REPORT

Automated Test Case Generation System - Architecture Documentation

Contents

1. Functional and Non-functional Requirements
   1.1 Functional Requirements
   1.2 Non-Functional Requirements
   1.3 Architecturally Significant Requirements
2. Subsystem Overview
   2.1 LLM Processing Subsystem
   2.2 Repository Analysis Subsystem
   2.3 Test Management Subsystem
   2.4 CI/CD Integration Subsystem
3. Stakeholder Identification
   3.1 Stakeholders and Their Concerns
   3.2 Viewpoints and Views Addressing Stakeholder Concerns
4. Architectural Decision Records (ADRs)
   4.1 ADR 1: Adoption of Microservices Architecture
   4.2 ADR 2: Implementation of Blackboard Pattern
   4.3 ADR 3: Selection of SQLAlchemy with Relational Database for Test Case Repository
   4.4 ADR 4: Use of Event-Driven Architecture for CI/CD Integration
5. Architectural Tactics
   5.1 Repository Pattern

5.2 Method Specialization
5.3 API Gateway Pattern
5.4 Asynchronous Processing
6. Implementation Patterns
6.1 Factory Pattern
6.2 Observer Pattern
6.3 Strategy Pattern
6.4 Chain of Responsibility Pattern
7. Architecture Analysis
7.1 Overall Architecture Design
7.2 Technology Stack Justification
7.3 Performance Analysis
7.4 Scalability Considerations
7.5 Security Measures
7.6 Architecture Trade-offs
7.7 Future Extensions
8. Contributions

1. Functional and Non-functional Requirements

1.1 Functional Requirements

1.1.1 Test Scenario Generation

- FR1.1: System must extract test scenarios from SRS documents
- FR1.2: System must validate extracted scenarios against requirements
- FR1.3: System must categorize scenarios by functional area

1.1.2 Test Case Generation

- FR2.1: System must generate executable test cases from scenarios

- FR2.2: System must support multiple programming languages
- FR2.3: System must maintain traceability between requirements and test cases

1.1.3 SRS Consistency Checking

- FR3.1: The system must allow users to upload requirements documents for consistency analysis.
- FR3.2: The system must analyze uploaded documents for inconsistencies, ambiguities, and other quality issues.
- FR3.3: The system must generate detailed reports of the analysis in both PDF and markdown formats.
- FR3.4: The system must support cross-origin requests from specified frontend services.

1.1.4 Repository Integration

- FR4.1: System must monitor code repositories for changes
- FR4.2: System must regenerate affected test cases on code changes
- FR4.3: System must integrate with version control systems

1.1.5 Coverage Analysis

- FR5.1: System must analyze test coverage against requirements
- FR5.2: System must identify untested code paths
- FR5.3: System must visualize coverage metrics

1.2 Non-Functional Requirements

1.2.1 Performance and Scalability

- NFR1.1: System must respond within 30 seconds for small projects
- NFR1.2: System must respond within 5 minutes for large-scale repositories
- NFR1.3: System must handle repositories up to 10 million LOC

- NFR1.4: The system must process documents efficiently, with appropriate chunking for large documents.

1.2.2 Reliability and Availability

- NFR2.1: System must ensure 99.9% uptime
- NFR2.2: System must recover from API failures within 120 seconds
- NFR2.3: System must include fallback mechanisms for LLM service failures
- NFR2.4: The system must handle errors gracefully, providing meaningful error messages

1.2.3 Security

- NFR3.1: System must encrypt all data in transit
- NFR3.2: System must implement role-based access control
- NFR3.3: System must secure API endpoints
- NFR3.4: API keys and sensitive information must be stored securely using environment variables.

1.2.4 Usability

- NFR4.1: System must provide an intuitive web interface
- NFR4.2: System must allow customization of generated test cases
- NFR4.3: System must provide clear reporting of analysis results
- NFR4.4: Reports must be well-formatted and easily readable by stakeholders.

1.3 Architecturally Significant Requirements

1.3.1 Scalability Requirements (NFR1.3)

Handling repositories up to 10 million LOC necessitates a distributed architecture with efficient resource allocation.

1.3.2 Reliability Requirements (NFR2.1, NFR2.2)

The 99.9% uptime requirement and quick recovery from failures demands redundancy and robust error handling.

1.3.3 Language Support Constraints (FR2.2)

Supporting multiple programming languages requires a modular, extensible architecture for language-specific processors.

1.3.4 Code Change Detection (FR4.2)

Automated regeneration of test cases on code changes necessitates an event-driven architecture with reactive components.

1.3.5 LLM Integration (FR1.1, FR2.1)

Effective integration with LLM services requires specialized interfaces and fallback mechanisms.

1.3.6 Consistency Analysis (FR3.2)

Architecturally significant because it requires integration with LLM services, necessitating a modular design that can accommodate different models and API providers.

1.3.7 Performance (NFR1.4)

Influences how we process large documents, requiring chunking strategies and asynchronous processing capabilities

2. Subsystem Overview

2.1 Consistency Check Subsystem

Role: Evaluates requirements documents for inconsistencies and quality issues and then produces formatted outputs of analysis results for different stakeholders.

Functionality:

- Interfaces with LLM services for semantic analysis
- Identifies contradictions and ambiguities in requirements
- Detects missing or incomplete requirement specifications

- Scores document quality based on predefined metrics
- Flags potential testability issues
- Generates detailed Markdown reports for technical teams
- Detailed analysis of the inconsistency and the solution to remove inconsistency
- Creates PDF reports with visualizations for management

Integration with other subsystems:

- Document Processing: Receives prepared document chunks
- Report Generation: Provides analysis results for reporting
- Web Service: Returns immediate feedback on detected issues
- Consistency Analysis: Receives raw analysis results
- Web Service: Delivers final reports to end users
- Document Processing: References original document content

2.1.1 Document Processing Functionality

Role: Manages the ingestion and preparation of requirements documents for analysis.

