# GRAVIX TESTING PLAN

## Complete Quality Assurance Specification

### Covers V1 Core + V2 Intelligence Layer, Observability, and Admin System

**Date:** February 2026
**Stack:** FastAPI + pytest (backend) | Next.js + Playwright (frontend) | Supabase (DB)
**Test runner:** `pytest` with `pytest-asyncio` for backend, `Playwright` for E2E
**CI:** GitHub Actions on every PR + nightly full suite

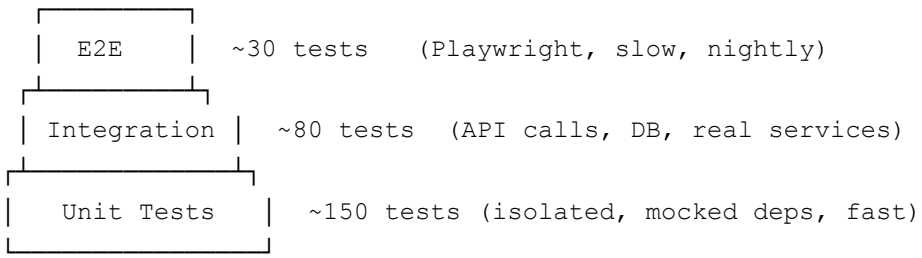
---

# TABLE OF CONTENTS

---

# 1. TESTING ARCHITECTURE

## 1.1 Test Pyramid




| Layer | Count | Runtime | Frequency | What It Catches |
|---|---|---|---|---|
| Unit | ~150 | <30s | Every PR | Logic errors in normalizer, classifier, aggregator, context builder |
| Integration | ~80 | ~3min | Every PR | API contract, DB operations, auth flows, middleware behavior |
| E2E | ~30 | ~8min | Nightly + pre-release | Full user flows, UI rendering, cross-page navigation |
| AI-specific | ~20 | ~5min | Weekly (costs API credits) | Prompt quality, response parsing, knowledge injection impact |
| Performance | ~10 | ~10min | Pre-release | Latency under load, DB query performance, concurrent users |

## 1.2 Test File Structure

```
tests/
├── conftest.py                    # Shared fixtures: test DB, mock user, moc
├── factories.py                   # Test data factories
```

```
│
├── unit/
│   ├── test_normalizer.py             # Substrate normalizer
│   ├── test_classifier.py             # Root cause classifier
│   ├── test_knowledge_context.py      # Context builder (mocked DB)
│   ├── test_aggregator_logic.py       # Aggregation math (mocked DB)
│   ├── test_token_estimator.py        # Token counting
│   ├── test_cost_calculator.py        # API cost estimation
│   └── test_validators.py            # Pydantic schema validation
│
├── integration/
│   ├── test_analyze_endpoint.py       # POST /v1/analyze
│   ├── test_specify_endpoint.py       # POST /v1/specify
│   ├── test_feedback_endpoint.py      # POST /v1/feedback
│   ├── test_admin_endpoints.py        # GET /v1/admin/*
│   ├── test_stats_endpoint.py         # GET /v1/stats/public
│   ├── test_cron_endpoints.py         # POST /v1/cron/*
│   ├── test_auth_flow.py             # Auth middleware + role checks
│   ├── test_request_logger.py         # Middleware logging
│   ├── test_self_learning_pipeline.py # Full feedback→aggregate→inject flow
│   ├── test_observability_pipeline.py # AI log→metrics→dashboard data
│   └── test_case_generation.py        # Auto case library creation
│
├── e2e/
│   ├── test_failure_analysis_flow.py  # Submit analysis → view results → give fe
│   ├── test_spec_engine_flow.py       # Submit spec → view results → export PDF
│   ├── test_auth_flow.py             # Sign in → access tool → sign out
│   ├── test_pricing_page.py          # View pricing → select plan → checkout
│   ├── test_case_library.py          # Browse cases → view detail → run analysis
│   ├── test_history_page.py          # View history → filter → click through
│   ├── test_admin_dashboard.py        # Navigate all admin pages
│   ├── test_feedback_email_flow.py    # Click email link → land on feedback page
│   └── test_cross_linking.py         # Failure → Run Spec → pre-filled form
│
├── ai/
│   ├── test_prompt_quality.py         # Validates AI output structure and releva
│   ├── test_knowledge_injection.py    # With vs without knowledge context
│   ├── test_response_parsing.py       # JSON parse reliability
│   └── test_confidence_calibration.py # Empirical vs estimated confidence
│
├── performance/
│   ├── test_api_latency.py           # Endpoint response times
│   ├── test_db_query_performance.py   # Slow query detection
│   ├── test_concurrent_analyses.py    # Multiple simultaneous AI calls
│   └── test_aggregation_at_scale.py   # Aggregator with 10K+ rows
│
└── security/
```

```
    ├── test_auth_enforcement.py       # Unauthenticated access blocked
    ├── test_admin_gating.py           # Non-admin can't access admin routes
    ├── test_rls_policies.py           # Users can't see other users' data
    ├── test_input_sanitization.py     # XSS, SQL injection attempts
    └── test_rate_limiting.py          # Free tier limits enforced
```

---

# 2. TEST ENVIRONMENT SETUP

## 2.1 conftest.py — Shared Fixtures

```
"""
tests/conftest.py — Shared fixtures for all test modules.
"""

import pytest
import pytest_asyncio
import asyncio
from httpx import AsyncClient, ASGITransport
from uuid import uuid4
from datetime import datetime, timedelta

from api.main import app
from database import db


# ── Database ──

@pytest_asyncio.fixture(autouse=True)
async def clean_db():
    """Truncate test-affected tables before each test."""
    tables = [
        "analysis_feedback", "knowledge_patterns", "ai_engine_logs",
        "api_request_logs", "daily_metrics", "admin_audit_log",
        "case_library", "usage_logs"
    ]
    for table in tables:
        await db.execute(f"TRUNCATE TABLE {table} CASCADE")
    yield
    # No teardown needed — next test truncates


# ── HTTP Client ──
```

```
@pytest_asyncio.fixture
async def client():
    """Async HTTP client for integration tests."""
    transport = ASGITransport(app=app)
    async with AsyncClient(transport=transport, base_url="http://test") as ac:
        yield ac


# ── Auth Fixtures ──

@pytest_asyncio.fixture
async def test_user():
    """Create a regular test user and return (user, auth_headers)."""
    user = await db.insert("users", {
        "id": uuid4(),
        "email": "test@example.com",
        "name": "Test User",
        "role": "user",
        "plan": "pro",
        "created_at": datetime.utcnow(),
    })
    token = create_test_token(user["id"])
    headers = {"Authorization": f"Bearer {token}"}
    return user, headers


@pytest_asyncio.fixture
async def free_user():
    """Free-tier test user."""
    user = await db.insert("users", {
        "id": uuid4(),
        "email": "free@example.com",
        "name": "Free User",
        "role": "user",
        "plan": "free",
        "created_at": datetime.utcnow(),
    })
    token = create_test_token(user["id"])
    headers = {"Authorization": f"Bearer {token}"}
    return user, headers


@pytest_asyncio.fixture
async def admin_user():
    """Admin test user."""
    user = await db.insert("users", {
```

```
        "id": uuid4(),
        "email": "admin@example.com",
        "name": "Admin User",
        "role": "admin",
        "plan": "pro",
        "created_at": datetime.utcnow(),
    })
    token = create_test_token(user["id"])
    headers = {"Authorization": f"Bearer {token}"}
    return user, headers


# ── AI Mock ──

@pytest.fixture
def mock_ai_response():
    """Standard mock AI response for testing without hitting Claude API."""
    return {
        "root_causes": [
            {"rank": 1, "cause": "Inadequate surface preparation",
             "confidence": 0.87, "explanation": "The oxide layer was not disrupte
             "mechanism": "Aluminum oxide creates weak boundary layer."},
            {"rank": 2, "cause": "Moisture contamination",
             "confidence": 0.62, "explanation": "Humidity during cure.",
             "mechanism": "CA polymerizes too fast with surface moisture."},
            {"rank": 3, "cause": "Material incompatibility",
             "confidence": 0.35, "explanation": "ABS plasticizer migration.",
             "mechanism": "Plasticizer weakens interface."},
        ],
        "contributing_factors": ["High ambient humidity", "No drying step"],
        "immediate_actions": ["Abrade with 180-grit", "Degrease with acetone"],
        "long_term_solutions": ["Implement water break test", "Add primer step"],
        "prevention_plan": ["Add surface prep SOP", "Incoming material inspection
        "confidence_score": 0.87,
        "processing_time_ms": 4200,
    }


# ── Test Data Factories ──

@pytest_asyncio.fixture
async def sample_analysis(test_user):
    """Create a completed failure analysis in the DB."""
    user, _ = test_user
    analysis = await db.insert("failure_analyses", {
        "id": uuid4(),
        "user_id": user["id"],
```

```
        "material_category": "adhesive",
        "material_subcategory": "cyanoacrylate",
        "failure_mode": "debonding",
        "failure_description": "CA bond failed after 2 weeks",
        "substrate_a": "Aluminum 6061",
        "substrate_b": "ABS plastic",
        "substrate_a_normalized": "aluminum_6061",
        "substrate_b_normalized": "abs",
        "root_cause_category": "surface_preparation",
        "industry": "automotive",
        "production_impact": "quality_hold",
        "temperature_range": "-20C to 80C",
        "humidity": "65%",
        "root_causes": [
            {"rank": 1, "cause": "Surface preparation", "confidence": 0.87},
            {"rank": 2, "cause": "Moisture contamination", "confidence": 0.62},
        ],
        "status": "completed",
        "created_at": datetime.utcnow() - timedelta(days=2),
    })
    return analysis


@pytest_asyncio.fixture
async def sample_feedback(test_user, sample_analysis):
    """Create a feedback entry for the sample analysis."""
    user, _ = test_user
    feedback = await db.insert("analysis_feedback", {
        "id": uuid4(),
        "user_id": user["id"],
        "analysis_id": sample_analysis["id"],
        "was_helpful": True,
        "root_cause_confirmed": 1,
        "outcome": "resolved",
        "what_worked": "Switched from IPA wipe to 180-grit abrasion + acetone deg
        "what_didnt_work": "IPA wipe alone",
        "time_to_resolution": "3 days",
        "estimated_cost_saved": 5000.00,
        "feedback_source": "in_app",
        "created_at": datetime.utcnow(),
    })
    return feedback


def create_test_token(user_id):
    """Generate a JWT for test authentication."""
    from auth import create_access_token
```

```
    return create_access_token({"sub": str(user_id)})
```

