KYB Tool - Feature Specifications Document

Part 4: Testing, Integration, and Implementation Guidelines

Document Information

- **Document Type**: Feature Specifications Document
- **Project**: KYB Tool - Enterprise-Grade Know Your Business Platform
- **Version**: 1.0
- **Date**: January 2025
- **Status**: Final Specification
- **Pages**: Part 4 of 4

---

8. Epic 8: Testing and Quality Assurance Framework

8.1 Epic Overview

Epic Description: Comprehensive testing strategy covering unit tests, integration tests, end-to-end tests, performance tests, and security tests to ensure platform reliability and quality.

Business Value: Reduces production defects by 95%, ensures 99.99% uptime reliability, and maintains customer confidence through consistent quality delivery.

Success Metrics:

- Test coverage: >90% code coverage across all services
- Automated test execution time: <20 minutes for full test suite
- Defect escape rate: <1% of releases require hotfixes

- Performance test pass rate: 100% of SLA requirements met

8.2 Testing Strategy and Requirements

Story 8.1: Automated Testing Pipeline

As a development team
I want comprehensive automated testing at all levels
So that we can deliver high-quality features with confidence

Acceptance Criteria:

```gherkin
Given any code changes are made to the platform
When the CI/CD pipeline runs
Then all tests must pass before deployment
And test results must be visible to all team members
And failed tests must block deployment automatically
And test coverage must not decrease below 90%
And performance regression tests must pass
And security vulnerability scans must show no critical issues

Scenario: Comprehensive test execution
Given a new feature is developed
When tests are executed in the pipeline
Then unit tests run in under 5 minutes
And integration tests complete in under 10 minutes
And end-to-end tests finish in under 15 minutes
And all tests must pass with detailed reporting
And coverage reports are generated automatically
```

Testing Framework Implementation:

python

```python
# Testing framework structure and standards

import pytest
import asyncio
import aiohttp
from unittest.mock import Mock, patch, AsyncMock
from dataclasses import dataclass
from typing import Dict, List, Any, Optional
import json

# Test configuration and fixtures
@pytest.fixture(scope="session")
async def test_app():
    """Create test application instance"""
    from kyb_platform import create_app

    app = create_app(testing=True)
    async with app.test_context():
        yield app

@pytest.fixture(scope="session")
async def test_client(test_app):
    """Create test client for API testing"""
    async with aiohttp.ClientSession() as session:
        yield TestClient(session, test_app)

@pytest.fixture
async def test_database():
    """Create clean test database for each test"""
    from kyb_platform.database import create_test_db, cleanup_test_db

    db = await create_test_db()
    try:
```

```
        yield db
    finally:
        await cleanup_test_db(db)

@pytest.fixture
def mock_external_apis():
    """Mock external API dependencies"""
    with patch('kyb_platform.integrations.business_registry.BusinessRegistryAPI') as mock_registry, \
         patch('kyb_platform.integrations.sanctions.SanctionsAPI') as mock_sanctions, \
         patch('kyb_platform.integrations.website_scraper.WebsiteScraper') as mock_scraper:

        # Configure realistic mock responses
        mock_registry.return_value.verify_business.return_value = {
            'status': 'verified',
            'business_name': 'Test Business LLC',
            'registration_date': '2020-01-15'
        }

        mock_sanctions.return_value.screen_entity.return_value = {
            'matches': [],
            'status': 'clear'
        }

        mock_scraper.return_value.scrape_website.return_value = {
            'content': 'Professional business website content',
            'ssl_certificate': True,
            'contact_info_present': True
        }

        yield {
            'registry': mock_registry,
            'sanctions': mock_sanctions,
            'scraper': mock_scraper
```

```
    }

# Unit Test Examples
class TestBusinessClassificationService:
    """Unit tests for business classification service"""

    @pytest.mark.asyncio
    async def test_classify_business_success(self, mock_external_apis):
        """Test successful business classification"""
        from kyb_platform.services.classification import ClassificationService

        service = ClassificationService()

        business_data = {
            'business_description': 'Restaurant serving Italian cuisine',
            'business_name': 'Mario\'s Italian Kitchen',
            'country': 'US'
        }

        result = await service.classify_business(business_data)

        # Assertions
        assert result.mcc.code == '5812'  # Eating Places
        assert result.mcc.confidence >= 0.85
        assert result.naics.code.startswith('722')  # Food Services
        assert result.processing_time_ms < 2000
        assert len(result.alternative_suggestions) >= 2

    @pytest.mark.asyncio
    async def test_classify_business_with_website(self, mock_external_apis):
        """Test classification enhanced with website analysis"""
        from kyb_platform.services.classification import ClassificationService
```

```
    service = ClassificationService()

    business_data = {
        'business_description': 'Technology services company',
        'website_url': 'https://techcorp.example.com',
        'country': 'US'
    }

    result = await service.classify_business(business_data)

    # Website analysis should improve classification specificity
    assert result.mcc.confidence >= 0.90
    assert result.analysis_details['website_analysis_used'] is True
    assert 'website_content_factors' in result.analysis_details

@pytest.mark.asyncio
async def test_classify_business_invalid_input(self):
    """Test error handling for invalid input"""
    from kyb_platform.services.classification import ClassificationService
    from kyb_platform.exceptions import ValidationError

    service = ClassificationService()

    with pytest.raises(ValidationError) as exc_info:
        await service.classify_business({
            'business_description': ''  # Empty description should fail
        })

    assert 'business_description' in str(exc_info.value)

@pytest.mark.asyncio
async def test_batch_classification(self, mock_external_apis):
    """Test batch classification processing"""
```

```python
        from kyb_platform.services.classification import ClassificationService

        service = ClassificationService()

        businesses = [
            {'id': '1', 'business_description': 'Software development company'},
            {'id': '2', 'business_description': 'Online clothing retailer'},
            {'id': '3', 'business_description': 'Dental practice'}
        ]

        job = await service.classify_batch(businesses)

        assert job.job_id is not None
        assert job.status == 'queued'
        assert job.total_count == 3

        # Simulate job completion
        await service._process_batch_job(job.job_id)

        results = await service.get_batch_results(job.job_id)
        assert len(results) == 3
        assert all(result.mcc.code is not None for result in results)

class TestRiskAssessmentService:
    """Unit tests for risk assessment service"""

    @pytest.mark.asyncio
    async def test_assess_risk_comprehensive(self, test_database, mock_external_apis):
        """Test comprehensive risk assessment"""
        from kyb_platform.services.risk_assessment import RiskAssessmentService
        from kyb_platform.models import Business

        # Create test business
```

```
    business = await Business.create({
        'legal_name': 'TechStart Inc',
        'business_description': 'Early-stage fintech startup',
        'website_url': 'https://techstart.example.com',
        'registration_date': '2024-01-01'  # Very new business
    })

    service = RiskAssessmentService()

    assessment = await service.assess_risk(
        business_id=business.id,
        include_predictions=True
    )

    # New fintech startup should have elevated risk
    assert assessment.overall_score >= 60  # Higher risk for new fintech
    assert assessment.risk_level in ['Medium', 'High']
    assert assessment.operational_risk >= 50  # New business = operational risk
    assert assessment.predictions is not None
    assert assessment.predictions['3_month'].predicted_score is not None
    assert len(assessment.risk_factors) >= 5
    assert len(assessment.recommendations) >= 3

@pytest.mark.asyncio
async def test_assess_risk_established_business(self, test_database, mock_external_apis):
    """Test risk assessment for established business"""
    from kyb_platform.services.risk_assessment import RiskAssessmentService
    from kyb_platform.models import Business

    # Create established business
    business = await Business.create({
        'legal_name': 'Established Corp',
        'business_description': 'Manufacturing company with 20 years history',
```

```
        'website_url': 'https://established-corp.com',
        'registration_date': '2004-03-15'  # Well-established
    })

    service = RiskAssessmentService()
    assessment = await service.assess_risk(business_id=business.id)

    # Established manufacturing should have lower risk
    assert assessment.overall_score <= 40  # Lower risk for established business
    assert assessment.risk_level in ['Low', 'Medium']
    assert assessment.operational_risk <= 30

@pytest.mark.parametrize("industry,expected_min_risk", [
    ("adult_entertainment", 80),
    ("gambling", 75),
    ("cryptocurrency", 70),
    ("restaurant", 40),
    ("software_services", 20)
])
@pytest.mark.asyncio
async def test_industry_specific_risk_scoring(self, industry, expected_min_risk,
                                              test_database, mock_external_apis):
    """Test that risk scores appropriately reflect industry risk"""
    from kyb_platform.services.risk_assessment import RiskAssessmentService
    from kyb_platform.models import Business

    business = await Business.create({
        'legal_name': f'Test {industry.title()} Business',
        'business_description': f'Business in {industry} industry',
        'industry_classification': industry,
        'registration_date': '2022-01-01'
    })
```

```python
        service = RiskAssessmentService()
        assessment = await service.assess_risk(business_id=business.id)

        assert assessment.overall_score >= expected_min_risk

# Integration Test Examples
class TestAPIIntegration:
    """Integration tests for API endpoints"""

    @pytest.mark.asyncio
    async def test_classify_endpoint_integration(self, test_client, mock_external_apis):
        """Test full classification API endpoint"""

        request_data = {
            'business_description': 'Online marketplace for handmade crafts',
            'business_name': 'Artisan Market',
            'website_url': 'https://artisan-market.example.com',
