Piotr Copek
Zuzanna Micorek
Hyunseok Cho
Szymon Molicki
Mateusz Znaleźniak

31.01.2026

Use Case Modelling

Software Engineering

Index

1. Introduction
2. System Implementation
3. Database Design
4. Client-Side Implementation
5. Server-Side Implementation
6. Testing and Validation
7. Performance Evaluation
8. Conclusion

1. Introduction

This document presents the final implementation of the Email Spam Detection System, developed as part of the Software Engineering course. The system builds on iterative development through Prototype 1 and Prototype 2 and results in a deployable web application for spam classification.

1.1 System Overview

The Email Spam Detection System is a stateless web application that provides real time spam classification using machine learning models. The system prioritizes privacy, transparency, and simplicity, and operates without data persistence.

Key features include:

- BERT based binary classification for spam detection
- Explainable AI through IntegratedGradients keyword attribution
- Robust error handling with exponential backoff retry logic
- Comprehensive logging system for debugging and monitoring
- Alternative sliding window classifier for long texts

1.2 Development Evolution

The system evolved through three major phases:

Prototype 1 established the foundational architecture with basic spam detection capability and web interface integration using Flask and Jinja2 templates.

Prototype 2 extended functionality with keyword explanations using IntegratedGradients, retry logic with exponential backoff, comprehensive logging infrastructure, and resource preloading for performance optimization.

Final Version completed the system with testing, performance benchmarking, security validation, and deployment preparation.

2. System Implementation

2.1 Architecture Overview

The system follows a layered architecture:

- **Presentation Layer**: `Flask` Blueprint with `Jinja2` templates for web interface
- **Service Layer**: `SpamService` providing classification facade and singleton pattern
- **Model Layer**: `SpamChecker` and `GeneralRiskOverseer` for inference
- **Core Layer**: Configuration management and logging infrastructure

2.2 Component Implementation

SpamChecker Module

The SpamChecker class implements the core classification logic using a pretrained BERT model. It provides three main methods: `predict(text)` performs binary classification and returns spam probability and confidence score. The method applies `PyTorch` inference with gradient computation disabled for efficiency. `explain(text, top_k)` generates keyword explanations using IntegratedGradients attribution. The method calculates token-level attributions, merges wordpieces, filters stop words, and returns the `top-k` most influential terms. `analyze(text, explain, top_k)` combines prediction and optional explanation in a single call, returning a `SpamPrediction` dataclass with results.

GeneralRiskOverseer Module

The `GeneralRiskOverseer` class implements an alternative classification approach using sliding windows. This method is particularly effective for long texts where spam patterns may be distributed across multiple sections. The classifier processes text in overlapping windows of 256 tokens with a stride of 96 tokens. Each window receives an independent spam probability, and the final classification combines these probabilities using the formula:

$$P(spam) = 1 - \prod(1 - p_i)$$

where p_i represents the spam probability for each window.

SpamService Facade

The `SpamService` class implements the singleton pattern to ensure model loading occurs only once. The class provides a unified interface for classification requests, managing the `SpamChecker` instance lifecycle and caching it for subsequent requests.

Logging Infrastructure

The `Logger` class implements professional logging with three handlers:

- File handler with rotation at 10MB with 5 backup files
- Error handler for ERROR level and above, separate file for critical issues
- Console handler for realtime monitoring during development

The logger employs a singleton pattern with class level caching to prevent duplicate handler registration and keep logging behavior consistent across modules.

3. Database Design

3.1 Stateless Architecture Decision

The Email Spam Detection System implements a completely stateless architecture with no persistent data storage. This design decision aligns with the privacy and simplicity goals from the initial system specification.