## 2.2 factories.py — Bulk Test Data

```
"""
tests/factories.py — Generate bulk test data for performance and aggregation test
"""

from uuid import uuid4
from datetime import datetime, timedelta
import random


SUBSTRATES = ["aluminum_6061", "abs", "polycarbonate", "stainless_304", "glass",
              "nylon_general", "copper", "hdpe", "carbon_fiber", "pmma"]
FAILURE_MODES = ["debonding", "cracking", "blooming", "creep", "discoloration"]
OUTCOMES = ["resolved", "partially_resolved", "not_resolved", "different_cause"]
ROOT_CAUSE_CATS = ["surface_preparation", "material_compatibility", "cure_conditi
                   "environmental", "application_process", "design"]


async def create_bulk_analyses(db, user_id, count=100):
    """Insert N completed failure analyses with normalized fields."""
    analyses = []
    for i in range(count):
        sub_a = random.choice(SUBSTRATES)
        sub_b = random.choice([s for s in SUBSTRATES if s != sub_a])
        a = await db.insert("failure_analyses", {
            "id": uuid4(),
            "user_id": user_id,
            "material_category": "adhesive",
            "material_subcategory": random.choice(["cyanoacrylate", "epoxy", "pol
            "failure_mode": random.choice(FAILURE_MODES),
            "failure_description": f"Test failure case {i}",
            "substrate_a": sub_a,
            "substrate_b": sub_b,
            "substrate_a_normalized": sub_a,
            "substrate_b_normalized": sub_b,
            "root_cause_category": random.choice(ROOT_CAUSE_CATS),
            "industry": random.choice(["automotive", "aerospace", "electronics"])
            "root_causes": [
                {"rank": 1, "cause": "Surface preparation issue", "confidence": r
                {"rank": 2, "cause": "Cure problem", "confidence": random.uniform
            ],
            "status": "completed",
```

```
            "created_at": datetime.utcnow() - timedelta(days=random.randint(1, 90
        })
        analyses.append(a)
    return analyses


async def create_bulk_feedback(db, user_id, analyses, feedback_rate=0.4):
    """Insert feedback for a fraction of analyses."""
    feedbacks = []
    for a in analyses:
        if random.random() < feedback_rate:
            f = await db.insert("analysis_feedback", {
                "id": uuid4(),
                "user_id": user_id,
                "analysis_id": a["id"],
                "was_helpful": random.choice([True, True, True, False]),
                "root_cause_confirmed": random.choice([0, 1, 1, 1, 2]),
                "outcome": random.choice(OUTCOMES),
                "what_worked": random.choice([
                    "Abrasion + acetone degrease",
                    "Switched to toughened epoxy",
                    "Added primer step",
                    "Increased cure temperature",
                    None,
                ]),
                "what_didnt_work": random.choice(["IPA wipe alone", "Extended cur
                "feedback_source": random.choice(["in_app", "email"]),
                "created_at": datetime.utcnow() - timedelta(days=random.randint(0
            })
            feedbacks.append(f)
    return feedbacks
```

---

# 3. UNIT TESTS — BACKEND SERVICES

## 3.1 Substrate Normalizer

**File:** `tests/unit/test_normalizer.py`

```
"""Tests for api/utils/normalizer.py"""

import pytest
from utils.normalizer import normalize_substrate, make_pair_key
```

```
class TestNormalizeSubstrate:

    # ── Exact matches ──

    def test_aluminum_6061(self):
        assert normalize_substrate("Aluminum 6061-T6") == "aluminum_6061"

    def test_aluminum_british_spelling(self):
        assert normalize_substrate("Aluminium 6061") == "aluminum_6061"

    def test_aluminum_generic(self):
        assert normalize_substrate("aluminum") == "aluminum_general"

    def test_stainless_304(self):
        assert normalize_substrate("Stainless Steel 304") == "stainless_304"

    def test_stainless_316(self):
        assert normalize_substrate("316 stainless steel") == "stainless_316"

    def test_abs(self):
        assert normalize_substrate("ABS") == "abs"

    def test_polycarbonate_full(self):
        assert normalize_substrate("Polycarbonate") == "polycarbonate"

    def test_polycarbonate_short(self):
        assert normalize_substrate("Polycarb") == "polycarbonate"

    def test_ptfe_teflon(self):
        assert normalize_substrate("Teflon") == "ptfe"

    def test_acetal_delrin(self):
        assert normalize_substrate("Delrin") == "acetal"

    # ── Case insensitivity ──

    def test_lowercase(self):
        assert normalize_substrate("abs") == "abs"

    def test_uppercase(self):
        assert normalize_substrate("ABS") == "abs"

    def test_mixed_case(self):
        assert normalize_substrate("Aluminum 6061") == "aluminum_6061"

    # ── Edge cases ──
```

```
    def test_empty_string(self):
        assert normalize_substrate("") == "unknown"

    def test_none(self):
        assert normalize_substrate(None) == "unknown"

    def test_whitespace(self):
        assert normalize_substrate("  aluminum  6061  ") == "aluminum_6061"

    def test_unknown_material(self):
        result = normalize_substrate("Unobtanium Grade 7")
        assert result == "unobtanium_grade_7"  # slugified fallback

    def test_special_characters(self):
        result = normalize_substrate("Aluminum (6061-T6)")
        assert result == "aluminum_6061"


class TestMakePairKey:

    def test_alphabetical_order(self):
        assert make_pair_key("ABS", "Aluminum 6061") == "abs::aluminum_6061"

    def test_reverse_order_same_result(self):
        key1 = make_pair_key("ABS", "Aluminum 6061")
        key2 = make_pair_key("Aluminum 6061", "ABS")
        assert key1 == key2

    def test_same_material(self):
        assert make_pair_key("ABS", "ABS") == "abs::abs"

    def test_normalizes_both(self):
        key = make_pair_key("Teflon", "Delrin")
        assert key == "acetal::ptfe"
```

## 3.2 Root Cause Classifier

**File:** `tests/unit/test_classifier.py`

```
"""Tests for api/utils/classifier.py"""

import pytest
from utils.classifier import classify_root_cause
```

```
class TestClassifyRootCause:

    def test_surface_prep(self):
        causes = [{"cause": "Surface preparation inadequate", "explanation": "Oxi
        assert classify_root_cause(causes) == "surface_preparation"

    def test_contamination_maps_to_surface(self):
        causes = [{"cause": "Contamination present", "explanation": "Mold release
        assert classify_root_cause(causes) == "surface_preparation"

    def test_material_compatibility(self):
        causes = [{"cause": "Material incompatibility", "explanation": "Plasticiz
        assert classify_root_cause(causes) == "material_compatibility"

    def test_cure_conditions(self):
        causes = [{"cause": "Under-cure", "explanation": "Mix ratio was incorrect
        assert classify_root_cause(causes) == "cure_conditions"

    def test_environmental(self):
        causes = [{"cause": "Thermal cycling stress", "explanation": "Temperature
        assert classify_root_cause(causes) == "environmental"

    def test_application_process(self):
        causes = [{"cause": "Bondline too thick", "explanation": "Excess adhesive
        assert classify_root_cause(causes) == "application_process"

    def test_design(self):
        causes = [{"cause": "Peel stress at joint edge", "explanation": "Joint de
        assert classify_root_cause(causes) == "design"

    def test_empty_causes_returns_unknown(self):
        assert classify_root_cause([]) == "unknown"

    def test_none_causes_returns_unknown(self):
        assert classify_root_cause(None) == "unknown"

    def test_ambiguous_defaults_to_highest_score(self):
        causes = [{"cause": "Surface contamination in humid environment",
                   "explanation": "Moisture and grease on substrate surface duri
        result = classify_root_cause(causes)
        # "surface" keywords should outnumber "cure" keywords
        assert result == "surface_preparation"

    def test_string_format_input(self):
        causes = ["Surface preparation was inadequate"]
        assert classify_root_cause(causes) == "surface_preparation"
```