            'include_similar': True
        }

        response = await test_client.post('/api/v1/classify', json=request_data)

        assert response.status == 200

        data = await response.json()
        assert 'classification_id' in data
        assert 'primary_classifications' in data
        assert data['primary_classifications']['mcc']['code'] is not None
        assert data['primary_classifications']['mcc']['confidence'] >= 0.5
        assert len(data['alternative_suggestions']) >= 1
        assert data['analysis_details']['processing_time_ms'] < 5000

    @pytest.mark.asyncio
```

```python
    async def test_risk_assessment_endpoint_integration(self, test_client, test_database, mock_external_ap
        """Test full risk assessment API endpoint"""

        # First create a business
        business_data = {
            'legal_name': 'Integration Test Business',
            'business_description': 'Test business for integration testing'
        }

        create_response = await test_client.post('/api/v1/businesses', json=business_data)
        assert create_response.status == 201

        business = await create_response.json()
        business_id = business['id']

        # Now assess risk
        risk_request = {
            'business_id': business_id,
            'include_predictions': True,
            'risk_tolerance': 'moderate'
        }

        risk_response = await test_client.post('/api/v1/risk/assess', json=risk_request)
        assert risk_response.status == 200

        risk_data = await risk_response.json()
        assert risk_data['overall_score'] >= 1
        assert risk_data['overall_score'] <= 100
        assert risk_data['risk_level'] in ['Low', 'Medium', 'High', 'Critical']
        assert 'predictions' in risk_data
        assert len(risk_data['risk_categories']) >= 5

# Performance Test Examples
```

```python
class TestPerformanceRequirements:
    """Performance tests to ensure SLA compliance"""

    @pytest.mark.asyncio
    @pytest.mark.performance
    async def test_classification_performance_sla(self, test_client, mock_external_apis):
        """Test classification meets SLA requirements"""
        import time

        request_data = {
            'business_description': 'Performance test business description'
        }

        start_time = time.time()
        response = await test_client.post('/api/v1/classify', json=request_data)
        end_time = time.time()

        response_time = end_time - start_time

        assert response.status == 200
        assert response_time < 2.0  # SLA requirement: <2 seconds

        data = await response.json()
        assert data['analysis_details']['processing_time_ms'] < 2000

    @pytest.mark.asyncio
    @pytest.mark.performance
    async def test_concurrent_classification_performance(self, test_client, mock_external_apis):
        """Test system handles concurrent requests within SLA"""
        import asyncio

        async def make_classification_request():
            request_data = {
```

```
            'business_description': f'Concurrent test business {asyncio.current_task().get_name()}'
        }
        return await test_client.post('/api/v1/classify', json=request_data)

    # Create 50 concurrent requests
    tasks = [make_classification_request() for _ in range(50)]

    start_time = time.time()
    responses = await asyncio.gather(*tasks)
    end_time = time.time()

    total_time = end_time - start_time

    # All requests should complete successfully
    assert all(response.status == 200 for response in responses)

    # Average response time should still meet SLA under load
    average_response_time = total_time / len(responses)
    assert average_response_time < 3.0  # Slightly higher under load

    # System should maintain throughput
    throughput = len(responses) / total_time
    assert throughput >= 20  # At least 20 requests per second

# Security Test Examples
class TestSecurityRequirements:
    """Security tests for authentication and authorization"""

    @pytest.mark.asyncio
    async def test_authentication_required(self, test_client):
        """Test that API endpoints require authentication"""

        # Remove authentication header
```

```
    response = await test_client.post('/api/v1/classify',
                                      json={'business_description': 'test'},
                                      skip_auth=True)

    assert response.status == 401
    error_data = await response.json()
    assert error_data['error'] == 'unauthorized'

@pytest.mark.asyncio
async def test_invalid_api_key_rejection(self, test_client):
    """Test that invalid API keys are rejected"""

    response = await test_client.post('/api/v1/classify',
                                      json={'business_description': 'test'},
                                      headers={'Authorization': 'Bearer invalid_key'})

    assert response.status == 401
    error_data = await response.json()
    assert error_data['code'] == 'INVALID_API_KEY'

@pytest.mark.asyncio
async def test_rate_limiting_enforcement(self, test_client):
    """Test that rate limiting is properly enforced"""

    # Make requests up to rate limit
    for _ in range(100):  # Assuming 100/min limit for test key
        response = await test_client.post('/api/v1/classify',
                                          json={'business_description': 'rate limit test'})
        if response.status == 429:
            break

    # Next request should be rate limited
    response = await test_client.post('/api/v1/classify',
```

```
                                json={'business_description': 'should be limited'})

        assert response.status == 429
        assert 'Retry-After' in response.headers

        error_data = await response.json()
        assert error_data['code'] == 'RATE_LIMIT_EXCEEDED'

# Test data factories
class BusinessDataFactory:
    """Factory for creating test business data"""

    @staticmethod
    def create_restaurant_business():
        return {
            'legal_name': 'Mario\'s Italian Kitchen',
            'business_description': 'Family-owned restaurant serving authentic Italian cuisine',
            'website_url': 'https://marios-kitchen.example.com',
            'industry': 'restaurant',
            'registration_date': '2018-03-15'
        }

    @staticmethod
    def create_tech_startup():
        return {
            'legal_name': 'InnovateTech Solutions',
            'business_description': 'AI-powered software solutions for enterprise clients',
            'website_url': 'https://innovatetech.example.com',
            'industry': 'software_services',
            'registration_date': '2023-11-01'
        }

    @staticmethod
```

```
    def create_high_risk_business():
        return {
            'legal_name': 'CryptoGaming LLC',
            'business_description': 'Online gambling platform using cryptocurrency',
            'website_url': 'https://cryptogaming.example.com',
            'industry': 'gambling',
            'registration_date': '2024-01-15'
        }

# Test utilities
class TestClient:
    """Enhanced test client with authentication and utilities"""

    def __init__(self, session: aiohttp.ClientSession, app):
        self.session = session
        self.app = app
        self.base_url = 'http://test.kybtool.com'
        self.api_key = 'test_api_key_123'

    async def post(self, path: str, json: Dict = None, headers: Dict = None, skip_auth: bool = False):
        """Make authenticated POST request"""
        url = f"{self.base_url}{path}"

        request_headers = headers or {}
        if not skip_auth:
            request_headers['Authorization'] = f'Bearer {self.api_key}'

        return await self.session.post(url, json=json, headers=request_headers)

    async def get(self, path: str, params: Dict = None, headers: Dict = None):
        """Make authenticated GET request"""
        url = f"{self.base_url}{path}"
```

```
        request_headers = headers or {}
        request_headers['Authorization'] = f'Bearer {self.api_key}'

        return await self.session.get(url, params=params, headers=request_headers)

# Configuration for different test environments
pytest_plugins = [
    'pytest_asyncio',
    'pytest_mock',
    'pytest_cov'
]

# Pytest configuration
def pytest_configure(config):
    """Configure pytest with custom markers"""
    config.addinivalue_line(
        "markers", "performance: marks tests as performance tests"
    )
    config.addinivalue_line(
        "markers", "integration: marks tests as integration tests"
    )
    config.addinivalue_line(
        "markers", "security: marks tests as security tests"
    )

# Test execution settings
@pytest.fixture(scope="session", autouse=True)
def setup_test_environment():
    """Setup test environment configuration"""
    import os
    os.environ['TESTING'] = 'true'
    os.environ['DATABASE_URL'] = 'postgresql://test:test@localhost/kyb_test'
```

```
os.environ['REDIS_URL'] = 'redis://localhost:6379/1'
os.environ['LOG_LEVEL'] = 'INFO'
```