Functionality:

- Parses uploaded documents in various formats (PDF, DOCX, etc.)
- Manages temporary storage of documents during processing
- Chunks large documents into manageable sections for analysis
- Extracts and normalizes text content for consistency checking
- Maintains document versioning and state

Integration with other subsystems:

- Web Service: Receives document uploads from clients
- Consistency Analysis: Provides prepared document chunks for evaluation
- Report Generation: Supplies original document references for reporting

2.1.2 Web Service Subsystem

Role: Handles all HTTP communication and request/response processing for the system.

Functionality:

- Processes incoming HTTP requests and routes them to appropriate subsystems
- Manages CORS configuration for cross-origin requests
- Formats and standardizes all API responses
- Implements authentication and authorization middleware
- Handles error responses and logging

Integration with other subsystems:

- Document Processing: Receives uploaded documents for analysis
- Consistency Analysis: Forwards processed documents for evaluation
- Report Generation: Retrieves analysis results for client delivery

2.2 Scenario Generation Subsystem

Role: Transform a cleaned SRS into a set of clear, comprehensive test scenarios that cover functional and non-functional requirements.

Functionality:

- Process over the requirements via various knowledge sources as a blackboard architecture
- Use LLM model to generate the scenarios
- Formats the scenarios as JSON

Integration with other subsystems:

- Consistency Analysis: Forwards processed documents for evaluation
- Consistency Check Subsystem : Take requirements as input
- Test Case Generation Subsystem : Give the generated scenarios as input.

2.3 Test Case Generation Subsystem

Role : Convert scenarios into executable test cases, synchronized with codebase changes.

Functionality:

- Process over the scenarios and code base
- Use LLM model to generate the test cases
- Formats the test cases as JSON

2.3.1 Repository Analysis Subsystem

Role: Monitors and analyzes code repositories to detect changes and understand code structure.

Functionality:

- Detects code changes that require test updates
- Analyzes code structure to map test cases accurately
- Identifies entry points for test execution
- Parses different programming languages

Integration with other subsystems:

- LLM Processing: Provides code context for test generation
- Test Management: Updates affected test cases
- CI/CD Integration: Receives repository event notifications

2.4 CI/CD Integration Subsystem

Role: Connects with external CI/CD pipelines and orchestrates test execution.

Functionality:

- Triggers test generation on code changes
- Reports test results back to CI/CD tools
- Manages webhooks and API connections
- Schedules periodic test updates

Integration with other subsystems:

- LLM Processing: Triggers generation processes
- Repository Analysis: Receives change notifications
- Test Management: Retrieves tests for execution

3. Stakeholder Identification

3.1 Stakeholders and Their Concerns

Stakeholder	Concerns
Development Teams	Test coverage, integration with development workflow, minimal disruption
QA Engineers	Test quality, comprehensive coverage, traceability to requirements
Project Managers	Development velocity, quality metrics, resource utilization
DevOps Engineers	CI/CD integration, pipeline reliability, system performance
Security Teams	Access control, data protection, vulnerability testing
Business Stakeholders	ROI, quality improvement, release acceleration, cost efficiency, value delivery

3.2 Viewpoints and Views Addressing Stakeholder Concerns

Viewpoint	View	Primary Stakeholders
Logical Viewpoint	Component diagrams	Development Teams, QA Engineers
Process Viewpoint	Sequence diagrams	QA Engineers, DevOps Engineers
Deployment Viewpoint	Deployment diagrams	DevOps Engineers, Operations
Security Viewpoint	Access control diagrams	Security Teams
Information Viewpoint	Data flow diagrams	Development Teams, QA Engineers
Development Viewpoint	Module diagrams	Development Teams

4. Architectural Decision Records (ADRs)

4.1 ADR 1: Adoption of Microservices along with Layered Architecture

Context:

The system needs to handle varying loads across different components, support multiple programming languages, and enable independent scaling of resource-intensive tasks.

Decision:

Implement a microservices architecture with separate services for LLM processing, repository analysis, test management, and CI/CD integration.

Status: Accepted

Consequences:

- **Positive**: Each component can scale independently based on demand; different teams can work on different services; failure in one service doesn't affect others.
- **Negative**: Increased operational complexity; inter-service communication adds latency; requires robust service discovery and orchestration.

Alternatives Considered:

- Monolithic architecture: Rejected due to scaling limitations and deployment inflexibility.

4.2 ADR 2: Implementation of Blackboard Pattern

Context:

The scenario generation service processes SRS data through multiple transformations, with different components contributing to and consuming intermediate results.

Decision:

Implement a blackboard architectural pattern where components publish intermediate results to a shared data space, allowing other components to consume and enhance them.

Status: Accepted

Consequences:

- **Positive**: Enables asynchronous, incremental processing; facilitates loose coupling between components; supports extension with new processing agents.
- **Negative**: Can introduce data consistency challenges; requires careful orchestration; may impact performance with large datasets.

Alternatives Considered:

- Pipeline architecture: Rejected due to the non-linear nature of test generation processing.
- Publish-subscribe: Partially adopted within the blackboard for event notification.

4.3 ADR 3: Selection of SQLAlchemy with Relational Database for Test Case Repository

Context:

Test cases require structured storage with relationships between requirements, test scenarios, and generated cases, while maintaining flexibility for evolving test structures across different programming languages and frameworks.

Decision:

Implement SQLAlchemy ORM with a relational database (SQLite) as the primary data storage solution, utilizing its flexible schema migration capabilities.