## 3.3 Knowledge Context Builder

**File:** `tests/unit/test_knowledge_context.py`

```
"""Tests for api/services/knowledge_context.py — mocked DB."""

import pytest
from unittest.mock import AsyncMock, patch
import json


class TestBuildKnowledgeContext:

    @pytest.mark.asyncio
    @patch("services.knowledge_context.db")
    async def test_no_data_returns_fallback(self, mock_db):
        from services.knowledge_context import build_knowledge_context
        mock_db.fetch_one = AsyncMock(return_value=None)
        mock_db.fetch_all = AsyncMock(return_value=[])

        result = await build_knowledge_context({
            "substrate_a": "Aluminum", "substrate_b": "ABS", "failure_mode": "deb
        })

        assert "No prior cases" in result
        assert "domain expertise only" in result

    @pytest.mark.asyncio
    @patch("services.knowledge_context.db")
    async def test_substrate_pair_data_injected(self, mock_db):
        from services.knowledge_context import build_knowledge_context

        mock_db.fetch_one = AsyncMock(side_effect=[
            # substrate pair pattern
            {"total_cases": 23, "cases_with_feedback": 14, "resolution_rate": 0.8
             "top_root_causes": json.dumps([
                 {"cause": "Surface prep", "confirmed_count": 9, "frequency": 0.6
             ]),
             "effective_solutions": json.dumps([
                 {"solution": "Abrasion + acetone", "tried": 9, "resolved": 7, "s
             ]),
             "ineffective_solutions": json.dumps([
                 {"solution": "IPA only", "reports": 3}
             ])},
            # failure mode pattern
```

```
        None,
    ])
    mock_db.fetch_all = AsyncMock(return_value=[])

    result = await build_knowledge_context({
        "substrate_a": "Aluminum", "substrate_b": "ABS", "failure_mode": "deb
    })

    assert "23 previous analyses" in result
    assert "Surface prep" in result
    assert "64%" in result
    assert "Abrasion + acetone" in result
    assert "78%" in result
    assert "IPA only" in result

@pytest.mark.asyncio
@patch("services.knowledge_context.db")
async def test_fewer_than_3_cases_excluded(self, mock_db):
    from services.knowledge_context import build_knowledge_context

    mock_db.fetch_one = AsyncMock(return_value={"total_cases": 2, "cases_with
    mock_db.fetch_all = AsyncMock(return_value=[])

    result = await build_knowledge_context({
        "substrate_a": "Aluminum", "substrate_b": "ABS", "failure_mode": "deb
    })

    assert "No prior cases" in result

@pytest.mark.asyncio
@patch("services.knowledge_context.db")
async def test_similar_resolved_cases_included(self, mock_db):
    from services.knowledge_context import build_knowledge_context

    mock_db.fetch_one = AsyncMock(return_value=None)
    mock_db.fetch_all = AsyncMock(return_value=[
        {"top_cause": "Surface prep", "outcome": "resolved", "what_worked": "
        {"top_cause": "Surface prep", "outcome": "resolved", "what_worked": "
    ])

    result = await build_knowledge_context({
        "substrate_a": "Aluminum", "substrate_b": "ABS", "failure_mode": "deb
    })

    assert "2 similar resolved cases" in result
    assert "Abrasion + solvent" in result
```

## 3.4 Token Estimator & Cost Calculator

**File:** `tests/unit/test_token_estimator.py`

```
"""Tests for token estimation and cost calculation."""

from services.ai_telemetry import estimate_tokens, PRICING


class TestEstimateTokens:

    def test_empty_string(self):
        assert estimate_tokens("") == 1  # min 1

    def test_short_string(self):
        assert estimate_tokens("hello") == 1

    def test_normal_text(self):
        text = "This is a normal sentence with about twenty tokens in it roughly.
        result = estimate_tokens(text)
        assert 10 < result < 25  # ~16 tokens, estimator should be in range

    def test_long_text(self):
        text = "word " * 1000  # ~5000 chars = ~1250 tokens
        result = estimate_tokens(text)
        assert 1000 < result < 1500


class TestCostCalculation:

    def test_pricing_dict_has_expected_model(self):
        assert "claude-sonnet-4-20250514" in PRICING

    def test_cost_sanity_check(self):
        # 1000 input tokens + 500 output tokens on Sonnet
        pricing = PRICING["claude-sonnet-4-20250514"]
        cost = (1000 * pricing["input"] + 500 * pricing["output"]) / 1_000_000
        assert 0.001 < cost < 0.05  # should be a few cents
```

## 3.5 Pydantic Schema Validation

**File:** `tests/unit/test_validators.py`

```
"""Tests for Pydantic request/response schemas."""
```

```
import pytest
from pydantic import ValidationError
from schemas.analysis import AnalysisRequest
from routers.feedback import FeedbackCreate


class TestAnalysisRequest:

    def test_valid_minimal_request(self):
        req = AnalysisRequest(
            failure_description="Bond failed",
            substrate_a="Aluminum",
            substrate_b="ABS",
            failure_mode="debonding",
        )
        assert req.substrate_a == "Aluminum"

    def test_optional_fields_default_none(self):
        req = AnalysisRequest(
            failure_description="Bond failed",
            substrate_a="Al", substrate_b="ABS", failure_mode="debonding",
        )
        assert req.industry is None
        assert req.production_impact is None

    def test_industry_accepted(self):
        req = AnalysisRequest(
            failure_description="x", substrate_a="Al", substrate_b="ABS",
            failure_mode="debonding", industry="automotive",
        )
        assert req.industry == "automotive"


class TestFeedbackCreate:

    def test_valid_outcome(self):
        fb = FeedbackCreate(analysis_id="550e8400-e29b-41d4-a716-446655440000", o
        assert fb.outcome == "resolved"

    def test_invalid_outcome_rejected(self):
        with pytest.raises(ValidationError):
            FeedbackCreate(analysis_id="550e8400-e29b-41d4-a716-446655440000", ou

    def test_root_cause_confirmed_range(self):
        fb = FeedbackCreate(analysis_id="550e8400-e29b-41d4-a716-446655440000", r
        assert fb.root_cause_confirmed == 1
```

```
    def test_root_cause_confirmed_out_of_range(self):
        with pytest.raises(ValidationError):
            FeedbackCreate(analysis_id="550e8400-e29b-41d4-a716-446655440000", ro

    def test_requires_analysis_or_spec(self):
        # Validation at endpoint level, not schema — schema allows both None
        fb = FeedbackCreate()
        assert fb.analysis_id is None
```

---

# 4. UNIT TESTS — UTILITIES

## 4.1 Aggregator Math

**File:** `tests/unit/test_aggregator_logic.py`

```
"""Test the computational logic of the aggregator (extracted for unit testing)."""

from collections import Counter


class TestResolutionRate:

    def test_all_resolved(self):
        resolved = 10
        total = 10
        rate = resolved / total
        assert rate == 1.0

    def test_none_resolved(self):
        rate = 0 / 10
        assert rate == 0.0

    def test_no_feedback(self):
        # Should be None, not 0
        total_with_feedback = 0
        rate = None if total_with_feedback == 0 else 0
        assert rate is None


class TestConfidenceLevel:

    def test_high_confidence(self):
```

```
        total = 25
        feedback = 12
        level = "high" if total >= 20 and feedback >= 10 else "medium" if total >
        assert level == "high"

    def test_medium_confidence(self):
        total = 8
        feedback = 4
        level = "high" if total >= 20 and feedback >= 10 else "medium" if total >
        assert level == "medium"

    def test_low_confidence(self):
        total = 3
        feedback = 1
        level = "high" if total >= 20 and feedback >= 10 else "medium" if total >
        assert level == "low"

    def test_many_cases_but_no_feedback_is_low(self):
        total = 50
        feedback = 1
        level = "high" if total >= 20 and feedback >= 10 else "medium" if total >
        assert level == "low"


class TestRootCauseDistribution:

    def test_single_cause(self):
        counter = Counter({"Surface prep": 5})
        total = sum(counter.values())
        dist = [{"cause": c, "frequency": round(n / total, 2)} for c, n in counte
        assert dist[0]["frequency"] == 1.0

    def test_multiple_causes(self):
        counter = Counter({"Surface prep": 6, "Cure": 3, "Environment": 1})
        total = sum(counter.values())
        dist = [{"cause": c, "frequency": round(n / total, 2)} for c, n in counte
        assert dist[0]["cause"] == "Surface prep"
        assert dist[0]["frequency"] == 0.6
        assert dist[1]["frequency"] == 0.3
```