---

9. Integration Requirements and Patterns

9.1 Third-Party Integration Standards

Story 9.1: Webhook Integration Framework

As a customer with existing systems

I want reliable webhook notifications for all KYB events

So that I can automate downstream processes and maintain data consistency

Acceptance Criteria:

```gherkin
```

Given I have configured webhook endpoints
When KYB events occur (classification complete, risk assessed, etc.)
Then webhooks should be delivered within 30 seconds
And webhook payloads should include all relevant event data
And failed webhook deliveries should be retried with exponential backoff
And webhook signatures should be included for verification
And webhook logs should be available for troubleshooting
And I should be able to test webhook endpoints before activation

Scenario: Successful webhook delivery
Given I have configured a webhook for classification.completed events
When a business classification is completed
Then a POST request is sent to my webhook URL within 30 seconds
And the payload includes the complete classification result
And the request includes an X-KYB-Signature header for verification
And my endpoint responds with 200 OK
And the delivery is marked as successful in logs

Scenario: Webhook delivery failure and retry
Given my webhook endpoint returns a 500 error
When KYB attempts webhook delivery
Then the system retries after 1 minute, then 5 minutes, then 15 minutes
And after 3 failed attempts, the webhook is marked as failed
And I receive an email notification about the failed webhook
And I can manually retry failed webhooks from the dashboard

Webhook Implementation:

python

```python
# Webhook delivery system implementation
import asyncio
import hmac
import hashlib
import json
from datetime import datetime, timedelta
from dataclasses import dataclass
from typing import Dict, List, Optional, Any
from enum import Enum

class WebhookEventType(Enum):
    CLASSIFICATION_COMPLETED = "classification.completed"
    RISK_ASSESSED = "risk.assessed"
    BUSINESS_CREATED = "business.created"
    BUSINESS_UPDATED = "business.updated"
    BATCH_JOB_COMPLETED = "batch.job.completed"
    SANCTIONS_MATCH_FOUND = "sanctions.match.found"

@dataclass
class WebhookEndpoint:
    id: str
    url: str
    secret: str
    events: List[WebhookEventType]
    active: bool = True
    max_retries: int = 3
    timeout_seconds: int = 30

@dataclass
class WebhookDelivery:
    id: str
    endpoint_id: str
    event_type: WebhookEventType
```

```
    payload: Dict[str, Any]
    status: str  # 'pending', 'delivered', 'failed', 'retrying'
    attempts: int = 0
    next_retry: Optional[datetime] = None
    delivered_at: Optional[datetime] = None
    response_status: Optional[int] = None
    response_body: Optional[str] = None
    created_at: datetime = None

class WebhookManager:
    """Manages webhook subscriptions and delivery"""

    def __init__(self, http_client, database):
        self.http_client = http_client
        self.database = database
        self.retry_delays = [60, 300, 900]  # 1min, 5min, 15min

    async def trigger_webhook(self, event_type: WebhookEventType,
                              payload: Dict[str, Any], tenant_id: str):
        """Trigger webhooks for a specific event"""

        # Get active webhook endpoints for this tenant and event type
        endpoints = await self.database.get_webhook_endpoints(
            tenant_id=tenant_id,
            event_type=event_type,
            active=True
        )

        # Create delivery records for each endpoint
        deliveries = []
        for endpoint in endpoints:
            delivery = WebhookDelivery(
                id=self.generate_delivery_id(),
```

```
            endpoint_id=endpoint.id,
            event_type=event_type,
            payload=self.prepare_payload(payload, event_type),
            status='pending',
            created_at=datetime.utcnow()
        )

        # Store delivery record
        await self.database.create_webhook_delivery(delivery)
        deliveries.append(delivery)

    # Schedule immediate delivery attempts
    for delivery in deliveries:
        asyncio.create_task(self.deliver_webhook(delivery))

async def deliver_webhook(self, delivery: WebhookDelivery):
    """Attempt to deliver a webhook"""

    endpoint = await self.database.get_webhook_endpoint(delivery.endpoint_id)

    try:
        # Prepare request
        headers = {
            'Content-Type': 'application/json',
            'X-KYB-Event-Type': delivery.event_type.value,
            'X-KYB-Delivery-ID': delivery.id,
            'X-KYB-Timestamp': str(int(datetime.utcnow().timestamp()))
        }

        # Add signature for verification
        signature = self.generate_signature(
            payload=delivery.payload,
            secret=endpoint.secret,
```

```
            timestamp=headers['X-KYB-Timestamp']
        )
        headers['X-KYB-Signature'] = signature

        # Make HTTP request
        async with self.http_client.post(
            url=endpoint.url,
            json=delivery.payload,
            headers=headers,
            timeout=endpoint.timeout_seconds
        ) as response:

            delivery.response_status = response.status
            delivery.response_body = await response.text()
            delivery.attempts += 1

            if 200 <= response.status < 300:
                # Success
                delivery.status = 'delivered'
                delivery.delivered_at = datetime.utcnow()
            else:
                # HTTP error - schedule retry if attempts remain
                await self.handle_delivery_failure(delivery, endpoint)

    except Exception as e:
        # Network/timeout error - schedule retry
        delivery.attempts += 1
        delivery.response_body = str(e)
        await self.handle_delivery_failure(delivery, endpoint)

    # Update delivery record
    await self.database.update_webhook_delivery(delivery)
```

```
async def handle_delivery_failure(self, delivery: WebhookDelivery,
                                  endpoint: WebhookEndpoint):
    """Handle failed webhook delivery with retry logic"""

    if delivery.attempts < endpoint.max_retries:
        # Schedule retry with exponential backoff
        delay_seconds = self.retry_delays[delivery.attempts - 1]
        delivery.next_retry = datetime.utcnow() + timedelta(seconds=delay_seconds)
        delivery.status = 'retrying'

        # Schedule retry task
        asyncio.get_event_loop().call_later(
            delay_seconds,
            lambda: asyncio.create_task(self.deliver_webhook(delivery))
        )
    else:
        # Max retries exceeded
        delivery.status = 'failed'

        # Notify customer of failed webhook
        await self.send_webhook_failure_notification(delivery, endpoint)

def generate_signature(self, payload: Dict, secret: str, timestamp: str) -> str:
    """Generate HMAC signature for webhook verification"""

    # Create signature payload
    sig_payload = f"{timestamp}.{json.dumps(payload, sort_keys=True)}"

    # Generate HMAC-SHA256 signature
    signature = hmac.new(
        secret.encode(),
        sig_payload.encode(),
        hashlib.sha256
```

```
        ).hexdigest()

        return f"sha256={signature}"

    def prepare_payload(self, payload: Dict, event_type: WebhookEventType) -> Dict:
        """Prepare standardized webhook payload"""

        return {
            'event_type': event_type.value,
            'timestamp': datetime.utcnow().isoformat(),
            'data': payload,
            'version': 'v1'
        }

# Webhook verification utility for customers
class WebhookVerifier:
    """Utility for customers to verify webhook signatures"""

    @staticmethod
    def verify_signature(payload: bytes, signature: str, secret: str,
                         timestamp: str, tolerance: int = 300) -> bool:
        """
        Verify webhook signature

        Args:
            payload: Raw request body bytes
            signature: X-KYB-Signature header value
            secret: Webhook endpoint secret
            timestamp: X-KYB-Timestamp header value
            tolerance: Maximum age of webhook in seconds (default 5min)

        Returns:
            True if signature is valid and timestamp is within tolerance
```

```
        """

        # Check timestamp tolerance
        webhook_time = int(timestamp)
        current_time = int(datetime.utcnow().timestamp())

        if abs(current_time - webhook_time) > tolerance:
            return False

        # Verify signature
        expected_sig = hmac.new(
            secret.encode(),
            f"{timestamp}.{payload.decode()}".encode(),
            hashlib.sha256
        ).hexdigest()

        expected_sig = f"sha256={expected_sig}"

        return hmac.compare_digest(signature, expected_sig)

# Example webhook endpoint for customers
async def kyb_webhook_handler(request):
    """Example webhook handler for customer implementation"""

    # Get headers and payload
    signature = request.headers.get('X-KYB-Signature')
    timestamp = request.headers.get('X-KYB-Timestamp')
    event_type = request.headers.get('X-KYB-Event-Type')

    payload = await request.body()

    # Verify webhook signature
    if not WebhookVerifier.verify_signature(
```

```
        payload=payload,
        signature=signature,
        secret='your_webhook_secret',
        timestamp=timestamp
    ):
        return {'error': 'Invalid signature'}, 401

    # Parse event data
    event_data = json.loads(payload)

    # Handle different event types
    if event_type == 'classification.completed':
        await handle_classification_completed(event_data)
    elif event_type == 'risk.assessed':
        await handle_risk_assessment(event_data)
    elif event_type == 'sanctions.match.found':
        await handle_sanctions_match(event_data)

    return {'status': 'processed'}, 200

async def handle_classification_completed(event_data):
    """Handle classification completed webhook"""
    classification = event_data['data']

    # Update local database with classification results
    await update_merchant_classification(
        merchant_id=classification['business_id'],
        mcc_code=classification['primary_classifications']['mcc']['code'],
        risk_score=classification.get('risk_score')
    )

    # Trigger downstream processes
```

```
    if classification['primary_classifications']['mcc']['confidence'] < 0.8:
        await queue_manual_review(classification['business_id'])
```