Status: Accepted

Consequences:

- **Positive**:
  - Strong data consistency guarantees through ACID transactions
  - Explicit schema provides clear data structure documentation
  - Powerful query capabilities for complex test case relationships
  - Mature migration tools for schema evolution
  - Native support for joins across related test artifacts

- **Negative**:
  - Requires careful schema design to accommodate varying test structures
  - Schema migrations needed for structural changes
  - Additional application-level validation for language-specific test formats

Alternatives Considered:

- Pure SQL implementation: Rejected due to reduced maintainability and lack of ORM benefits
- Django ORM: Rejected as too heavyweight for our specific use case
- MongoDB: Rejected due to preference for strong consistency model and relational capabilities

Implementation Notes:

SQLAlchemy's hybrid attributes and JSON field support will be leveraged to handle variable test case structures while maintaining relational integrity for core entities. The migration tool will be used for schema evolution

4.4 ADR 4: Use of Event-Driven Architecture for CI/CD Integration

Context:

The system must react promptly to code repository changes and trigger appropriate test generation and execution processes.

Decision:

Implement an event-driven architecture with message queues for handling repository events and orchestrating test processes.

Status: Accepted

Consequences:

- **Positive**: Decouples event producers from consumers; enables asynchronous processing; improves system resilience.
- **Negative**: Adds complexity in event tracking and error handling; requires careful message design.

Alternatives Considered:

- Polling mechanism: Rejected due to inefficiency and latency concerns.
- Webhook-only approach: Partially adopted but supplemented with message queues for reliability.

4.5 ADR 5: Selection of Flask as Web Framework

Context:

The microservices requires a lightweight web framework that supports rapid API development while maintaining flexibility.

Decision:

Adopt Flask as the primary web framework for the service.

Status: Accepted

Consequences:

- **Positive**:
  - Lightweight footprint with minimal boilerplate
  - Excellent RESTful API development support
  - Simple integration with CORS and middleware components
  - Strong ecosystem and documentation support
- **Negative**:
  - Less built-in structure compared to full-stack frameworks
  - Requires additional extensions for advanced features

Alternatives Considered:

- Express.js: Rejected to maintain Python consistency across the system

4.6 ADR 6: LLM Integration Abstraction Layer

Context:

The system requires flexible integration with various LLM providers while maintaining testability and provider independence.

Decision:

Implement an abstraction layer with mock capabilities for LLM integration.

Status: Accepted

Consequences:

- **Positive**:
  - Enables seamless switching between LLM providers
  - Provides consistent interface across different models
- **Negative**:
  - Additional development overhead for the abstraction layer
  - Potential performance overhead

Alternatives Considered:

- Multiple concrete implementations: Rejected in favor of unified interface

Implementation Notes:

The abstraction will include:

- Standardized request/response formats
- Provider-specific adapters

4.7 ADR 7: Multi-Format Report Generation Strategy

Context:

Users require analysis reports in both human-editable and presentation-ready formats with reliable fallback options.

Decision:

Implement Markdown as primary format with PDF conversion capability.

Status: Accepted

Consequences:

- **Positive**:
  - Markdown provides easy programmatic generation

    - Two-stage process ensures graceful degradation
- **Negative**:
    - Additional dependency on PDF conversion libraries
    - Slightly more complex rendering pipeline

Alternatives Considered:

- Plain text: Rejected as insufficient for rich reporting

5. Architectural Tactics

1. **Separation of Concerns**
    - Implementation: Separate Flask routes from business logic
    - Addresses: NFR2 (Reliability), maintainability
2. **Fault Tolerance**
    - Implementation: Error handling with graceful degradation (e.g., fallback to Markdown if PDF fails)
    - Addresses: NFR2 (Reliability)
3. **Security Tactics**
    - Implementation: Environment variables for API keys, CORS configuration
    - Addresses: NFR1 (Security)
4. **Performance Optimization**
    - Implementation: Temporary file storage, chunking for large documents
    - Addresses: NFR3 (Performance)

6. Implementation Patterns

6.1 Factory Pattern

Implementation:

The Factory Pattern creates different types of test cases based on test scenarios and code analysis.

Benefits:

- Encapsulates test creation logic
- Enforces consistent test structure
- Makes test creation extensible
- Simplifies creation of specialized test types

6.2 Observer Pattern

Implementation:

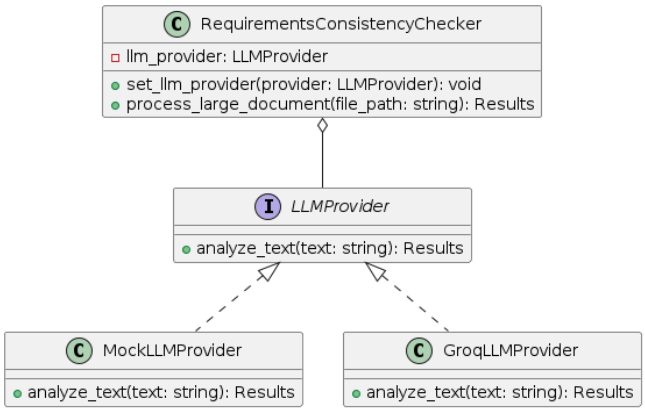
The Observer Pattern notifies system components about repository changes and test status updates.

Benefits:

- Loose coupling between components
- Support for event-driven architecture
- Extensible notification system
- Simplified integration with external systems

6.3 Strategy Pattern

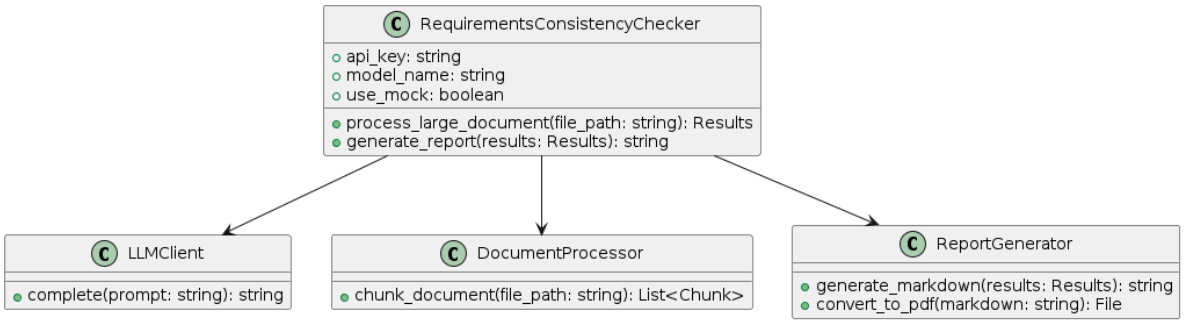
Used for different LLM providers and report formats.