---

# 5. INTEGRATION TESTS — API ENDPOINTS

## 5.1 Failure Analysis Endpoint

**File:** `tests/integration/test_analyze_endpoint.py`

```
"""Integration tests for POST /v1/analyze"""

import pytest
from unittest.mock import patch, AsyncMock


class TestAnalyzeEndpoint:

    @pytest.mark.asyncio
    async def test_unauthenticated_rejected(self, client):
        res = await client.post("/v1/analyze", json={
            "failure_description": "Bond failed",
            "substrate_a": "Aluminum", "substrate_b": "ABS",
            "failure_mode": "debonding",
        })
        assert res.status_code == 401

    @pytest.mark.asyncio
    @patch("services.ai_engine.AIEngine.analyze_failure")
    async def test_successful_analysis(self, mock_ai, client, test_user, mock_ai_
        _, headers = test_user
        mock_ai.return_value = mock_ai_response

        res = await client.post("/v1/analyze", json={
            "failure_description": "CA bond failed after 2 weeks",
            "substrate_a": "Aluminum 6061",
            "substrate_b": "ABS plastic",
            "failure_mode": "debonding",
            "industry": "automotive",
            "production_impact": "quality_hold",
        }, headers=headers)

        assert res.status_code == 200
        data = res.json()
        assert "root_causes" in data
        assert len(data["root_causes"]) >= 1
        assert "confidence_score" in data

    @pytest.mark.asyncio
    @patch("services.ai_engine.AIEngine.analyze_failure")
    async def test_normalized_substrates_saved(self, mock_ai, client, test_user,
        _, headers = test_user
        mock_ai.return_value = mock_ai_response
```

```
    await client.post("/v1/analyze", json={
        "failure_description": "Bond failed",
        "substrate_a": "Aluminum 6061",
        "substrate_b": "ABS",
        "failure_mode": "debonding",
    }, headers=headers)

    # Check DB directly
    from database import db
    analysis = await db.fetch_one(
        "SELECT substrate_a_normalized, substrate_b_normalized, root_cause_ca
    )
    assert analysis["substrate_a_normalized"] == "aluminum_6061"
    assert analysis["substrate_b_normalized"] == "abs"
    assert analysis["root_cause_category"] in [
        "surface_preparation", "material_compatibility", "application_process
        "cure_conditions", "environmental", "design", "unknown"
    ]

@pytest.mark.asyncio
async def test_missing_required_fields(self, client, test_user):
    _, headers = test_user
    res = await client.post("/v1/analyze", json={
        "failure_description": "Bond failed",
        # missing substrates and failure_mode
    }, headers=headers)
    assert res.status_code == 422

@pytest.mark.asyncio
@patch("services.ai_engine.AIEngine.analyze_failure")
async def test_free_tier_limit_enforced(self, mock_ai, client, free_user, moc
    _, headers = free_user
    mock_ai.return_value = mock_ai_response

    # Submit 5 analyses (free tier limit)
    for i in range(5):
        res = await client.post("/v1/analyze", json={
            "failure_description": f"Failure {i}",
            "substrate_a": "Al", "substrate_b": "ABS", "failure_mode": "debon
        }, headers=headers)
        assert res.status_code == 200

    # 6th should be blocked
    res = await client.post("/v1/analyze", json={
        "failure_description": "One too many",
        "substrate_a": "Al", "substrate_b": "ABS", "failure_mode": "debonding
    }, headers=headers)
```

```
        assert res.status_code == 429

    @pytest.mark.asyncio
    @patch("services.ai_engine.AIEngine.analyze_failure")
    async def test_ai_engine_log_created(self, mock_ai, client, test_user, mock_a
        _, headers = test_user
        mock_ai.return_value = mock_ai_response

        await client.post("/v1/analyze", json={
            "failure_description": "Bond failed",
            "substrate_a": "Al", "substrate_b": "ABS", "failure_mode": "debonding
        }, headers=headers)

        from database import db
        log = await db.fetch_one("SELECT * FROM ai_engine_logs ORDER BY created_a
        assert log is not None
        assert log["request_type"] == "failure_analysis"
        assert log["latency_ms"] > 0
        assert log["total_tokens"] > 0
```

## 5.2 Feedback Endpoint

**File:** `tests/integration/test_feedback_endpoint.py`

```
"""Integration tests for POST /v1/feedback"""

import pytest


class TestFeedbackEndpoint:

    @pytest.mark.asyncio
    async def test_submit_quick_feedback(self, client, test_user, sample_analysis
        _, headers = test_user
        res = await client.post("/v1/feedback", json={
            "analysis_id": str(sample_analysis["id"]),
            "was_helpful": True,
        }, headers=headers)

        assert res.status_code == 201
        data = res.json()
        assert data["message"] == "Thank you. Your feedback improves future analy

    @pytest.mark.asyncio
    async def test_submit_full_feedback(self, client, test_user, sample_analysis)
```

```
    _, headers = test_user
    res = await client.post("/v1/feedback", json={
        "analysis_id": str(sample_analysis["id"]),
        "was_helpful": True,
        "root_cause_confirmed": 1,
        "outcome": "resolved",
        "what_worked": "Abrasion + acetone",
        "what_didnt_work": "IPA only",
        "time_to_resolution": "3 days",
        "estimated_cost_saved": 5000.00,
    }, headers=headers)

    assert res.status_code == 201

@pytest.mark.asyncio
async def test_upsert_existing_feedback(self, client, test_user, sample_analy
    _, headers = test_user

    # First submission
    await client.post("/v1/feedback", json={
        "analysis_id": str(sample_analysis["id"]),
        "was_helpful": True,
    }, headers=headers)

    # Update with outcome
    res = await client.post("/v1/feedback", json={
        "analysis_id": str(sample_analysis["id"]),
        "was_helpful": True,
        "outcome": "resolved",
    }, headers=headers)

    assert res.status_code == 201

    # Verify only 1 feedback row exists
    from database import db
    count = await db.fetch_one(
        "SELECT COUNT(*) as cnt FROM analysis_feedback WHERE analysis_id = $1
        sample_analysis["id"])
    assert count["cnt"] == 1

@pytest.mark.asyncio
async def test_cannot_feedback_other_users_analysis(self, client, admin_user,
    # sample_analysis belongs to test_user, not admin_user
    _, headers = admin_user
    res = await client.post("/v1/feedback", json={
        "analysis_id": str(sample_analysis["id"]),
        "was_helpful": True,
```

```
    }, headers=headers)

    assert res.status_code == 404

@pytest.mark.asyncio
async def test_must_provide_analysis_or_spec(self, client, test_user):
    _, headers = test_user
    res = await client.post("/v1/feedback", json={
        "was_helpful": True,
    }, headers=headers)

    assert res.status_code == 400

@pytest.mark.asyncio
async def test_pending_feedback_list(self, client, test_user, sample_analysis
    _, headers = test_user
    res = await client.get("/v1/feedback/pending/list", headers=headers)
    assert res.status_code == 200
    data = res.json()
    assert isinstance(data, list)

@pytest.mark.asyncio
async def test_cases_improved_count(self, client, test_user, sample_analysis)
    _, headers = test_user

    # Create a second analysis with same substrates
    from database import db
    from uuid import uuid4
    user, _ = test_user
    await db.insert("failure_analyses", {
        "id": uuid4(), "user_id": user["id"],
        "material_category": "adhesive", "material_subcategory": "cyanoacryla
        "failure_mode": "debonding", "failure_description": "Another failure"
        "substrate_a": "Aluminum 6061", "substrate_b": "ABS plastic",
        "substrate_a_normalized": "aluminum_6061", "substrate_b_normalized":
        "status": "completed",
    })

    res = await client.post("/v1/feedback", json={
        "analysis_id": str(sample_analysis["id"]),
        "outcome": "resolved",
    }, headers=headers)

    data = res.json()
    assert data["cases_improved"] >= 1
```

## 5.3 Admin Endpoints

**File:** `tests/integration/test_admin_endpoints.py`

```
"""Integration tests for /v1/admin/* endpoints."""

import pytest


class TestAdminGating:

    @pytest.mark.asyncio
    async def test_regular_user_blocked(self, client, test_user):
        _, headers = test_user
        res = await client.get("/v1/admin/overview", headers=headers)
        assert res.status_code == 403

    @pytest.mark.asyncio
    async def test_unauthenticated_blocked(self, client):
        res = await client.get("/v1/admin/overview")
        assert res.status_code == 401

    @pytest.mark.asyncio
    async def test_admin_allowed(self, client, admin_user):
        _, headers = admin_user
        res = await client.get("/v1/admin/overview", headers=headers)
        assert res.status_code == 200


class TestAdminOverview:

    @pytest.mark.asyncio
    async def test_returns_expected_shape(self, client, admin_user):
        _, headers = admin_user
        res = await client.get("/v1/admin/overview", headers=headers)
        data = res.json()

        assert "today" in data
        assert "totals" in data
        assert "mrr" in data
        assert "ai_cost_this_month" in data
        assert "profit_margin" in data


class TestAdminAIEngine:
```

```
    @pytest.mark.asyncio
    async def test_returns_performance_data(self, client, admin_user):
        _, headers = admin_user
        res = await client.get("/v1/admin/ai-engine?days=7", headers=headers)
        data = res.json()

        assert "performance" in data
        assert "tokens" in data
        assert "knowledge_impact" in data
        assert "errors" in data
        assert "confidence_distribution" in data
        assert "latency_trend" in data


class TestAdminKnowledge:

    @pytest.mark.asyncio
    async def test_returns_calibration_data(self, client, admin_user):
        _, headers = admin_user
        res = await client.get("/v1/admin/knowledge", headers=headers)
        data = res.json()

        assert "coverage" in data
        assert "top_patterns" in data
        assert "calibration" in data
        assert "top_1_accuracy" in data["calibration"]


class TestAdminAuditLog:

    @pytest.mark.asyncio
    async def test_admin_actions_logged(self, client, admin_user):
        _, headers = admin_user

        # Trigger an admin action
        await client.get("/v1/admin/overview", headers=headers)

        from database import db
        log = await db.fetch_one("SELECT * FROM admin_audit_log ORDER BY created_
        assert log is not None
        assert log["action"] == "viewed_overview"
```

---

# 6. INTEGRATION TESTS — SELF-LEARNING PIPELINE

The most important test section. Tests the complete feedback → aggregation → injection loop.