9.2 API Integration Patterns

Story 9.2: Standardized Error Handling and Response Formats

As a API consumer

I want consistent error handling and response formats across all endpoints

So that I can build robust error handling in my integration

Acceptance Criteria:

```gherkin
```

Given I make requests to any KYB API endpoint
When an error occurs
Then the error response should follow a consistent format
And error codes should be machine-readable and documented
And error messages should be human-readable and actionable
And the response should include a unique request ID for support
And HTTP status codes should follow REST conventions
And rate limiting information should be included in headers

Scenario: Validation error response
Given I send invalid data to an API endpoint
When the API validates the request
Then I receive a 400 Bad Request status
And the response includes specific field validation errors
And each error includes the field name and description
And the response includes the request ID for troubleshooting

Scenario: Rate limiting error response
Given I exceed my API rate limits
When I make an additional request
Then I receive a 429 Too Many Requests status
And the response includes when I can retry (Retry-After header)
And the response explains the rate limit that was exceeded
And the response includes current usage information

API Response Standards:

typescript

```typescript
// Standardized API response formats

// Success response format
interface SuccessResponse<T> {
  success: true;
  data: T;
  request_id: string;
  timestamp: string;
  processing_time_ms: number;
}

// Error response format
interface ErrorResponse {
  success: false;
  error: {
    code: string;          // Machine-readable error code
    message: string;       // Human-readable error message
    details?: any;         // Additional error details
    validation_errors?: ValidationError[];
  };
  request_id: string;
  timestamp: string;
  documentation_url?: string;
}

interface ValidationError {
  field: string;
  code: string;
  message: string;
  value?: any;
}

// Rate limiting response headers
```

```typescript
interface RateLimitHeaders {
  'X-RateLimit-Limit': string;       // "1000"
  'X-RateLimit-Remaining': string;   // "999"
  'X-RateLimit-Reset': string;       // "1640995200"
  'X-RateLimit-Window': string;      // "3600"
  'Retry-After'?: string;            // "60" (only when rate limited)
}

// Error code constants
enum APIErrorCodes {
  // Authentication errors (401)
  INVALID_API_KEY = 'INVALID_API_KEY',
  EXPIRED_API_KEY = 'EXPIRED_API_KEY',
  MISSING_API_KEY = 'MISSING_API_KEY',

  // Authorization errors (403)
  INSUFFICIENT_PERMISSIONS = 'INSUFFICIENT_PERMISSIONS',
  FEATURE_NOT_AVAILABLE = 'FEATURE_NOT_AVAILABLE',
  PLAN_LIMIT_EXCEEDED = 'PLAN_LIMIT_EXCEEDED',

  // Validation errors (400)
  VALIDATION_FAILED = 'VALIDATION_FAILED',
  INVALID_REQUEST_FORMAT = 'INVALID_REQUEST_FORMAT',
  MISSING_REQUIRED_FIELD = 'MISSING_REQUIRED_FIELD',
  INVALID_FIELD_VALUE = 'INVALID_FIELD_VALUE',

  // Resource errors (404)
  RESOURCE_NOT_FOUND = 'RESOURCE_NOT_FOUND',
  BUSINESS_NOT_FOUND = 'BUSINESS_NOT_FOUND',
  CLASSIFICATION_NOT_FOUND = 'CLASSIFICATION_NOT_FOUND',

  // Rate limiting (429)
  RATE_LIMIT_EXCEEDED = 'RATE_LIMIT_EXCEEDED',
```

```
  CONCURRENT_REQUEST_LIMIT = 'CONCURRENT_REQUEST_LIMIT',

  // Server errors (500+)
  INTERNAL_SERVER_ERROR = 'INTERNAL_SERVER_ERROR',
  SERVICE_UNAVAILABLE = 'SERVICE_UNAVAILABLE',
  EXTERNAL_SERVICE_ERROR = 'EXTERNAL_SERVICE_ERROR',
  MODEL_PREDICTION_ERROR = 'MODEL_PREDICTION_ERROR'
}

// Example error responses
const validationErrorExample = {
  success: false,
  error: {
    code: 'VALIDATION_FAILED',
    message: 'The request contains validation errors',
    validation_errors: [
      {
        field: 'business_description',
        code: 'FIELD_TOO_SHORT',
        message: 'Business description must be at least 10 characters',
        value: 'test'
      },
      {
        field: 'website_url',
        code: 'INVALID_URL_FORMAT',
        message: 'Website URL must be a valid HTTP/HTTPS URL',
        value: 'not-a-url'
      }
    ]
  },
  request_id: 'req_1234567890abcdef',
  timestamp: '2025-01-15T10:30:00Z',
  documentation_url: 'https://docs.kybtool.com/api/errors#validation-failed'
```

```
};

const rateLimitErrorExample = {
  success: false,
  error: {
    code: 'RATE_LIMIT_EXCEEDED',
    message: 'Rate limit exceeded. You have made 1001 requests in the current hour. Please wait 1800 secor
    details: {
      limit: 1000,
      window_seconds: 3600,
      requests_made: 1001,
      reset_time: '2025-01-15T12:00:00Z'
    }
  },
  request_id: 'req_abcdef1234567890',
  timestamp: '2025-01-15T11:30:00Z'
};
```