3.2 Explanation for No Database

The absence of a database layer provides several advantages:

- **Privacy Protection**: User-submitted email content is never stored, ensuring complete anonymity and eliminating data breach risks
- **Simplified Deployment**: No database server configuration, maintenance, or backup procedures required
- **Reduced Attack Surface**: Elimination of SQL injection vulnerabilities and database-related security concerns
- **Regulatory Compliance**: No data retention regulations to manage since no personal data is stored
- **Horizontal Scalability**: Each application instance is independent with no shared state synchronization required

3.3 In-Memory State Management

The system maintains minimal in-memory state for operational efficiency:

- **Model Cache**: SpamService singleton maintains loaded model instance across requests
- **Logger Cache**: Logger class maintains handler instances to prevent duplicate registration
- **Configuration**: Static configuration loaded from environment variables at application startup

This approach keeps resource usage efficient while maintaining the stateless nature of individual classification requests.

3.4 File-Based Logging

Even though the system does not persist user data, operational logs are maintained in rotating files for debugging and monitoring purposes:

- `app.log`: General application events and information messages
- `errors.log`: Error messages for critical issues and exceptions

Log entries contain no user content, only metadata such as text length, classification results, and system events. This supports debugging while maintaining privacy.

4. Client-Side Implementation

4.1 Web Interface Design

The client-side implementation consists of HTML templates rendered through Jinja2, providing a straightforward user interface for spam detection.

4.2 Template Structure

The application uses three primary templates:

- `home.html` serves as the landing page, providing system overview and navigation to the spam detection form.
- `check_spam.html` implements the main interaction interface with a textarea for email input and dynamic result display. The template conditionally renders classification results based on the show_result flag.

- `404.html` provides user-friendly error messaging when classification fails after maximum retry attempts.

4.3 Form Implementation

The spam detection form accepts `POST` requests with email content in the `email_body` field. The form includes:

- Textarea input field with name attribute `email_body`
- Submit button triggering `POST` request to `/form` endpoint
- Result display section showing classification verdict and keywords

4.4 Resource Optimization

The client-side implementation includes performance optimizations:

- HTTP Link headers for CSS stylesheet preloading
- Material Icons preloading for improved initial render time
- Minimal JavaScript usage for fast page load and browser compatibility

4.5 Security Considerations

The client-side implementation incorporates security controls:

- Automatic HTML escaping through Jinja2 to prevent XSS attacks
- No client side data persistence or localStorage usage
- Form data transmitted via `POST` to prevent URL based information leakage

5. Server-Side Implementation

5.1 Flask Application Structure

The server-side implementation uses `Flask` as the web framework, organized through the application factory pattern. The `create_app` function initializes the `Flask` instance, registers blueprints, and configures the `home` route.

5.2 Blueprint Architecture

The spam detection functionality is encapsulated in a `Flask` Blueprint named `spam_bp`, registered at the root URL prefix. This modular approach supports separation of concerns and allows additional blueprints for future features.

5.3 Request Handling Flow

The `show_form` route implements dual functionality for `GET` and `POST` requests:

- **GET requests** render the empty form template, enabling users to submit email text for classification.
- **POST requests** extract the `email_body` field from form data, invoke `SpamService`.`classify_text` with explanation enabled and `top_k` set to 5, and render results with spam verdict and keywords if applicable.

5.4 Error Handling and Retry Logic

The server implements error handling with an exponential backoff retry mechanism for transient failures:

- Maximum 3 retry attempts for `ConnectionError`, `ConnectionResetError`, `Timeout`, and `RuntimeError` exceptions
- Exponential backoff with base delay of 1 second: attempts wait 1, 2, and 4 seconds
- Comprehensive logging at each retry attempt with exception type and attempt number
- User-friendly 404 error page display when maximum retries are exceeded

5.5 Response Optimization

The `push_resources` `after_request` hook optimizes client-side performance by adding HTTP Link headers for resource preloading. This hook checks for successful HTML responses and adds preload directives for CSS stylesheets and Material Icons fonts, reducing initial page load time.

5.6 Configuration Management

Server-side configuration is centralized in the `config.py` module, which defines:

- `MODEL_DIR`: Path to pretrained BERT model, configurable via `SPAM_MODEL_DIR` environment variable
- `MAX_LENGTH`: Maximum token sequence length, by default 256
- `DEFAULT_TOP_K`: Number of keywords to return in explanations, by default 5

- `STOP_WORDS`: Set of common words filtered from keyword explanations
- `LOG_LEVEL`: Logging verbosity, configurable via `LOG_LEVEL` environment variable

6. Testing and Validation

6.1 Testing Strategy

The system implements testing across multiple dimensions to ensure correctness, robustness, and performance. The testing strategy includes unit tests, integration tests, functional tests, security tests, and performance benchmarks.