**File:** `tests/integration/test_self_learning_pipeline.py`

```
"""
End-to-end integration test for the self-learning pipeline:
  Analysis → Feedback → Aggregation → Knowledge Injection → Improved Analysis

This is the test that proves the moat works.
"""

import pytest
import json
from uuid import uuid4
from datetime import datetime, timedelta
from database import db
from services.knowledge_aggregator import run_aggregation
from services.knowledge_context import build_knowledge_context
from utils.normalizer import make_pair_key


class TestSelfLearningPipeline:

    @pytest.mark.asyncio
    async def test_full_pipeline_end_to_end(self, test_user):
        """The golden path test. Simulates the complete flywheel."""
        user, _ = test_user

        # ── PHASE 1: Seed data — Create 5 analyses with same substrate pair ──
        analysis_ids = []
        for i in range(5):
            a = await db.insert("failure_analyses", {
                "id": uuid4(), "user_id": user["id"],
                "material_category": "adhesive", "material_subcategory": "cyanoac
                "failure_mode": "debonding",
                "failure_description": f"CA debonded from aluminum case {i}",
                "substrate_a": "Aluminum 6061", "substrate_b": "ABS",
                "substrate_a_normalized": "aluminum_6061", "substrate_b_normalize
                "root_cause_category": "surface_preparation",
                "industry": "automotive",
                "root_causes": json.dumps([
                    {"rank": 1, "cause": "Surface preparation", "confidence": 0.8
                    {"rank": 2, "cause": "Moisture", "confidence": 0.55},
                ]),
                "status": "completed",
```

```
        "created_at": datetime.utcnow() - timedelta(days=30 - i),
    })
    analysis_ids.append(a["id"])

# ── PHASE 2: Submit feedback for 3 of the 5 ──
for i, aid in enumerate(analysis_ids[:3]):
    await db.insert("analysis_feedback", {
        "id": uuid4(), "user_id": user["id"], "analysis_id": aid,
        "was_helpful": True,
        "root_cause_confirmed": 1,  # Surface prep was correct
        "outcome": "resolved" if i < 2 else "partially_resolved",
        "what_worked": "Abrasion + acetone degrease" if i < 2 else "Added
        "what_didnt_work": "IPA wipe alone" if i == 0 else None,
        "feedback_source": "in_app",
    })

# ── PHASE 3: Run aggregation ──
result = await run_aggregation()
assert result["substrate_pairs_updated"] >= 1

# ── PHASE 4: Verify knowledge pattern created ──
pair_key = make_pair_key("Aluminum 6061", "ABS")
pattern = await db.fetch_one(
    "SELECT * FROM knowledge_patterns WHERE pattern_type = 'substrate_pai
    pair_key)

assert pattern is not None
assert pattern["total_cases"] == 5
assert pattern["cases_with_feedback"] == 3
assert pattern["resolved_cases"] >= 2
assert pattern["confidence_level"] == "medium"  # 5 cases, 3 feedback = m

causes = json.loads(pattern["top_root_causes"]) if isinstance(pattern["to
assert len(causes) >= 1
assert causes[0]["cause"] == "Surface preparation"
assert causes[0]["confirmed_count"] == 3

solutions = json.loads(pattern["effective_solutions"]) if isinstance(patt
assert any("Abrasion" in s["solution"] for s in solutions)

# ── PHASE 5: Build knowledge context for new similar analysis ──
context = await build_knowledge_context({
    "substrate_a": "Aluminum 6061",
    "substrate_b": "ABS",
    "failure_mode": "debonding",
    "material_subcategory": "cyanoacrylate",
})
```

```
    assert "5 previous analyses" in context
    assert "Surface prep" in context or "Surface preparation" in context
    assert "Abrasion" in context
    assert "IPA" in context  # ineffective solution should appear

@pytest.mark.asyncio
async def test_confidence_upgrades_with_more_data(self, test_user):
    """Verify confidence_level goes from low → medium → high as data grows."""
    user, _ = test_user

    # 3 cases, 1 feedback → low
    for i in range(3):
        a = await db.insert("failure_analyses", {
            "id": uuid4(), "user_id": user["id"],
            "material_category": "adhesive", "material_subcategory": "epoxy",
            "failure_mode": "cracking",
            "failure_description": f"Epoxy cracked {i}",
            "substrate_a": "Steel", "substrate_b": "Glass",
            "substrate_a_normalized": "steel_general", "substrate_b_normalize
            "root_causes": json.dumps([{"rank": 1, "cause": "Thermal cycling"
            "status": "completed",
            "created_at": datetime.utcnow() - timedelta(days=i),
        })
        if i == 0:
            await db.insert("analysis_feedback", {
                "id": uuid4(), "user_id": user["id"], "analysis_id": a["id"],
                "root_cause_confirmed": 1, "outcome": "resolved",
                "what_worked": "Toughened epoxy",
            })

    await run_aggregation()
    p = await db.fetch_one(
        "SELECT confidence_level FROM knowledge_patterns WHERE pattern_key =
        "glass::steel_general")
    assert p["confidence_level"] == "low"

    # Add more to reach medium (5+ cases, 3+ feedback)
    for i in range(4):
        a = await db.insert("failure_analyses", {
            "id": uuid4(), "user_id": user["id"],
            "material_category": "adhesive", "material_subcategory": "epoxy",
            "failure_mode": "cracking",
            "failure_description": f"More epoxy cracked {i}",
            "substrate_a": "Steel", "substrate_b": "Glass",
            "substrate_a_normalized": "steel_general", "substrate_b_normalize
            "root_causes": json.dumps([{"rank": 1, "cause": "Thermal cycling"
```

```
            "status": "completed",
            "created_at": datetime.utcnow() - timedelta(days=i),
        })
        if i < 3:
            await db.insert("analysis_feedback", {
                "id": uuid4(), "user_id": user["id"], "analysis_id": a["id"],
                "root_cause_confirmed": 1, "outcome": "resolved",
                "what_worked": "Toughened epoxy",
            })

    await run_aggregation()
    p = await db.fetch_one(
        "SELECT confidence_level FROM knowledge_patterns WHERE pattern_key =
        "glass::steel_general")
    assert p["confidence_level"] == "medium"

@pytest.mark.asyncio
async def test_no_knowledge_for_new_substrate_pair(self, test_user):
    """First analysis of a substrate pair should get no knowledge context."""
    context = await build_knowledge_context({
        "substrate_a": "Titanium",
        "substrate_b": "PEEK",
        "failure_mode": "debonding",
    })
    assert "No prior cases" in context

@pytest.mark.asyncio
async def test_aggregation_idempotent(self, test_user):
    """Running aggregation twice produces the same result."""
    user, _ = test_user

    for i in range(3):
        a = await db.insert("failure_analyses", {
            "id": uuid4(), "user_id": user["id"],
            "substrate_a_normalized": "copper", "substrate_b_normalized": "pm
            "failure_mode": "debonding", "failure_description": f"test {i}",
            "material_category": "adhesive", "material_subcategory": "epoxy",
            "root_causes": json.dumps([{"rank": 1, "cause": "test", "confiden
            "status": "completed",
        })

    await run_aggregation()
    p1 = await db.fetch_one(
        "SELECT total_cases FROM knowledge_patterns WHERE pattern_key = $1",

    await run_aggregation()
    p2 = await db.fetch_one(
```

```
            "SELECT total_cases FROM knowledge_patterns WHERE pattern_key = $1",

        assert p1["total_cases"] == p2["total_cases"]
```

---

# 7. INTEGRATION TESTS — OBSERVABILITY PIPELINE

**File:** `tests/integration/test_observability_pipeline.py`

```
"""Tests for the complete observability pipeline."""

import pytest
from datetime import date, timedelta
from database import db
from services.metrics_aggregator import aggregate_daily_metrics


class TestRequestLogging:

    @pytest.mark.asyncio
    async def test_requests_logged(self, client, test_user):
        _, headers = test_user
        await client.get("/v1/stats/public")  # This path is SKIP_PATHS
        await client.get("/v1/feedback/pending/list", headers=headers)  # This sh

        log = await db.fetch_one(
            "SELECT * FROM api_request_logs WHERE path = '/v1/feedback/pending/li
        assert log is not None
        assert log["method"] == "GET"
        assert log["status_code"] == 200
        assert log["latency_ms"] >= 0

    @pytest.mark.asyncio
    async def test_skip_paths_not_logged(self, client):
        await client.get("/v1/health")

        log = await db.fetch_one("SELECT * FROM api_request_logs WHERE path = '/v
        assert log is None

    @pytest.mark.asyncio
    async def test_error_responses_logged(self, client):
        res = await client.get("/v1/nonexistent")

        log = await db.fetch_one(
```

```
            "SELECT * FROM api_request_logs WHERE path = '/v1/nonexistent' ORDER
        assert log is not None
        assert log["status_code"] == 404


class TestDailyMetricsAggregation:

    @pytest.mark.asyncio
    async def test_aggregation_creates_row(self, test_user, sample_analysis, samp
        yesterday = date.today() - timedelta(days=1)
        result = await aggregate_daily_metrics(yesterday)

        assert result["status"] == "aggregated"

        row = await db.fetch_one("SELECT * FROM daily_metrics WHERE date = $1", y
        assert row is not None

    @pytest.mark.asyncio
    async def test_aggregation_idempotent(self, test_user):
        yesterday = date.today() - timedelta(days=1)
        await aggregate_daily_metrics(yesterday)
        await aggregate_daily_metrics(yesterday)

        count = await db.fetch_one("SELECT COUNT(*) as cnt FROM daily_metrics WHE
        assert count["cnt"] == 1  # Upserted, not duplicated


class TestAIEngineLogging:

    @pytest.mark.asyncio
    async def test_log_fields_populated(self, client, test_user, mock_ai_response
        from unittest.mock import patch
        _, headers = test_user

        with patch("services.ai_engine.AIEngine.analyze_failure", return_value=mo
            await client.post("/v1/analyze", json={
                "failure_description": "Bond failed",
                "substrate_a": "Al", "substrate_b": "ABS", "failure_mode": "debon
            }, headers=headers)

        log = await db.fetch_one("SELECT * FROM ai_engine_logs ORDER BY created_a
        assert log is not None
        assert log["request_type"] == "failure_analysis"
        assert log["model"] is not None
        assert log["total_tokens"] > 0
        assert log["latency_ms"] > 0
        assert log["estimated_cost_usd"] > 0
```

```
        assert log["error"] is False
```

