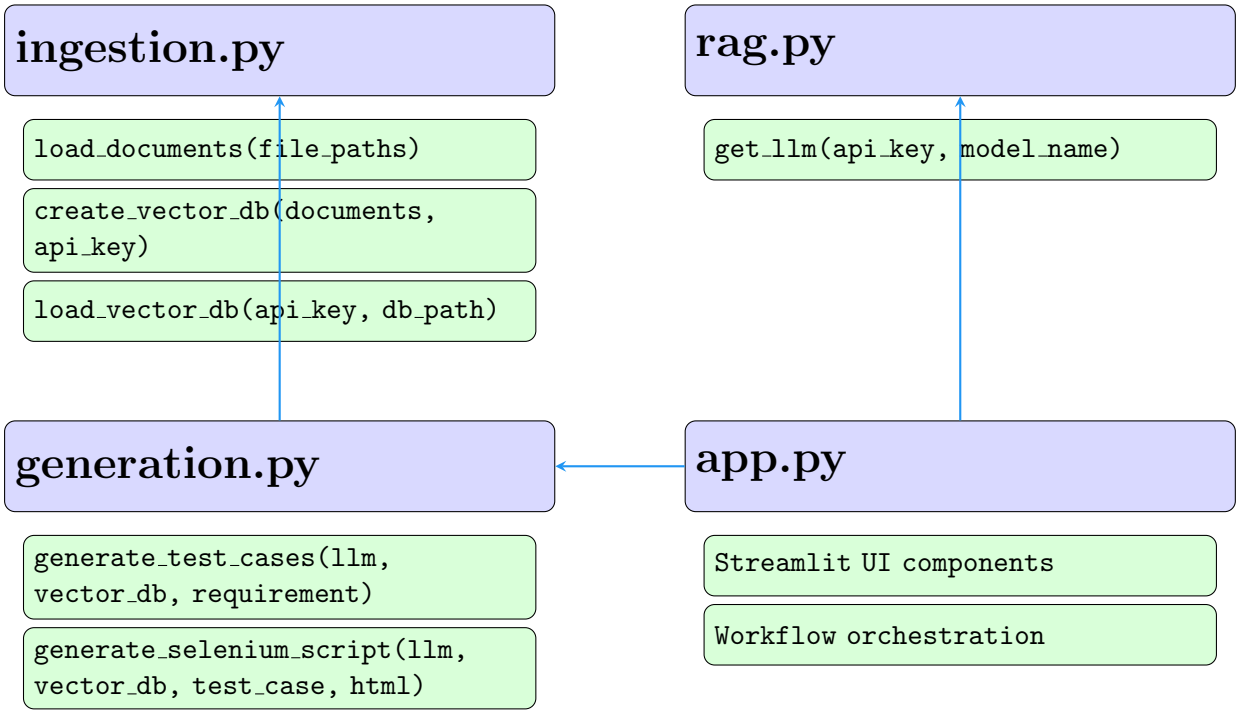


Figure 5: Backend Module Structure

5 Detailed Sequence Diagrams

5.1 Complete User Workflow

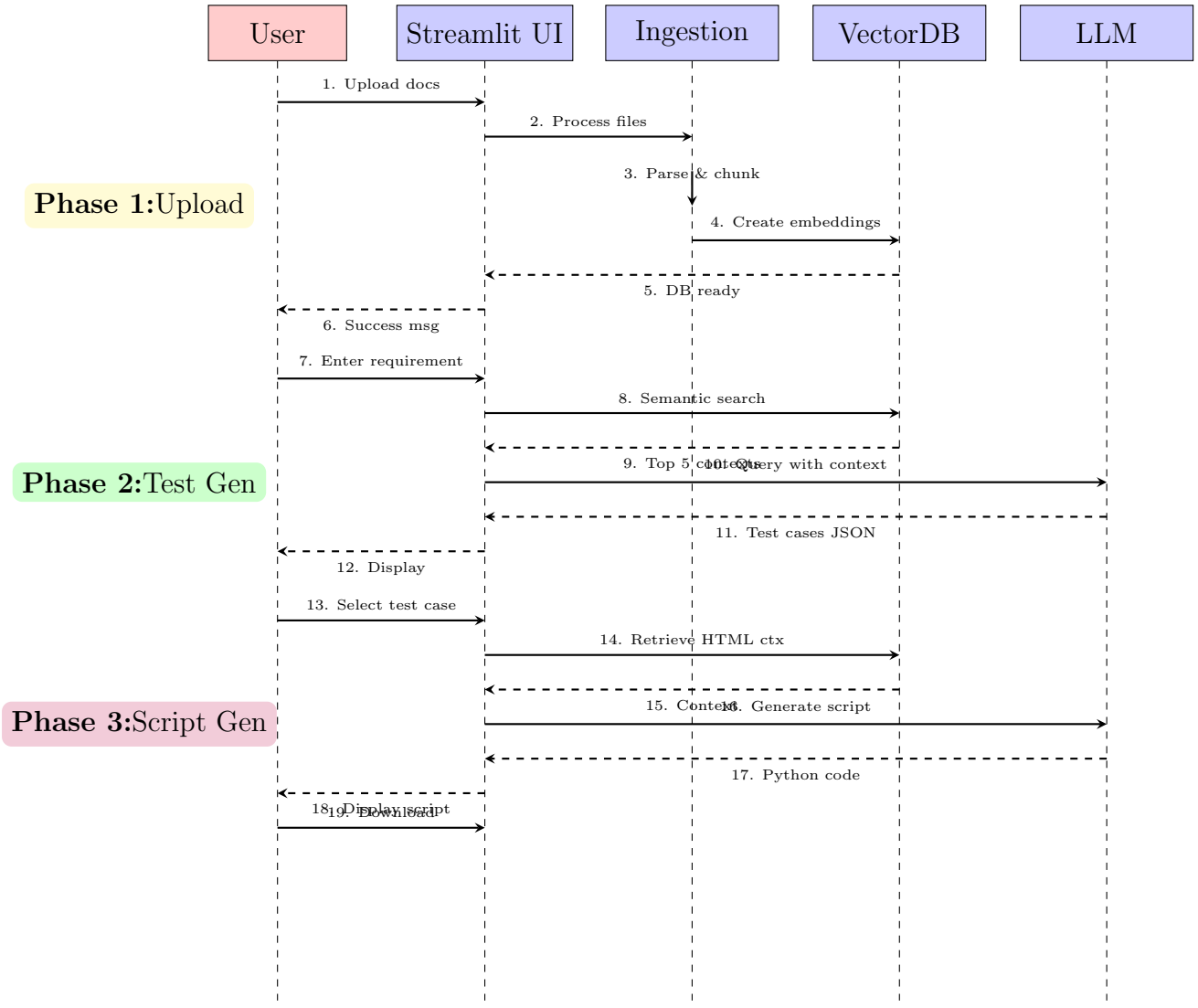


Figure 6: Complete User Workflow Sequence

6 Technology Stack

6.1 Technology Components

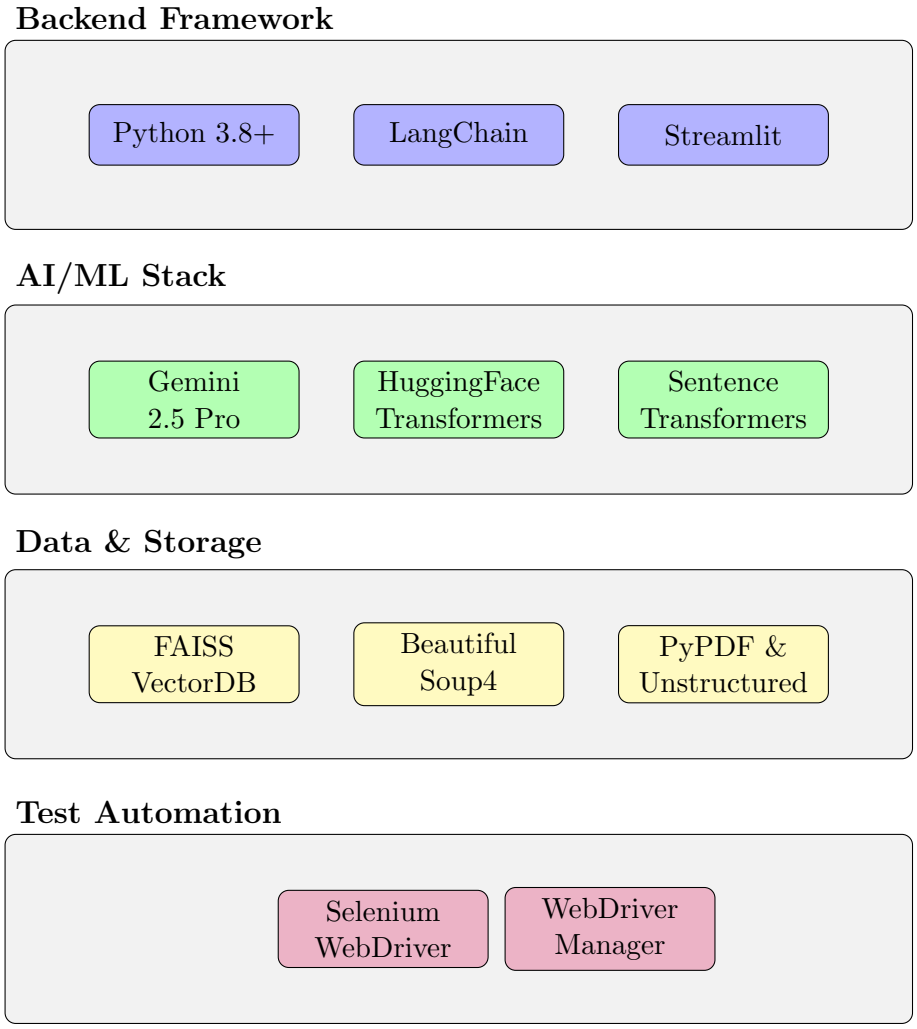


Figure 7: Technology Stack Layers

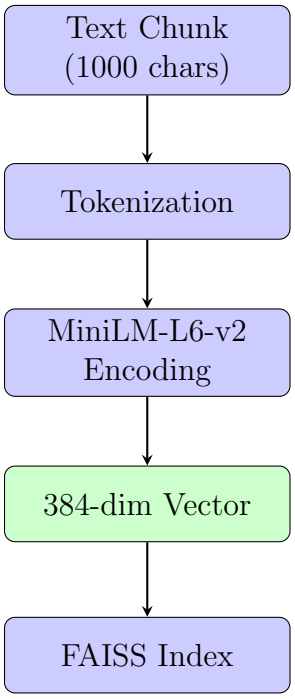
7 Core Algorithms

7.1 RAG Pipeline Algorithm

RAG Algorithm

```python
def generate_test_cases(llm, vector_db, requirement):
    # Step 1: Initialize retriever
    retriever = vector_db.as_retriever(
        search_kwargs={"k": 5}
    )

    # Step 2: Define prompt template
    template = """You are an expert QA engineer...
    Context: {context}
    Requirement: {requirement}
    Output Format: JSON list of test cases..."""

    prompt = PromptTemplate.from_template(template)

    # Step 3: Build RAG chain
    rag_chain = (
        {
            "context": retriever | format_docs,
            "requirement": RunnablePassthrough()
        }
        | prompt | llm | StrOutputParser()
    )

    # Step 4: Execute and return
    return rag_chain.invoke(requirement)
```

7.2 Vector Embedding Process



Model Details:
- Model: all-MiniLM-L6-v2
- Type: Sentence Transformer
- Output dim: 384
- Cosine similarity metric
- Local execution (no API)

FAISS Configuration:
- Index type: Flat (L2)
- Persistent storage
- Fast similarity search
- Memory efficient