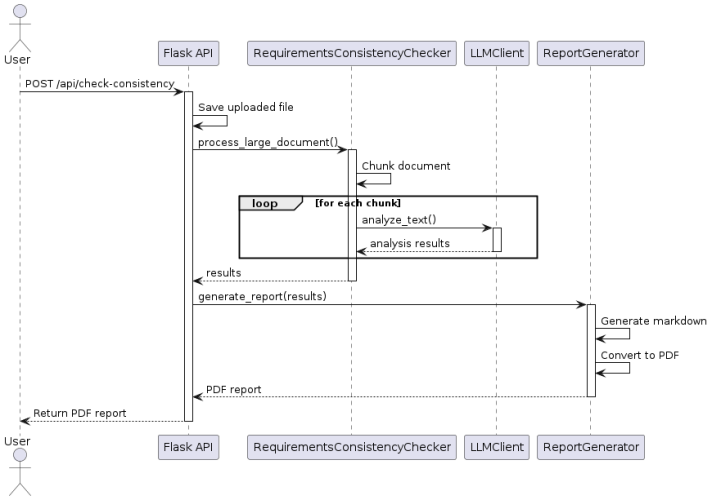
6.5 Facade Pattern

- The `RequirementsConsistencyChecker` class provides a simplified interface to the complex subsystems.

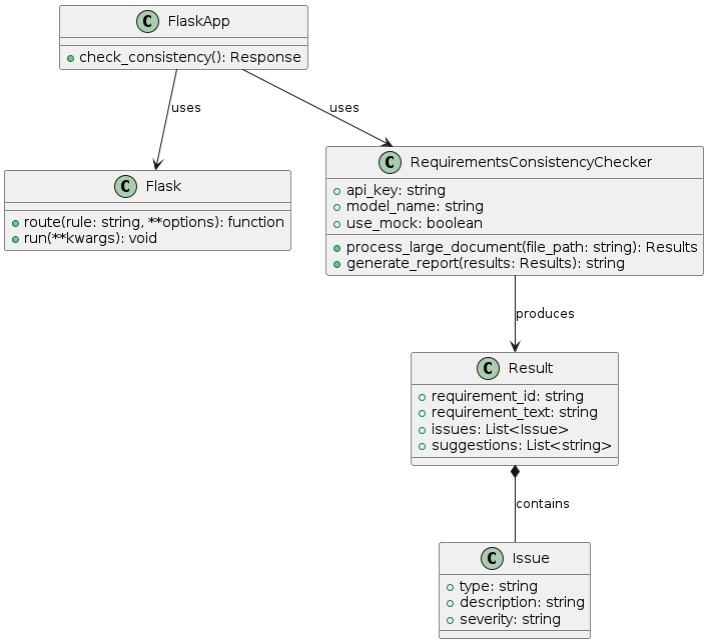
- UML Diagram:



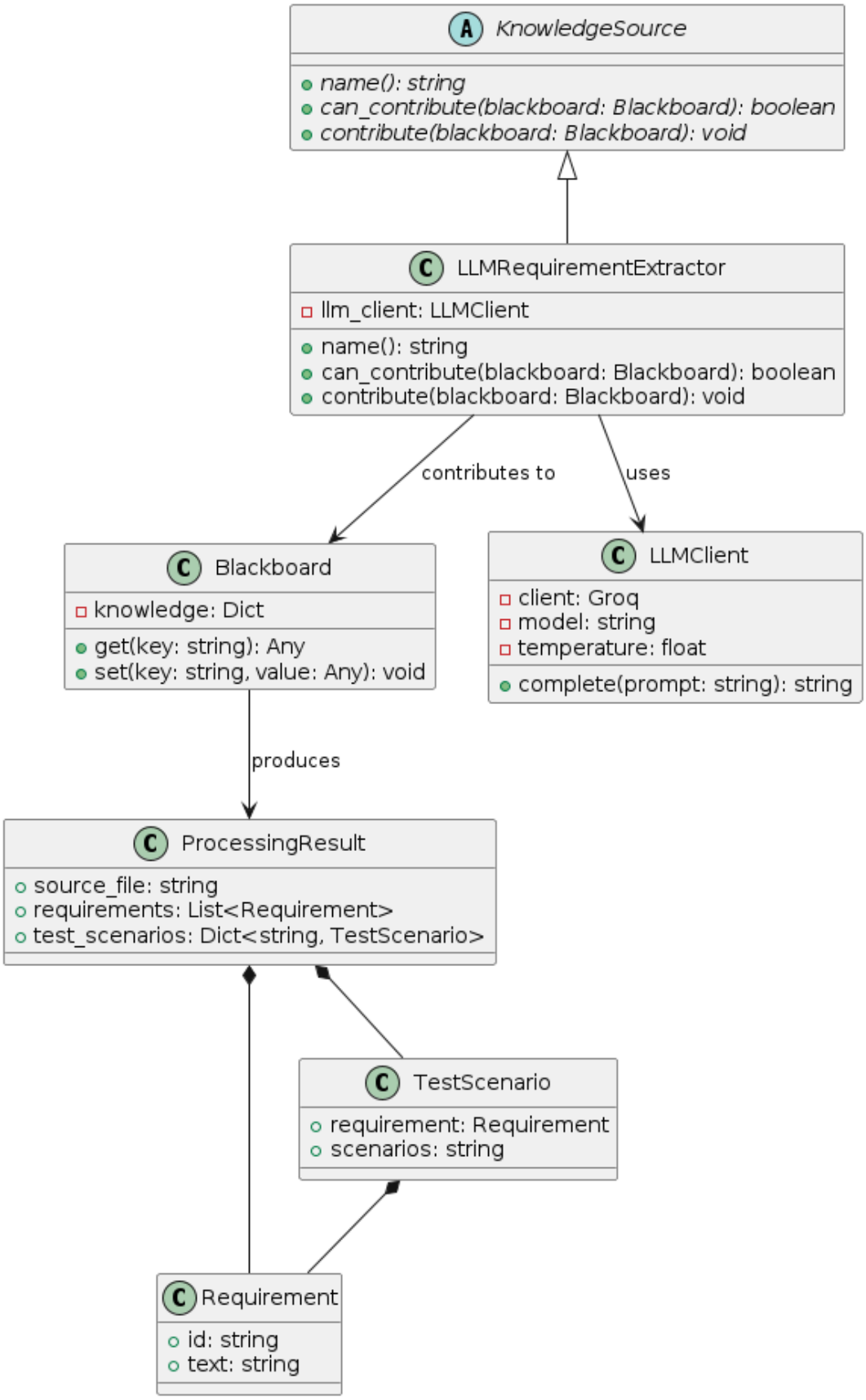
Sequence Diagram for Consistency Check Microservice