---

10. Implementation Guidelines and Best Practices

10.1 Code Quality and Standards

Development Standards:

```
yaml
```

```yaml
# Development standards and guidelines

Code_Quality:
  test_coverage: ">90%"
  code_complexity: "<10 cyclomatic complexity"
  documentation: "All public APIs documented"
  type_hints: "Required for Python, TypeScript for JavaScript"

Code_Review:
  required_reviewers: 2
  automated_checks: "linting, testing, security scanning"
  review_checklist: "security, performance, maintainability"

Git_Workflow:
  branching_strategy: "GitFlow"
  commit_messages: "Conventional commits format"
  feature_branches: "Required for all changes"

Security:
  dependency_scanning: "Automated with Snyk/OWASP"
  secret_management: "HashiCorp Vault"
  code_scanning: "Semgrep for security issues"

Performance:
  response_time_targets: "<2s for API calls"
  database_query_optimization: "All queries <100ms"
  caching_strategy: "Redis for frequently accessed data"
```

10.2 Definition of Done

Feature Completion Checklist:

markdown

Definition of Done Checklist

Development
- [ ] Feature implemented according to acceptance criteria
- [ ] Code review completed and approved by 2+ reviewers
- [ ] Unit tests written with >90% coverage
- [ ] Integration tests passing
- [ ] Performance tests meeting SLA requirements
- [ ] Security review completed
- [ ] Documentation updated (API docs, README, etc.)

Quality Assurance
- [ ] Manual testing completed for all user scenarios
- [ ] Cross-browser testing (Chrome, Firefox, Safari, Edge)
- [ ] Mobile responsiveness verified
- [ ] Accessibility standards met (WCAG 2.1 AA)
- [ ] Error handling and edge cases tested
- [ ] Load testing completed for performance-critical features

Compliance and Security
- [ ] Security scan passed with no critical vulnerabilities
- [ ] Compliance requirements verified (SOC 2, PCI DSS, GDPR)
- [ ] Data privacy impact assessed
- [ ] Audit logging implemented where required
- [ ] Error messages don't expose sensitive information

Production Readiness
- [ ] Feature flags implemented for controlled rollout
- [ ] Monitoring and alerting configured
- [ ] Rollback plan prepared and tested
- [ ] Database migrations tested in staging
- [ ] API versioning strategy followed
- [ ] Rate limiting and throttling configured

```
## Documentation and Communication
- [ ] User-facing documentation updated
- [ ] API documentation generated/updated
- [ ] Release notes prepared
- [ ] Customer success team briefed on new features
- [ ] Support team training materials prepared

## Deployment
- [ ] Staging deployment successful
- [ ] Production deployment plan reviewed
- [ ] Database backups verified before deployment
- [ ] Post-deployment verification checklist prepared
- [ ] Stakeholder approval obtained for production release
```