Figure 8: Vector Embedding Pipeline

8 User Interface Design

8.1 Streamlit UI Structure

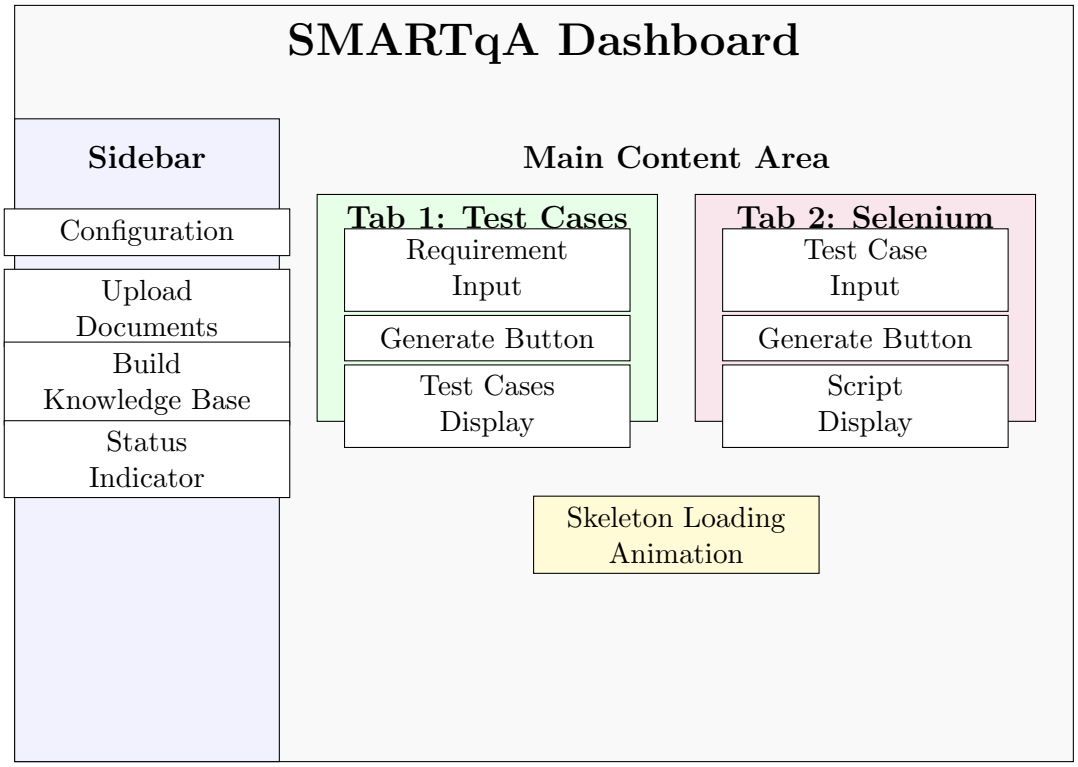


Figure 9: Streamlit UI Layout

9 Key Metrics and Performance

Metric	Value
Supported Document Formats	5 (PDF, MD, TXT, JSON, HTML)
Embedding Dimension	384 (MiniLM-L6-v2)
Context Window per Query	5 documents
Text Chunk Size	1000 characters
Chunk Overlap	200 characters
LLM Temperature	0.2 (deterministic)
Retrieval Strategy	Semantic similarity (FAISS)
Average Query Time	¡ 5 seconds
Embedding Model Size	80 MB (local)

Table 2: System Performance Metrics

10 Example Usage

10.1 Test Case Generation Example

Example Input

User Requirement:
"Test the discount code functionality with valid and invalid codes"

Generated Output

```
[
  {
    "Test_ID": "TC_001",
    "Feature": "Discount Code Validation",
    "Test_Scenario": "Apply valid 20% discount code",
    "Expected_Result": "Discount applied, price reduced",
    "Grounded_In": "product_specs.md"
  },
  {
    "Test_ID": "TC_002",
    "Feature": "Discount Code Validation",
    "Test_Scenario": "Apply invalid discount code",
    "Expected_Result": "Error message displayed",
    "Grounded_In": "ui_ux_guide.txt"
  }
]
```