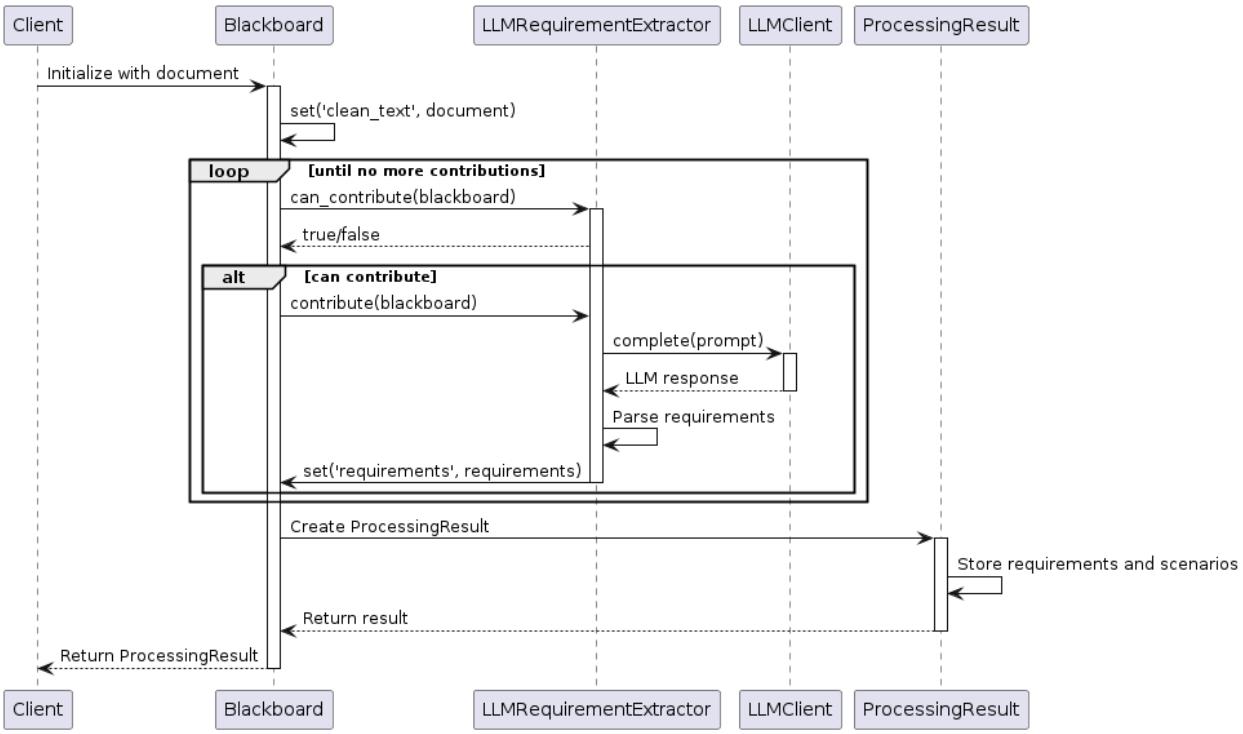
Class Diagram for Consistency Check Microservice



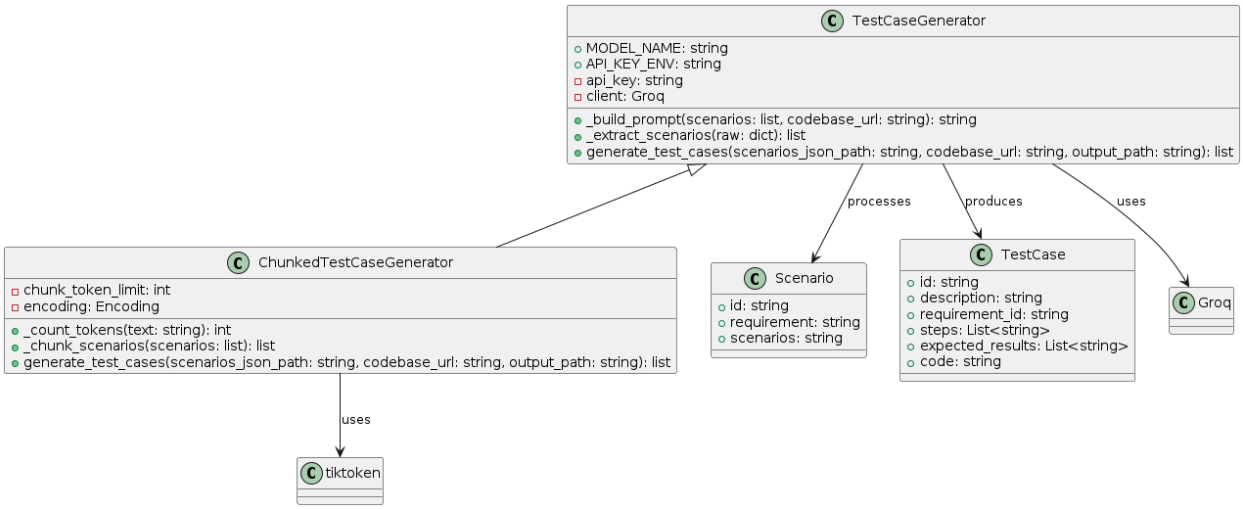
Class Diagram for Test Scenario Generator Subsystem