---

# 8. E2E TESTS — USER FLOWS

**File:** `tests/e2e/test_failure_analysis_flow.py`

```
"""
E2E test: Complete failure analysis flow.
Submit → View results → Give feedback → Check feedback saved.
"""

import pytest
from playwright.async_api import async_playwright, expect


@pytest.fixture(scope="session")
async def browser():
    async with async_playwright() as p:
        browser = await p.chromium.launch()
        yield browser
        await browser.close()


@pytest.fixture
async def page(browser):
    page = await browser.new_page()
    yield page
    await page.close()


class TestFailureAnalysisE2E:

    @pytest.mark.asyncio
    async def test_submit_analysis_and_view_results(self, page):
        # Sign in (assumes test user exists with magic link bypass in test env)
        await page.goto("http://localhost:3000/failure")

        # Fill form
        await page.fill('[name="failure_description"]', 'Cyanoacrylate debonded f
        await page.fill('[name="substrate_a"]', 'Aluminum 6061')
        await page.fill('[name="substrate_b"]', 'ABS plastic')

        # Select failure mode card
```

```
    await page.click('text=Adhesive Failure')

    # Select optional fields
    await page.select_option('[name="industry"]', 'automotive')
    await page.select_option('[name="production_impact"]', 'quality_hold')

    # Submit
    await page.click('text=Analyze Failure')

    # Wait for results
    await page.wait_for_selector('text=Root Cause', timeout=30000)

    # Verify results rendered
    await expect(page.locator('text=Confidence')).to_be_visible()
    await expect(page.locator('[data-testid="root-cause-card"]').first).to_be

@pytest.mark.asyncio
async def test_feedback_prompt_visible(self, page):
    # After analysis results are shown
    await expect(page.locator('text=Was this analysis helpful')).to_be_visibl

@pytest.mark.asyncio
async def test_submit_feedback(self, page):
    # Click helpful
    await page.click('text=Helpful')

    # Select root cause
    await page.click('text=#1')  # Confirm first root cause

    # Submit
    await page.click('text=Submit feedback')

    # Verify success
    await expect(page.locator('text=Your feedback improves')).to_be_visible()

@pytest.mark.asyncio
async def test_confidence_badge_renders(self, page):
    # Check confidence badge is either "AI Estimated" or "Empirically Validat
    badge = page.locator('[data-testid="confidence-badge"]')
    await expect(badge).to_be_visible()
    text = await badge.text_content()
    assert "Estimated" in text or "Validated" in text

@pytest.mark.asyncio
async def test_export_pdf_button_exists(self, page):
    await expect(page.locator('text=Export PDF')).to_be_visible()
```

```
    @pytest.mark.asyncio
    async def test_cross_link_to_spec(self, page):
        # Click "Run Spec Analysis"
        await page.click('text=Run Spec Analysis')

        # Should navigate to /tool with substrates pre-filled
        await page.wait_for_url("**/tool**")

        sub_a_value = await page.input_value('[name="substrate_a"]')
        assert "Aluminum" in sub_a_value or "aluminum" in sub_a_value


class TestAuthFlowE2E:

    @pytest.mark.asyncio
    async def test_unauthenticated_redirected(self, page):
        await page.goto("http://localhost:3000/dashboard")
        await page.wait_for_url("**/")
        # Auth modal should appear
        await expect(page.locator('text=Sign in to Gravix')).to_be_visible()

    @pytest.mark.asyncio
    async def test_magic_link_flow(self, page):
        await page.goto("http://localhost:3000")
        await page.click('text=Sign In')

        await expect(page.locator('text=Sign in to Gravix')).to_be_visible()
        await page.fill('[name="email"]', 'test@example.com')
        await page.click('text=Send Magic Link')

        await expect(page.locator('text=Check your inbox')).to_be_visible()


class TestCaseLibraryE2E:

    @pytest.mark.asyncio
    async def test_case_library_loads(self, page):
        await page.goto("http://localhost:3000/cases")
        await page.wait_for_selector('[data-testid="case-card"]')

        cards = await page.locator('[data-testid="case-card"]').count()
        assert cards >= 1  # At least seed cases

    @pytest.mark.asyncio
    async def test_case_detail_page(self, page):
        await page.goto("http://localhost:3000/cases")
        await page.click('[data-testid="case-card"] >> nth=0')
```

```
        await page.wait_for_selector('[data-testid="case-detail"]')
        await expect(page.locator('text=Root Cause')).to_be_visible()
        await expect(page.locator('text=Solution')).to_be_visible()
        await expect(page.locator('text=Lessons Learned')).to_be_visible()


class TestPricingPageE2E:

    @pytest.mark.asyncio
    async def test_pricing_renders_three_plans(self, page):
        await page.goto("http://localhost:3000/pricing")

        await expect(page.locator('text=$0')).to_be_visible()
        await expect(page.locator('text=$49')).to_be_visible()
        await expect(page.locator('text=$149')).to_be_visible()

    @pytest.mark.asyncio
    async def test_faq_accordion_works(self, page):
        await page.goto("http://localhost:3000/pricing")

        await page.click('text=What counts as an analysis')
        await expect(page.locator('text=failure diagnosis or material specificati
```

---

# 9. E2E TESTS — ADMIN DASHBOARD

**File:** `tests/e2e/test_admin_dashboard.py`

```
"""E2E tests for admin dashboard pages."""

import pytest
from playwright.async_api import expect


class TestAdminDashboardE2E:

    @pytest.mark.asyncio
    async def test_non_admin_redirected(self, page):
        # Login as regular user
        await page.goto("http://localhost:3000/admin")
        await page.wait_for_url("**/dashboard")

    @pytest.mark.asyncio
```

```
async def test_overview_page_loads(self, page):
    # Login as admin user
    await page.goto("http://localhost:3000/admin")

    await expect(page.locator('text=Gravix Admin')).to_be_visible()
    await expect(page.locator('[data-testid="metric-card"]').first).to_be_vis

@pytest.mark.asyncio
async def test_ai_engine_page_loads(self, page):
    await page.goto("http://localhost:3000/admin/ai-engine")

    await expect(page.locator('text=Avg Latency')).to_be_visible()
    await expect(page.locator('text=Knowledge Impact')).to_be_visible()

@pytest.mark.asyncio
async def test_engagement_page_loads(self, page):
    await page.goto("http://localhost:3000/admin/engagement")

    await expect(page.locator('text=Feedback Funnel')).to_be_visible()

@pytest.mark.asyncio
async def test_knowledge_page_loads(self, page):
    await page.goto("http://localhost:3000/admin/knowledge")

    await expect(page.locator('text=Calibration')).to_be_visible()

@pytest.mark.asyncio
async def test_system_page_loads(self, page):
    await page.goto("http://localhost:3000/admin/system")

    await expect(page.locator('text=Endpoint Performance')).to_be_visible()

@pytest.mark.asyncio
async def test_sidebar_navigation(self, page):
    await page.goto("http://localhost:3000/admin")

    # Click through each nav item
    for label in ["AI Engine", "Engagement", "Knowledge Moat", "System Health
        await page.click(f'text={label}')
        await page.wait_for_load_state("networkidle")
```

---

# 10. AI ENGINE TESTS

These tests hit the actual Claude API. Run weekly or pre-release (costs API credits).

