Product Requirements Document (PRD): Guardian API Backend

1. Executive Summary

Guardian API is a high-performance, AI-powered threat detection service that analyzes text content in real-time to identify 14 categories of security threats. Built with FastAPI, it provides a production-ready backend with comprehensive monitoring, distributed rate limiting, and AI enrichment capabilities.

2. Product Overview

2.1 Purpose

Guardian API serves as a security layer for applications that process user-generated content, providing real-time threat detection to prevent phishing, malware distribution, social engineering, and other malicious activities.

2.2 Target Users

Platform Developers: Building chat applications, forums, or content platforms

Security Teams: Monitoring and filtering user-generated content

Compliance Officers: Ensuring content meets regulatory requirements

AI Application Developers: Protecting LLM applications from prompt injection and jailbreaking

3. Core Features

3.1 Threat Detection Engine (classifier.py)

Capabilities:

14 Threat Categories: Detects phishing, social engineering, credential harvesting, financial fraud, malware instructions, code injection, prompt injection, PII exfiltration, privacy violation, toxic content, hate speech, misinformation, self-harm risk, and jailbreak prompting

Multi-Language Support: Optimized patterns for English, Spanish, French, German, and Portuguese

Dynamic Confidence Scoring: Context-aware confidence calculation based on pattern strength, category weight, and text characteristics

Weighted Risk Scoring: 0-100 risk score with diminishing returns for multiple threats in the same category

Technical Implementation:

Regex-based pattern matching with 100+ threat patterns

Language detection using langdetect library

Category-specific weights (e.g., malware_instruction: 0.95, misinformation: 0.60)

Overlap detection to prevent duplicate threat reporting

3.2 AI Enrichment (gemini.py)

Capabilities:

Propaganda/Disinformation Detection: Additional confidence scoring for propaganda and disinformation

AI-Generated Content Detection: Identifies AI-generated text with fallback heuristics

Language Detection: BCP-47 language code identification

Risk Score Adjustment: Increases base risk score by up to 25 points based on AI analysis

Technical Implementation:

Google Gemini API integration (supports both SDK and REST endpoints)

Retry logic with exponential backoff (3 attempts, 1-10s delay)

In-memory caching with 1-hour TTL

Graceful degradation on API failures

Timeout handling (5s connect, 15s read)

3.3 Authentication & Authorization (deps.py)

Capabilities:

Database-Backed API Keys: Supabase integration for API key management

Dual Hash Support: Argon2 (production) and SHA-256 (legacy migration)

Environment Fallback: Allowlist-based authentication when database unavailable

Secure Comparison: Constant-time comparison to prevent timing attacks

Technical Implementation:

Deterministic lookup for SHA-256 keys (fast)

Non-deterministic verification for Argon2 keys (secure but slower)

Retry logic for Supabase queries (3 attempts with exponential backoff)

API key format validation (6-128 chars, alphanumeric + -_.)

3.4 Rate Limiting (rate_limiter.py)

Capabilities:

Dual-Layer Protection: API key-based and IP-based rate limiting

Distributed Rate Limiting: Redis-backed for multi-instance deployments

In-Memory Fallback: Continues operation when Redis unavailable

Sliding Window Algorithm: Accurate rate limiting with Redis sorted sets

Technical Implementation:

Configurable limits (default: 100 req/min per API key, 1000 req/min per IP)

Rate limit headers in responses (X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, Retry-After)

Namespace isolation for test environments

Development mode bypass for localhost

3.5 Health Monitoring (health_monitor.py)

Capabilities:

Dependency Health Checks: Redis, Supabase, Gemini API, and system resources

Concurrent Checks: All health checks run in parallel

Response Time Tracking: Measures latency for each dependency

System Metrics: CPU and memory usage monitoring

Technical Implementation:

Configurable timeout (default: 5 seconds)

Health status with metadata (healthy/degraded/unhealthy)

Graceful handling of unavailable dependencies

Exposed via GET /healthz endpoint

3.6 Logging System (logging_client.py)

Capabilities:

Asynchronous Batch Logging: Non-blocking log writes to Supabase

Structured Log Entries: Correlation ID, trace ID, API key ID, risk score, threats, and request metadata

Queue-Based Processing: Configurable batch size (50) and flush interval (10s)

Retry Logic: Exponential backoff for failed inserts

Technical Implementation:

Background worker task with graceful shutdown

Queue overflow protection (max 10,000 entries)

Dead-letter queue consideration for production

Correlation and trace ID propagation

3.7 Metrics Collection (metrics_collector.py)

Capabilities:

In-Memory Metrics: Request count, error count, latency distribution, risk score distribution

Prometheus Support: Optional Prometheus-formatted metrics

Percentile Calculations: P95 latency tracking

Threat Category Counters: Per-category threat detection counts

Technical Implementation:

Thread-safe with locks

Deque-based storage (last 1000 entries)

Prometheus client integration

Exposed via GET /metrics endpoint

3.8 Alerting System (alerting_system.py)

Capabilities:

Rule-Based Alerts: High error rate, high latency, unhealthy dependencies

Cooldown Periods: Prevents alert spam (default: 5 minutes)

Webhook Notifications: Sends alerts to configured webhook URLs

Alert Resolution Tracking: Marks alerts as resolved

Technical Implementation:

Configurable thresholds (error rate: 5%, latency: 5000ms)

Active alert tracking with triggered count

Async notification sending

Environment context in alert payloads

4. API Endpoints

4.1 Primary Endpoint

POST /v1/analyze

Request:

```
{
  "text": "string (1-100,000 chars)",
  "config": {
    "model_version": "string (optional)",
    "compliance_mode": "strict|moderate|permissive (optional)"
  }
}
```

Response:

```
{
  "request_id": "string",
  "risk_score": 0-100,
  "threats_detected": [
    {
      "category": "string",
      "confidence_score": 0.0-1.0,
      "details": "string"
    }
  ],
  "metadata": {
    "is_ai_generated": boolean,
    "language": "string (BCP-47)",
    "gemini_error": "string (optional)"
  }
}
```

Features:

Input sanitization (XSS, SQL injection, control characters)

Rate limiting with headers

Correlation ID tracking

Async logging

4.2 Monitoring Endpoints

GET /healthz

Returns health status of all dependencies

Status codes: 200 (healthy), 503 (unhealthy)

GET /metrics

Returns JSON metrics summary or Prometheus format

Includes request count, error rate, latency, risk scores

5. Data Models

5.1 Core Models (models.py)

AnalyzeRequest:

text: Input text (1-100,000 chars, UTF-8)

config: Optional configuration

Automatic sanitization on validation

AnalyzeResponse:

request_id: Correlation ID

- risk_score: 0-100 integer
- threats_detected: List of Threat objects
- metadata: AI enrichment results

Threat:

- category: Threat category name
- confidence_score: 0.0-1.0 float
- details: Human-readable description

5.2 Security Features

Input Sanitization:

- HTML tag removal
- JavaScript pattern removal
- SQL injection pattern removal
- HTML entity escaping
- Control character removal
- Whitespace normalization

6. Configuration

6.1 Environment Variables (config.py)

Categories:

General: ENV (development/production)

Authentication: GUARDIAN_API_KEY, GUARDIAN_API_KEYS

Dependencies: GEMINI_API_KEY, SUPABASE_URL, REDIS_URL

Rate Limiting: DEFAULT_RATE_LIMIT_PER_KEY, DEFAULT_RATE_LIMIT_PER_IP

Logging: LOG_LEVEL, LOG_TO_FILE, LOG_FILE_PATH

Health Checks: HEALTH_CHECK_TIMEOUT_SECONDS, enable flags per dependency

Metrics: METRICS_ENABLED, PROMETHEUS_METRICS_ENABLED

Alerting: ALERTING_ENABLED, ALERT_WEBHOOK_URL, thresholds

7. Technical Architecture

7.1 Technology Stack

Framework: FastAPI (async/await)

Language: Python 3.11+

Database: Supabase (PostgreSQL)