6.2 Unit Testing

Text Processing Tests

The `test_text_processing.py` module validates wordpiece merging and filtering logic:

- `test_flush`: Validates the `_flush` helper function for accumulating token scores
- `test_merge_wordpieces_basic`: Verifies BERT subword token reconstruction into complete words
- `test_merge_filters_stop_words_and_short_tokens`: Confirms stop word removal and minimum length enforcement

GeneralRiskOverseer Tests

The `test_gro.py` module validates the sliding window classifier:

- `test_gro_predict_calculation`: Verifies the probabilistic combination formula for window level predictions
- `test_gro_predict_empty_windows`: Ensures graceful handling of empty text inputs
- `test_gro_predict_threshold_equal`: Validates threshold comparison with boundary value

6.3 Integration Testing

API and Service Tests

The `test_api_and_service.py` module validates end-to-end request processing:

- `test_home_get`: Verifies home page accessibility and response status

- `test_form_get`: Confirms form page rendering
- `test_form_post_with_mocked_service`: Validates `POST` request handling with mocked classification service

Edge Case Tests

The `test_edge_cases.py` module validates special scenarios:

- `test_spamservice_singleton`: Confirms singleton pattern implementation for model caching
- `test_show_form_retries_and_success`: Validates retry logic with transient failures followed by success

6.4 Functional Testing

The `test_functional.py` module validates user-facing scenarios:

- `test_get_form_shows_textarea`: Verifies form element presence in `GET` responses
- `test_post_form_not_spam_shows_not_spam_and_no_keywords`: Confirms non spam results omit keyword section
- `test_post_form_spam_shows_keywords_and_user_text`: Validates spam results include keywords and original text
- `test_retry_on_transient_errors`: Confirms exponential backoff retry behavior

6.5 Security Testing

The `test_security.py` module validates security controls:

XSS Prevention Tests

Parametrized tests verify HTML escaping for malicious payloads:

```
- Script injection attempts: <script>alert("XSS")</script>
- Image event handlers: <img src=x onerror=alert(1)>
- Element event attributes: <div onclick=malicious()>
```

All tests confirm that malicious content is properly escaped to HTML entities, preventing execution in the browser.

Input Validation Tests

Parametrized tests validate robustness against various input types:

- Empty strings and whitespace-only inputs
- Very long inputs (50,000 characters)
- Unicode text in Chinese, Arabic, and emojis
- Special characters and control characters
- SQL injection like patterns
- Path traversal attempts

All tests confirm that the system handles these inputs without crashes or exceptions visible to the user.

6.6 Performance Testing

The `test_performance.py` module implements comprehensive performance benchmarks:

- `test_form_get_response_time_fast`: `GET` requests must complete within 100ms
- `test_form_post_response_time_fast`: `POST` requests with classification must complete within 500ms
- `test_form_post_large_input_performance`: 50KB inputs must complete within 1 second
- `test_concurrent_requests_throughput`: Average response time for 10 sequential requests must remain below 200ms
- `test_form_response_size_reasonable`: Response HTML must be under 10KB for non spam results
- `test_repeated_classification_consistent`: Multiple classifications of identical text must produce identical results

All performance tests pass with mocked service to isolate application performance from model inference time.