11 Project Structure

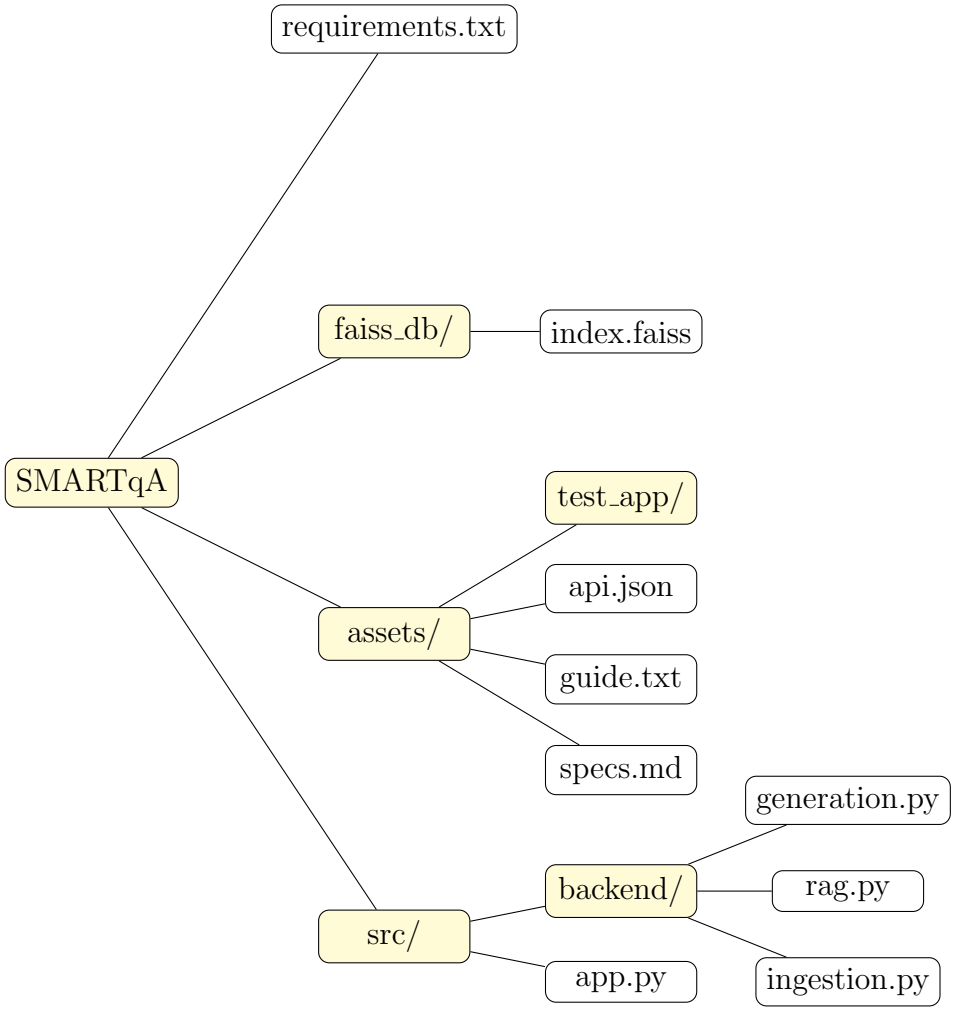


Figure 10: Project Directory Structure

12 Future Enhancements

- **Multi-Framework Support**: Playwright and Cypress script generation
- **Multi-Language**: JavaScript, TypeScript, Java test generation
- **Execution Dashboard**: Live test results and reporting
- **CI/CD Integration**: GitHub Actions, Jenkins pipeline support
- **Visual Regression**: Screenshot comparison testing
- **API Testing**: REST and GraphQL test generation
- **Performance Testing**: Load and stress test scenarios

13 Conclusion

SMARTqA represents a significant advancement in automated testing by combining the power of RAG with modern LLMs to create a truly intelligent QA agent. By grounding all test generation in actual project documentation, it eliminates hallucinations and ensures that tests are always relevant and accurate.

The modular architecture allows for easy extension and customization, while the use of open-source components (FAISS, HuggingFace) ensures cost-effectiveness and local control over sensitive data.

Key Takeaways

- **Zero Hallucination**: All outputs grounded in documentation
- **Production Ready**: Clean, executable Selenium scripts
- **Cost Effective**: Local embeddings, minimal API usage
- **User Friendly**: Intuitive Streamlit interface
- **Extensible**: Modular architecture for easy enhancement