**File:** `tests/ai/test_prompt_quality.py`

```
"""
Tests that validate AI output quality and structure.
These call the real Claude API — run sparingly.
"""

import pytest
import json
from services.ai_engine import AIEngine

ai = AIEngine()

STANDARD_INPUT = {
    "material_category": "adhesive",
    "material_subcategory": "cyanoacrylate",
    "failure_mode": "debonding",
    "failure_description": "CA bond to aluminum failed after 2 weeks in automotiv
    "substrate_a": "Aluminum 6061",
    "substrate_b": "ABS plastic",
    "temperature_range": "-20C to 80C",
    "humidity": "65%",
    "surface_preparation": "IPA wipe",
}


class TestOutputStructure:

    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_response_has_required_fields(self):
        result = await ai.analyze_failure(STANDARD_INPUT)

        assert "root_causes" in result
        assert isinstance(result["root_causes"], list)
        assert len(result["root_causes"]) >= 1
        assert len(result["root_causes"]) <= 5

        for rc in result["root_causes"]:
            assert "rank" in rc
            assert "cause" in rc
            assert "confidence" in rc
            assert isinstance(rc["confidence"], (int, float))
            assert 0 <= rc["confidence"] <= 1

        assert "contributing_factors" in result
```

```
        assert "immediate_actions" in result
        assert "long_term_solutions" in result
        assert "confidence_score" in result

    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_root_causes_ranked_by_confidence(self):
        result = await ai.analyze_failure(STANDARD_INPUT)

        confidences = [rc["confidence"] for rc in result["root_causes"]]
        assert confidences == sorted(confidences, reverse=True)

    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_confidence_score_matches_top_cause(self):
        result = await ai.analyze_failure(STANDARD_INPUT)

        top_confidence = result["root_causes"][0]["confidence"]
        overall_confidence = result["confidence_score"]

        # Overall should be close to top cause's confidence
        assert abs(top_confidence - overall_confidence) < 0.15


class TestKnowledgeInjectionImpact:

    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_response_cites_gravix_data_when_injected(self):
        # Inject synthetic knowledge context
        from services.knowledge_context import build_knowledge_context
        from unittest.mock import AsyncMock, patch

        mock_pattern = {
            "total_cases": 23, "cases_with_feedback": 14, "resolution_rate": 0.86
            "top_root_causes": json.dumps([{"cause": "Surface preparation", "conf
            "effective_solutions": json.dumps([{"solution": "Abrasion + acetone",
            "ineffective_solutions": json.dumps([{"solution": "IPA only", "report
        }

        with patch("services.knowledge_context.db") as mock_db:
            mock_db.fetch_one = AsyncMock(side_effect=[mock_pattern, None])
            mock_db.fetch_all = AsyncMock(return_value=[])
            context = await build_knowledge_context(STANDARD_INPUT)

        # Call AI with knowledge context prepended
        enriched_input = {**STANDARD_INPUT, "_knowledge_context": context}
```

```
        result = await ai.analyze_failure(enriched_input)

        # The AI should reference the empirical data
        full_text = json.dumps(result)
        has_reference = any(phrase in full_text.lower() for phrase in [
            "gravix", "confirmed", "cases", "empirical", "database",
            "23 previous", "64%", "78%"
        ])
        assert has_reference, "AI response should reference injected knowledge da


class TestResponseParsing:

    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_json_parse_succeeds(self):
        result = await ai.analyze_failure(STANDARD_INPUT)

        # Result should be a dict (already parsed)
        assert isinstance(result, dict)

    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_no_html_in_output(self):
        result = await ai.analyze_failure(STANDARD_INPUT)

        full_text = json.dumps(result)
        assert "<html>" not in full_text
        assert "<script>" not in full_text
```

---

# 11. DATABASE & MIGRATION TESTS

**File:** `tests/integration/test_database.py`

```
"""Tests for database schema integrity after migrations."""

import pytest
from database import db


class TestSchemaIntegrity:

    @pytest.mark.asyncio
```

```
async def test_failure_analyses_has_new_columns(self):
    columns = await db.fetch_all(
        "SELECT column_name FROM information_schema.columns WHERE table_name
    col_names = {c["column_name"] for c in columns}

    assert "substrate_a_normalized" in col_names
    assert "substrate_b_normalized" in col_names
    assert "root_cause_category" in col_names
    assert "industry" in col_names
    assert "production_impact" in col_names

@pytest.mark.asyncio
async def test_feedback_table_exists(self):
    tables = await db.fetch_all(
        "SELECT table_name FROM information_schema.tables WHERE table_schema
    table_names = {t["table_name"] for t in tables}

    assert "analysis_feedback" in table_names
    assert "knowledge_patterns" in table_names
    assert "ai_engine_logs" in table_names
    assert "api_request_logs" in table_names
    assert "daily_metrics" in table_names
    assert "admin_audit_log" in table_names

@pytest.mark.asyncio
async def test_rls_enabled_on_sensitive_tables(self):
    rls_tables = await db.fetch_all("""
        SELECT tablename FROM pg_tables
        WHERE schemaname = 'public'
        AND tablename IN ('analysis_feedback', 'ai_engine_logs', 'api_request
        AND rowsecurity = true
    """)
    assert len(rls_tables) == 4

@pytest.mark.asyncio
async def test_indexes_exist(self):
    indexes = await db.fetch_all(
        "SELECT indexname FROM pg_indexes WHERE schemaname = 'public'")
    idx_names = {i["indexname"] for i in indexes}

    assert "idx_fa_substrates_norm" in idx_names
    assert "idx_feedback_analysis" in idx_names
    assert "idx_kp_lookup" in idx_names
    assert "idx_ai_logs_created" in idx_names

@pytest.mark.asyncio
async def test_check_constraints_on_enums(self):
```

```
        # Verify enum constraints work
        from asyncpg.exceptions import CheckViolationError

        with pytest.raises(Exception):  # CheckViolationError or equivalent
            await db.execute(
                "INSERT INTO failure_analyses (root_cause_category) VALUES ('inva

    @pytest.mark.asyncio
    async def test_feedback_reference_constraint(self):
        # Must provide either analysis_id or spec_id
        with pytest.raises(Exception):
            await db.execute("""
                INSERT INTO analysis_feedback (id, user_id, analysis_id, spec_id)
                VALUES (gen_random_uuid(), gen_random_uuid(), NULL, NULL)
            """)

    @pytest.mark.asyncio
    async def test_knowledge_pattern_unique_constraint(self):
        await db.execute("""
            INSERT INTO knowledge_patterns (id, pattern_type, pattern_key, total_
            VALUES (gen_random_uuid(), 'substrate_pair', 'test::test', 1)
        """)

        with pytest.raises(Exception):  # UniqueViolation
            await db.execute("""
                INSERT INTO knowledge_patterns (id, pattern_type, pattern_key, to
                VALUES (gen_random_uuid(), 'substrate_pair', 'test::test', 2)
            """)
```

---

# 12. PERFORMANCE & LOAD TESTS

**File:** `tests/performance/test_api_latency.py`

```
"""Performance tests — measure latency under load."""

import pytest
import asyncio
import time
from httpx import AsyncClient, ASGITransport
from api.main import app


class TestAPILatency:
```

```
    @pytest.mark.asyncio
    async def test_stats_endpoint_fast(self, client):
        start = time.time()
        res = await client.get("/v1/stats/public")
        elapsed = (time.time() - start) * 1000

        assert res.status_code == 200
        assert elapsed < 200, f"Stats endpoint took {elapsed}ms (max 200ms)"

    @pytest.mark.asyncio
    async def test_feedback_endpoint_fast(self, client, test_user, sample_analysi
        _, headers = test_user

        start = time.time()
        res = await client.post("/v1/feedback", json={
            "analysis_id": str(sample_analysis["id"]),
            "was_helpful": True,
        }, headers=headers)
        elapsed = (time.time() - start) * 1000

        assert res.status_code == 201
        assert elapsed < 500, f"Feedback endpoint took {elapsed}ms (max 500ms)"

    @pytest.mark.asyncio
    async def test_admin_overview_acceptable(self, client, admin_user):
        _, headers = admin_user

        start = time.time()
        res = await client.get("/v1/admin/overview", headers=headers)
        elapsed = (time.time() - start) * 1000

        assert res.status_code == 200
        assert elapsed < 2000, f"Admin overview took {elapsed}ms (max 2000ms)"


class TestConcurrency:

    @pytest.mark.asyncio
    async def test_10_concurrent_stats_requests(self):
        transport = ASGITransport(app=app)
        async with AsyncClient(transport=transport, base_url="http://test") as cl
            start = time.time()
            tasks = [client.get("/v1/stats/public") for _ in range(10)]
            responses = await asyncio.gather(*tasks)
            elapsed = (time.time() - start) * 1000
```

```
            assert all(r.status_code == 200 for r in responses)
            assert elapsed < 2000, f"10 concurrent requests took {elapsed}ms"


class TestAggregationPerformance:

    @pytest.mark.asyncio
    async def test_aggregation_with_1000_analyses(self, test_user):
        from tests.factories import create_bulk_analyses, create_bulk_feedback
        from services.knowledge_aggregator import run_aggregation
        from database import db

        user, _ = test_user
        analyses = await create_bulk_analyses(db, user["id"], count=1000)
        await create_bulk_feedback(db, user["id"], analyses, feedback_rate=0.3)

        start = time.time()
        result = await run_aggregation()
        elapsed = (time.time() - start) * 1000

        assert result["substrate_pairs_updated"] > 0
        assert elapsed < 30000, f"Aggregation of 1000 rows took {elapsed}ms (max


class TestDBQueryPerformance:

    @pytest.mark.asyncio
    async def test_knowledge_lookup_fast(self, test_user):
        from database import db

        # Insert a pattern
        await db.execute("""
            INSERT INTO knowledge_patterns (id, pattern_type, pattern_key, total_
            VALUES (gen_random_uuid(), 'substrate_pair', 'aluminum_6061::abs', 50
        """)

        start = time.time()
        for _ in range(100):
            await db.fetch_one(
                "SELECT * FROM knowledge_patterns WHERE pattern_type = 'substrate
                "aluminum_6061::abs")
        elapsed = (time.time() - start) * 1000

        per_query = elapsed / 100
        assert per_query < 10, f"Knowledge lookup averaged {per_query}ms (max 10m
```