6.7 Test Coverage Summary

The test suite achieves comprehensive coverage across all major components:

Component	Test Files	Test Cases
Text Processing	`test_text_processing.py`	3

Component	Test Files	Test Cases
GeneralRiskOverseer	`test_gro.py`	3
API and Service	`test_api_and_service.py`	3
Edge Cases	`test_edge_cases.py`	2
Functional Scenarios	`test_functional.py`	4
Security	`test_security.py`	15
Performance	`test_performance.py`	10
Total	7 Files	40 tests

7. Performance Evaluation

7.1 Model Performance Metrics

The system was evaluated on a held out test set with threshold 0.300 for spam classification. The model performs strongly across all metrics:

Metric	Value
Accuracy	0.993270 (99.33%)
F1 Score	0.992095 (99.21%)
Precision	0.994716 (99.47%)
Recall	0.989488 (98.95%)
Specificity	0.996086 (99.61%)
Negative Predictive Value	0.992203 (99.22%)

7.2 Error Analysis

The confusion matrix reveals minimal classification errors:

Error Type	Count / Rate
True Positives (Spam as Spam)	1506
True Negatives (Ham as Ham)	2036
False Positives (Ham as Spam)	8 (FPR: 0.39%)
False Negatives (Spam as Ham)	16 (FNR: 1.05%)

The false positive rate of 0.39% indicates high specificity, so legitimate emails are rarely misclassified. The false negative rate of 1.05% represents a reasonable trade-off, as these cases can be mitigated through user awareness and manual review.

7.3 Application Performance

Performance benchmarks were conducted with mocked classification service to isolate application overhead:

Operation	Response Time
GET /form	< 100ms
POST /form (classification)	< 500ms
Large input (50KB)	< 1000ms
Average (10 requests)	< 200ms

In production, model inference adds approximately 200ms for standard classification and 500ms to 3000ms for IntegratedGradients explanations, depending on text length.

8. Conclusion

8.1 Project Summary

The Email Spam Detection System represents a production ready web application for automated spam classification. The system achieves 99.33% accuracy while maintaining core principles of privacy, transparency, and simplicity.

8.2 Achievement of Objectives

The final system successfully addresses all requirements established in the initial specification:

- **Privacy Protection**: Complete statelessness with no persistent storage of user data
- **Transparency**: IntegratedGradients based keyword explanations for spam classifications
- **Simplicity**: Intuitive web interface requiring no user authentication
- **Performance**: Response times under 500ms for standard classification
- **Reliability**: Exponential backoff retry logic for transient failures
- **Security**: Comprehensive XSS prevention and input validation

8.3 Technical Contributions

The project demonstrates several technical achievements:

- **Machine Learning Implementation**: Successful deployment of BERT based classification with 99.47% precision and 98.95% recall, demonstrating good performance in spam detection.
- **Explainable AI**: Integration of IntegratedGradients attribution method with proper wordpiece merging and stop word filtering, providing users with transparent classification reasoning.
- **Software Engineering Practices**: Comprehensive test coverage with 40 test cases across 7 test modules, professional logging infrastructure with file rotation, and error handling.
- **Alternative Approaches**: Implementation of GeneralRiskOverseer sliding window classifier for long text handling, demonstrating architectural flexibility.

8.4 Testing Validation

The testing strategy validates system correctness across multiple dimensions:

- Unit tests confirm mathematical correctness of probabilistic calculations and text processing
- Integration tests verify end-to-end request processing with proper error handling
- Functional tests validate user scenarios based on use case specifications
- Security tests confirm XSS prevention and input validation robustness
- Performance tests establish response time benchmarks and throughput characteristics

8.5 Deployment Readiness

The system is production ready with the following deployment characteristics:

- **Containerization**: Flask application easily containerized with Docker for cloud deployment
- **Horizontal Scaling**: Stateless architecture enables deployment across multiple instances
- **Monitoring**: Comprehensive logging provides operational visibility
- **Security**: Input validation and XSS prevention protect against common attack vectors
- **Maintainability**: Modular architecture with comprehensive test coverage facilitates updates

8.6 Final Remarks

The Email Spam Detection System applies software engineering principles to a practical machine learning problem. The iterative development process through Prototype 1, Prototype 2, and the final version enabled systematic refinement of functionality and architecture. The project achieves strong classification performance while maintaining user privacy through stateless architecture, provides transparency through explainable AI, and ensures reliability through testing and error handling. The system is ready for deployment and follows standard practices in web application development, machine learning integration, and software quality assurance. This final report documents the implementation, testing validation, and performance evaluation, establishing a foundation for future enhancements and serving as reference documentation for maintenance and extension.