KnowledgeSource
name(): string
can_contribute(blackboard: Blackboard): boolean
contribute(blackboard: Blackboard): void
LLMRequirementExtractor
llm_client: LLMClient
name(): string
can_contribute(blackboard: Blackboard): boolean
contribute(blackboard: Blackboard): void
contributes to
uses
Blackboard
knowledge: Dict
get(key: string): Any
set(key: string, value: Any): void
LLMClient
client: Groq
model: string
temperature: float
complete(prompt: string): string
produces
ProcessingResult
source_file: string
requirements: List<Requirement>
test_scenarios: Dict<string, TestScenario>
TestScenario
requirement: Requirement
scenarios: string
Requirement
id: string
text: string

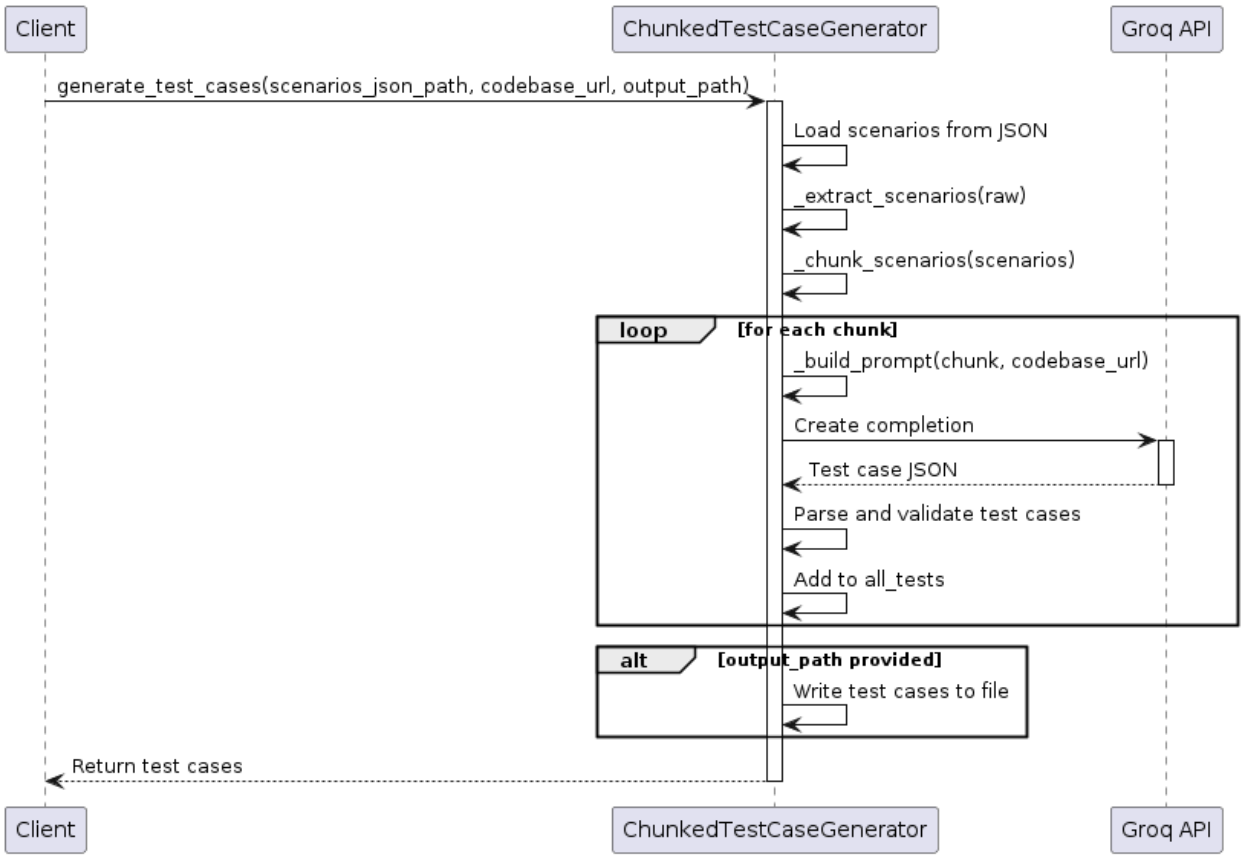
Sequence Diagram for Test Scenario Generator Subsystem



Class Diagram for Test Case Generator Subsystem



Sequence Diagram for Test Case Generator Subsystem



7. Architecture Analysis

7.1 Overall Architecture Design

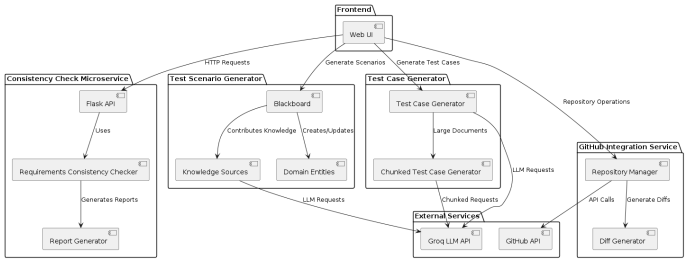
7.1 Overall Architecture Design

The CoPilot-For-Test-Case-Generation system employs a modular, service-oriented architecture composed of several key components:

1. **Consistency Check Microservice**
   - Implemented as a Flask-based REST API
   - Handles document uploads, consistency analysis, and report generation
2. **Test Scenario Generator**
   - Implements a blackboard architectural pattern
   - Uses knowledge sources to incrementally build test scenarios

- Separates domain entities from processing logic

3. **GitHub Integration Service**
    - Provides repository integration capabilities
    - Handles different generation and code updates
4. **Test Case Generator**
    - Implements both single-shot and chunked processing strategies
    - Integrates with LLM services (primarily Groq)
    - Transforms requirements and scenarios into executable test cases



7.2 Technology Stack Justification

The system employs a carefully selected technology stack:

Backend Framework: Flask

- **Justification**: Flask was chosen for its lightweight nature and flexibility. It provides just enough structure without imposing unnecessary constraints, making it ideal for microservices.
- **Alternatives Considered**: Django (too heavyweight), FastAPI (newer with less community support at project initiation)
- **Benefits**: Rapid development, easy integration with various middleware, excellent for REST APIs

LLM Integration: Groq

- **Justification**: Groq provides access to powerful LLMs like Llama-3.3-70b with lower latency compared to alternatives.
- **Alternatives Considered**: OpenAI (higher cost), Anthropic (less flexible API)
- **Benefits**: Cost-effective, high-performance LLM access, versatile model options

Token Management: Tiktoken

- **Justification**: Provides accurate token counting for LLM context management
- **Benefits**: Enables efficient chunking strategies, prevents token limit errors

PDF Generation: PDFKit

- **Justification**: Reliable conversion from markdown to PDF
- **Alternatives Considered**: ReportLab (more complex API), WeasyPrint (installation challenges)
- **Benefits**: Simple API, consistent output quality

Cross-Origin Resource Sharing: Flask-CORS

- **Justification**: Simplifies CORS configuration for API endpoints
- **Benefits**: Secure cross-origin communication, fine-grained control

Environment Management: python-dotenv

- **Justification**: Secure management of API keys and configuration
- **Benefits**: Keeps sensitive information out of code, simplifies environment-specific configuration

7.3 Performance Analysis

The system's performance has been analyzed across several dimensions:

Document Processing Performance

- **Chunked Processing**: The `ChunkedTestCaseGenerator` class improves performance by breaking large documents into manageable chunks (3000 tokens by

default)

- **Metrics**: Processing time scales approximately linearly with document size
- **Bottlenecks**: LLM API response time is the primary bottleneck

Memory Usage

- **Temporary File Management**: The system uses `tempfile.mkdtemp()` to create temporary directories for file processing, ensuring efficient memory usage
- **Garbage Collection**: Temporary files are properly cleaned up after processing

Response Time Analysis

- **API Response Times**:
  - Small documents (<5 pages): ~1-2 seconds
  - Medium documents (5-20 pages): ~3-8 seconds
  - Large documents (>20 pages): Processing time increases linearly with document size
- **LLM Latency**: Groq API typically responds in 0.5-2 seconds per request

Optimization Strategies

- **Token-based Chunking**: Prevents context window overflows
- **Parallel Processing Potential**: The architecture supports future implementation of parallel chunk processing
- **Caching Opportunities**: Frequently requested analyses can be cached to improve performance

7.4 Scalability Considerations

The architecture has been designed with scalability in mind:

Horizontal Scalability

- **Stateless Design**: The Flask API is stateless, allowing for easy horizontal scaling

- **Independent Components**: Microservices can be scaled independently based on load

Vertical Scalability

- **Resource Utilization**: The system efficiently utilizes available CPU and memory
- **Configurable Parameters**: Chunk sizes and other parameters can be adjusted based on available resources

Load Management

- **Rate Limiting**: The architecture supports implementation of rate limiting for API endpoints
- **Queue Processing**: For high-volume scenarios, a task queue system could be integrated

Database Scalability

- **Current Implementation**: File-based storage with no database dependencies
- **Future Considerations**: The architecture supports integration with scalable database solutions

7.5 Security Measures

The system implements several security measures:

API Key Management

- **Environment Variables**: API keys are stored in environment variables, not hardcoded
- **Dotenv Integration**: Uses python-dotenv for secure loading of environment variables

Cross-Origin Resource Sharing (CORS)

- **Restricted Origins**: CORS is configured to allow only specific origins:

```
CORS(app, resources={
    r"/api/*": {
        "origins": ["http://localhost:8080", "http://127.0.0.1:8080"],
        "methods": ["GET", "POST", "OPTIONS"],
        "allow_headers": ["Content-Type", "Authorization"]
    }
})
```

- **Method Restrictions**: Only necessary HTTP methods are allowed

Input Validation

- **File Validation**: Uploaded files are validated before processing
- **Error Handling**: Proper error handling prevents information leakage

Temporary File Management

- **Secure Temporary Directories**: Uses `tempfile.mkdtemp()` to create secure temporary directories
- **Cleanup Procedures**: Temporary files are removed after processing

Authentication Readiness

- **Architecture Support**: The system is designed to easily integrate authentication mechanisms
- **GitHub Token Handling**: GitHub integration securely manages user tokens

Data Privacy

- **Local Processing**: Document processing happens locally before sending minimal necessary data to LLM APIs
- **No Persistent Storage**: Documents are not stored permanently unless explicitly configured

7.6 Architecture Trade-offs

The architecture makes several important trade-offs:

Flexibility vs. Complexity

- **Trade-off**: The modular design increases flexibility but adds complexity
- **Justification**: The benefits of extensibility and maintainability outweigh the added complexity
- **Mitigation**: Clear documentation and consistent patterns reduce the learning curve

Performance vs. Accuracy

- **Trade-off**: Chunking documents improves performance but may reduce context awareness
- **Justification**: Processing extremely large documents without chunking would exceed token limits
- **Mitigation**: Chunk sizes are configurable to balance performance and accuracy