Cache/Rate Limiting: Redis

AI: Google Gemini

Monitoring: Prometheus (optional)

Logging: structlog

7.2 Design Patterns

Middleware Pattern: Correlation ID, logging, rate limiting

Singleton Pattern: Metrics collector, alerting system, logging client

Retry Pattern: Exponential backoff for external services

Circuit Breaker: Graceful degradation for Gemini API

Queue Pattern: Async logging with background worker

7.3 Performance Characteristics

Latency: <100ms for pattern matching, <500ms with AI enrichment

Throughput: 1000+ req/s per instance (with Redis)

Scalability: Horizontal scaling with Redis-backed rate limiting

Availability: Graceful degradation when dependencies unavailable

8. Security Considerations

8.1 Input Validation

Text length limits (100,000 chars)

UTF-8 encoding validation

Control character filtering

Null byte rejection

8.2 Authentication

API key format validation

Secure hash storage (Argon2)

Constant-time comparison

Rate limiting per API key

8.3 Output Sanitization

HTML entity escaping

XSS prevention

SQL injection prevention

9. Deployment Requirements

9.1 Infrastructure

Compute: Python 3.11+ runtime

Database: Supabase or PostgreSQL

Cache: Redis 5.0+

External APIs: Google Gemini API access

9.2 Dependencies

See requirements.txt:

```
fastapi==0.115.0

uvicorn[standard]==0.30.6

pydantic==2.8.2

redis==5.0.1

supabase==2.6.0

google-generativeai==0.7.2

langdetect==1.0.9

prometheus-client==0.19.0
```

10. Monitoring & Observability

10.1 Structured Logging

Correlation ID for request tracing

Trace ID for distributed tracing

Context-aware log entries

Log levels: DEBUG, INFO, WARNING, ERROR

10.2 Metrics

Request count by status code

Latency distribution (avg, P95)

Error rate percentage

Risk score distribution

Threat category counts

10.3 Health Checks

Dependency health status

Response time tracking

System resource monitoring

Degraded state detection

10.4 Alerting

High error rate alerts

High latency alerts

Dependency failure alerts

Webhook notifications

11. Testing Strategy

11.1 Test Coverage

Unit tests for each module

Integration tests for API endpoints

Performance tests (load testing)

Security tests (authentication, input validation)

11.2 Test Files

test_api_endpoints.py: API endpoint tests

test_analyze.py: Threat detection tests

test_auth.py: Authentication tests

test_classifier_comprehensive.py: Classifier tests

test_gemini_integration.py: Gemini API tests

test_health.py: Health check tests

test_performance.py: Performance tests

test_supabase_integration.py: Database tests

12. Future Enhancements

12.1 Potential Features

Custom Threat Patterns: User-defined regex patterns

Webhook Callbacks: Real-time threat notifications

Batch Analysis: Analyze multiple texts in one request

Historical Analytics: Trend analysis and reporting

Multi-Model Support: Additional AI models beyond Gemini

Advanced Caching: Distributed cache with Redis Cluster

GraphQL API: Alternative to REST API

Real-time Streaming: WebSocket support for live analysis

12.2 Scalability Improvements

Horizontal Scaling: Load balancer + multiple instances

Database Sharding: Partition logs by time or API key

CDN Integration: Cache static responses

Message Queue: Kafka/RabbitMQ for async processing

13. Success Metrics

13.1 Performance KPIs

- Latency: P95 < 500ms
- Availability: 99.9% uptime
- Throughput: 1000+ req/s per instance
- Error Rate: < 0.1%

13.2 Detection KPIs

- Accuracy: > 95% true positive rate
- False Positive Rate: < 5%
- Coverage: All 14 threat categories
- Language Support: 5 languages

14. Conclusion

Guardian API provides a comprehensive, production-ready backend for real-time threat detection in text content. With its multi-layered detection engine, AI enrichment, and robust infrastructure components (rate limiting, health monitoring, logging, metrics, alerting), it offers a scalable and reliable solution for securing user-generated content across platforms.

The modular architecture allows for easy extension and customization, while the comprehensive monitoring and observability features ensure operational excellence in production environments.