10.3 Monitoring and Observability Requirements

Observability Implementation:

```python
```

```python
# Monitoring and observability implementation
from prometheus_client import Counter, Histogram, Gauge, Info
import logging
import structlog
from datetime import datetime
import json

# Structured logging configuration
structlog.configure(
    processors=[
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.processors.JSONRenderer()
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    wrapper_class=structlog.stdlib.BoundLogger,
    cache_logger_on_first_use=True,
)

# Application metrics
app_requests_total = Counter(
    'kyb_app_requests_total',
    'Total application requests',
    ['method', 'endpoint', 'status_code', 'tenant_id']
)
```

```
app_request_duration = Histogram(
    'kyb_app_request_duration_seconds',
    'Application request duration',
    ['method', 'endpoint'],
    buckets=[0.1, 0.25, 0.5, 1.0, 2.5, 5.0, 10.0]
)

business_classifications_total = Counter(
    'kyb_business_classifications_total',
    'Total business classifications performed',
    ['classification_type', 'confidence_level', 'tenant_id']
)

risk_assessments_total = Counter(
    'kyb_risk_assessments_total',
    'Total risk assessments performed',
    ['risk_level', 'assessment_type', 'tenant_id']
)

ml_model_inference_duration = Histogram(
    'kyb_ml_model_inference_duration_seconds',
    'ML model inference time',
    ['model_name', 'model_version'],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.0]
)

active_webhook_subscriptions = Gauge(
    'kyb_active_webhook_subscriptions_total',
    'Number of active webhook subscriptions',
    ['event_type']
)

class ApplicationLogger:
```

```python
    """Enhanced application logger with structured logging"""

    def __init__(self, component_name: str):
        self.logger = structlog.get_logger(component=component_name)

    def log_api_request(self, method: str, endpoint: str, status_code: int,
                        duration: float, tenant_id: str, request_id: str):
        """Log API request with structured data"""

        # Update metrics
        app_requests_total.labels(
            method=method,
            endpoint=endpoint,
            status_code=status_code,
            tenant_id=tenant_id
        ).inc()

        app_request_duration.labels(
            method=method,
            endpoint=endpoint
        ).observe(duration)

        # Structured log
        self.logger.info(
            "API request processed",
            method=method,
            endpoint=endpoint,
            status_code=status_code,
            duration_ms=int(duration * 1000),
            tenant_id=tenant_id,
            request_id=request_id
        )
```

```python
def log_business_classification(self, classification_result: dict,
                                 tenant_id: str, processing_time: float):
    """Log business classification with metrics"""

    confidence_level = self.categorize_confidence(
        classification_result['primary_classifications']['mcc']['confidence']
    )

    business_classifications_total.labels(
        classification_type='mcc',
        confidence_level=confidence_level,
        tenant_id=tenant_id
    ).inc()

    self.logger.info(
        "Business classification completed",
        classification_id=classification_result['classification_id'],
        mcc_code=classification_result['primary_classifications']['mcc']['code'],
        confidence=classification_result['primary_classifications']['mcc']['confidence'],
        processing_time_ms=int(processing_time * 1000),
        tenant_id=tenant_id
    )

def log_risk_assessment(self, assessment_result: dict, tenant_id: str):
    """Log risk assessment with metrics"""

    risk_assessments_total.labels(
        risk_level=assessment_result['risk_level'],
        assessment_type=assessment_result.get('assessment_type', 'initial'),
        tenant_id=tenant_id
    ).inc()

    self.logger.info(
```

```
            "Risk assessment completed",
            assessment_id=assessment_result['assessment_id'],
            business_id=assessment_result['business_id'],
            overall_score=assessment_result['overall_score'],
            risk_level=assessment_result['risk_level'],
            tenant_id=tenant_id
        )

    def log_error(self, error: Exception, context: dict = None):
        """Log error with full context"""

        self.logger.error(
            "Application error occurred",
            error_type=type(error).__name__,
            error_message=str(error),
            context=context or {},
            exc_info=True
        )

    def categorize_confidence(self, confidence: float) -> str:
        """Categorize confidence score for metrics"""
        if confidence >= 0.9:
            return 'high'
        elif confidence >= 0.7:
            return 'medium'
        else:
            return 'low'

# Health check endpoints
class HealthCheckManager:
    """Manages application health checks"""

    def __init__(self):
```

```
    self.checks = {
        'database': self.check_database,
        'redis': self.check_redis,
        'ml_models': self.check_ml_models,
        'external_apis': self.check_external_apis
    }

async def get_health_status(self) -> dict:
    """Get comprehensive health status"""

    health_status = {
        'status': 'healthy',
        'timestamp': datetime.utcnow().isoformat(),
        'checks': {},
        'version': self.get_application_version()
    }

    # Run all health checks
    for check_name, check_func in self.checks.items():
        try:
            check_result = await check_func()
            health_status['checks'][check_name] = check_result

            if check_result['status'] != 'healthy':
                health_status['status'] = 'degraded'

        except Exception as e:
            health_status['checks'][check_name] = {
                'status': 'unhealthy',
                'error': str(e)
            }
            health_status['status'] = 'unhealthy'
```

```
        return health_status

    async def check_database(self) -> dict:
        """Check database connectivity and performance"""
        try:
            start_time = time.time()
            # Simple query to test database
            result = await self.execute_health_query()
            duration = time.time() - start_time

            return {
                'status': 'healthy',
                'response_time_ms': int(duration * 1000),
                'connection_count': result.get('connection_count'),
                'last_checked': datetime.utcnow().isoformat()
            }
        except Exception as e:
            return {
                'status': 'unhealthy',
                'error': str(e),
                'last_checked': datetime.utcnow().isoformat()
            }
```

This completes the comprehensive Feature Specifications Document covering all aspects from core features through testing, integration patterns, and implementation guidelines. The document provides detailed specifications for:

1. **Core Features**: Business classification, risk assessment, web dashboard, API gateway, compliance screening
2. **Advanced Features**: Analytics, reporting, SDK ecosystem
3. **Quality Assurance**: Comprehensive testing strategy with examples

4. **Integration Standards**: Webhook frameworks, API patterns, error handling
5. **Implementation Guidelines**: Code quality standards, definition of done, monitoring requirements

The specifications are designed to serve as the definitive source of truth for development teams using Cursor, with detailed acceptance criteria, technical implementations, and quality requirements to ensure successful platform delivery.