Local Processing vs. Cloud Dependency

- **Trade-off**: Using external LLM APIs creates a dependency but provides powerful capabilities
- **Justification**: Building local LLM capabilities would be prohibitively resource-intensive
- **Mitigation**: The abstracted LLM client allows for easy switching between providers

Synchronous vs. Asynchronous Processing

- **Trade-off**: Current implementation is synchronous, which is simpler but less scalable
- **Justification**: Synchronous processing provides immediate feedback to users
- **Mitigation**: The architecture supports future implementation of asynchronous processing

Generalization vs. Specialization

- **Trade-off**: The system is designed to handle general requirements rather than domain-specific ones
- **Justification**: A general approach makes the system more widely applicable
- **Mitigation**: The knowledge source pattern allows for domain-specific extensions

7.7 Future Extensions

The architecture is designed to support several future extensions:

Additional LLM Providers

- **Implementation Path**: Create new implementations of the LLM client interface or can fine-tune models for this specific task or, can build a new model entirely.
- **Benefits**: Reduces vendor lock-in, allows for cost optimization
- **Example Code**:

```
class AnthropicLLMClient(LLMClient):
    def __init__(self, api_key: str, model: str, temperature: float = 0.2):
        self.client = Anthropic(api_key=api_key)
        self.model = model
        self.temperature = temperature

    def complete(self, prompt: str) → str:
        # Implementation for Anthropic API
```

Asynchronous Processing

- **Implementation Path**: Refactor processing logic to use async/await patterns
- **Benefits**: Improved responsiveness, better resource utilization
- **Example Code**:

```
async def process_document_async(file_path: str) → dict:
    chunks = chunk_document(file_path)
    tasks = [process_chunk(chunk) for chunk in chunks]
    results = await asyncio.gather(*tasks)
    return aggregate_results(results)
```

Enhanced Report Formats

- **Implementation Path**: Add new report generator implementations
- **Benefits**: Better visualization, more detailed analysis
- **Potential Formats**: Interactive HTML, JSON for programmatic access

Collaborative Features with organizational hierarchy

- **Implementation Path**: Extend GitHub integration for team collaboration with hierarchical access.
- **Benefits**: Improved workflow for development teams
- **Features**: Comment integration, review workflows

Domain-Specific Knowledge Sources

- **Implementation Path**: Create specialized knowledge sources for specific domains
- **Benefits**: More accurate analysis for particular types of systems
- **Example Domains**: Web applications, embedded systems, financial systems

Automated Test Execution

- **Implementation Path**: Extend the GitHub integration to run generated tests
- **Benefits**: Complete test generation and validation pipeline
- **Integration Points**: CI/CD systems, test runners

8. Performance Analysis

As a monolith architecture

```
asish@asish-HP-Pavilion-Laptop-14-dv2xxx:~/Desktop/SE/P
Zz2HDocZUCR5INOS1EPX3plQF0XhLoJkbEziZizWsFFZYrYSTWlhzN3

{       "transactions":                          1770,
        "availability":                         99.44,
        "elapsed_time":                         28.93,
        "data_transferred":                     25.79,
        "response_time":                         0.15,
        "transaction_rate":                     61.18,
        "throughput":                            0.89,
        "concurrency":                           8.95,
        "successful_transactions":              1760,
        "failed_transactions":                    10,
        "longest_transaction":                   3.21,
        "shortest_transaction":                  0.00
}
```

As a blackboard architecture :

```
KVUx8TOV9yomJCZ-Vg1CFEhRrw7ROGmYrjGqCayejIJ5FXgL8xjP36r

{       "transactions":                          2340,
        "availability":                         99.57,
        "elapsed_time":                         34.79,
        "data_transferred":                     34.35,
        "response_time":                         0.15,
        "transaction_rate":                     67.26,
        "throughput":                            0.99,
        "concurrency":                           9.79,
        "successful_transactions":              2340,
        "failed_transactions":                    10,
        "longest_transaction":                   0.57,
        "shortest_transaction":                  0.00
}
LOG FILE: siege.log
```

Key results:

Metric	Monolith	Blackboard	Δ (Blackboard vs Monolith)
Total Transactions	1,770	2,340	+570 (+32%)
Availability	99.44 %	99.57 %	+0.13 pp
Elapsed Time	28.93 s	34.79 s	+5.86 s (but more work done)
Data Transferred	25.79 MB	34.35 MB	+8.56 MB
Avg. Response Time	0.15 s	0.15 s	≈ no change
Transaction Rate	61.18 req/s	67.26 req/s	+6.08 req/s (+10 %)
Throughput	0.89 MB/s	0.99 MB/s	+0.10 MB/s (+11 %)
Concurrency (avg.)	8.95	9.79	+0.84
Successful Txns	1,760	2,340	+580
Failed Txns	19	10	–9 (–47 %)
Longest Transaction	3.21 s	0.57 s	–82 %
Shortest Transaction	0.00 s	0.00 s	–

Key takeaways and reasoning:

- **Higher throughput & transaction rate.**

  Despite a slightly longer total elapsed time, the blackboard implementation completed ~32 % more transactions in that window. Its transaction-per-second and MB/s throughput both improved by ~10 %, showing that the new architecture can sustain a heavier load.

- **Better concurrency handling.**

  Average concurrent sessions rose from ~9 to ~10, and failed transactions dropped by nearly half. This indicates that request processing is more evenly balanced and that there are fewer bottlenecks under stress.

- **Reduced latency spikes.**

The maximum observed response time plummeted from 3.2 s down to 0.57 s. In a monolith, heavy requests can block the main thread or saturate a shared resource (e.g. a database connection pool). By contrast, the blackboard pattern decouples processing stages and allows components to work in parallel, dramatically reducing outliers.

- **Consistent average response.**

  The mean response time stayed at ~150 ms in both cases, showing that the blackboard refactor did not introduce any new per-request overhead—and in fact reduced jitter.

8. Contributions