# 13. SECURITY TESTS

**File:** `tests/security/test_auth_enforcement.py`

```
"""Security tests — authentication and authorization."""

import pytest


class TestAuthEnforcement:

    PROTECTED_ENDPOINTS = [
        ("POST", "/v1/analyze"),
        ("POST", "/v1/specify"),
        ("POST", "/v1/feedback"),
        ("GET", "/v1/feedback/pending/list"),
        ("GET", "/v1/admin/overview"),
        ("GET", "/v1/admin/ai-engine"),
        ("GET", "/v1/admin/engagement"),
        ("GET", "/v1/admin/knowledge"),
        ("GET", "/v1/admin/system"),
    ]

    @pytest.mark.asyncio
    @pytest.mark.parametrize("method,path", PROTECTED_ENDPOINTS)
    async def test_unauthenticated_rejected(self, client, method, path):
        if method == "GET":
            res = await client.get(path)
        else:
            res = await client.post(path, json={})

        assert res.status_code in [401, 403], f"{method} {path} returned {res.sta

    @pytest.mark.asyncio
    async def test_expired_token_rejected(self, client):
        headers = {"Authorization": "Bearer expired.invalid.token"}
        res = await client.get("/v1/feedback/pending/list", headers=headers)
        assert res.status_code in [401, 403]

    @pytest.mark.asyncio
    async def test_invalid_token_format(self, client):
        headers = {"Authorization": "NotBearer xyz"}
        res = await client.get("/v1/feedback/pending/list", headers=headers)
        assert res.status_code in [401, 403]
```

```
class TestAdminGating:

    ADMIN_ENDPOINTS = [
        "/v1/admin/overview",
        "/v1/admin/ai-engine",
        "/v1/admin/engagement",
        "/v1/admin/knowledge",
        "/v1/admin/system",
    ]

    @pytest.mark.asyncio
    @pytest.mark.parametrize("path", ADMIN_ENDPOINTS)
    async def test_regular_user_cannot_access_admin(self, client, test_user, path
        _, headers = test_user
        res = await client.get(path, headers=headers)
        assert res.status_code == 403


class TestCronProtection:

    @pytest.mark.asyncio
    async def test_cron_without_secret_rejected(self, client):
        res = await client.post("/v1/cron/aggregate-knowledge")
        assert res.status_code in [403, 422]

    @pytest.mark.asyncio
    async def test_cron_with_wrong_secret_rejected(self, client):
        res = await client.post("/v1/cron/aggregate-knowledge",
                                headers={"X-Cron-Secret": "wrong_secret"})
        assert res.status_code == 403


class TestDataIsolation:

    @pytest.mark.asyncio
    async def test_user_cannot_see_other_users_analyses(self, client, test_user,
        _, admin_headers = admin_user

        # Admin tries to get test_user's feedback
        res = await client.get(
            f"/v1/feedback/{sample_analysis['id']}", headers=admin_headers)

        # Should return None or 404 (not the actual feedback)
        assert res.status_code in [200, 404]
        if res.status_code == 200:
            assert res.json() is None
```

```
class TestInputSanitization:

    @pytest.mark.asyncio
    async def test_xss_in_description(self, client, test_user):
        _, headers = test_user
        from unittest.mock import patch, AsyncMock

        with patch("services.ai_engine.AIEngine.analyze_failure", new_callable=As
            mock.return_value = {"root_causes": [], "confidence_score": 0.5}

            res = await client.post("/v1/analyze", json={
                "failure_description": '<script>alert("xss")</script>Bond failed'
                "substrate_a": "Al", "substrate_b": "ABS", "failure_mode": "debon
            }, headers=headers)

        # Should not crash — script tags are stored as text, rendered safely by f
        assert res.status_code in [200, 422]

    @pytest.mark.asyncio
    async def test_sql_injection_in_substrate(self, client, test_user):
        _, headers = test_user
        from unittest.mock import patch, AsyncMock

        with patch("services.ai_engine.AIEngine.analyze_failure", new_callable=As
            mock.return_value = {"root_causes": [], "confidence_score": 0.5}

            res = await client.post("/v1/analyze", json={
                "failure_description": "Bond failed",
                "substrate_a": "'; DROP TABLE users; --",
                "substrate_b": "ABS", "failure_mode": "debonding",
            }, headers=headers)

        # Parameterized queries prevent injection — should not crash
        assert res.status_code in [200, 422]

        # Verify users table still exists
        from database import db
        check = await db.fetch_one("SELECT COUNT(*) as cnt FROM users")
        assert check["cnt"] >= 0
```

---

# 14. EMAIL & CRON TESTS

**File:** `tests/integration/test_cron_endpoints.py`

```
"""Tests for cron-triggered services."""

import pytest
from unittest.mock import patch, AsyncMock
from config import settings


class TestCronEndpoints:

    @pytest.mark.asyncio
    async def test_aggregate_knowledge_works(self, client, test_user, sample_anal
        res = await client.post("/v1/cron/aggregate-knowledge",
                                headers={"X-Cron-Secret": settings.CRON_SECRET})
        assert res.status_code == 200
        data = res.json()
        assert "substrate_pairs_updated" in data

    @pytest.mark.asyncio
    async def test_aggregate_metrics_works(self, client):
        res = await client.post("/v1/cron/aggregate-metrics",
                                headers={"X-Cron-Secret": settings.CRON_SECRET})
        assert res.status_code == 200

    @pytest.mark.asyncio
    @patch("services.feedback_email.resend")
    async def test_send_followups_works(self, mock_resend, client):
        mock_resend.Emails.send = AsyncMock(return_value={"id": "test"})

        res = await client.post("/v1/cron/send-followups",
                                headers={"X-Cron-Secret": settings.CRON_SECRET})
        assert res.status_code == 200
        data = res.json()
        assert "sent" in data
```

---

# 15. REGRESSION TEST SUITE

Tests that must pass on every deployment. Subset of the full suite, optimized for speed.

**File:** `tests/regression.py`

```
"""
Regression suite — run on every deploy.
Maximum runtime: 60 seconds.
```

```
Tests the critical paths that, if broken, make the product unusable.
"""

REGRESSION_TESTS = [
    # Auth
    "tests/integration/test_auth_flow.py::test_unauthenticated_rejected",
    "tests/security/test_auth_enforcement.py::TestAuthEnforcement",

    # Core analysis flow
    "tests/integration/test_analyze_endpoint.py::TestAnalyzeEndpoint::test_succes
    "tests/integration/test_analyze_endpoint.py::TestAnalyzeEndpoint::test_normal

    # Feedback
    "tests/integration/test_feedback_endpoint.py::TestFeedbackEndpoint::test_subm
    "tests/integration/test_feedback_endpoint.py::TestFeedbackEndpoint::test_cann

    # Knowledge pipeline
    "tests/integration/test_self_learning_pipeline.py::TestSelfLearningPipeline::

    # Admin gating
    "tests/integration/test_admin_endpoints.py::TestAdminGating",

    # Schema integrity
    "tests/integration/test_database.py::TestSchemaIntegrity::test_feedback_table

    # Normalizer (fast, catches regressions from map changes)
    "tests/unit/test_normalizer.py",
    "tests/unit/test_classifier.py",
]
```

Run with: `pytest tests/regression.py -x --timeout=60`

---

# 16. MANUAL QA CHECKLIST

Perform before each release. Estimated time: 45 minutes.

## 16.1 Core User Flow

```
[ ] Navigate to / — landing page renders with all 8 sections
[ ] Click "Try Spec Engine →" — arrives at /tool
[ ] Fill spec form, submit — results render with confidence badge
[ ] Click "Diagnose a Failure" in nav — arrives at /failure
[ ] Fill failure form with all fields including industry + production impact
```

```
[ ] Submit — loading state shows, results render within 15s
[ ] Root cause cards show rank, cause, confidence
[ ] Confidence badge shows "AI Estimated" (or "Empirically Validated" if data exi
[ ] Feedback prompt visible at bottom — click thumbs up
[ ] Root cause selector appears — select #1
[ ] Click "Tell us more" — detailed form expands
[ ] Select outcome, type "what worked" — submit
[ ] Success message appears
[ ] Click "Export PDF" — PDF downloads with analysis content
[ ] Click "Run Spec Analysis →" — navigates to /tool with substrates pre-filled
```

## 16.2 Auth Flow

```
[ ] Visit /tool without auth — auth modal appears
[ ] Enter email — "Send Magic Link" works
[ ] "Check your inbox" message appears
[ ] Click Google button — OAuth flow completes
[ ] After sign in — redirected to original page
[ ] User avatar in nav shows name
[ ] Click avatar → "Sign Out" — returns to landing page
```

## 16.3 Pricing & Billing

```
[ ] Visit /pricing — 3 plans render with correct prices ($0, $49, $149)
[ ] Pro card has "Most Popular" badge
[ ] FAQ accordion opens/closes
[ ] Click "Upgrade to Pro" — Stripe checkout opens
[ ] Complete test payment — plan updates
[ ] Visit /settings — shows "Pro" plan, "Manage Subscription" works
```

## 16.4 Free Tier Limits

```
[ ] As free user, submit 5 analyses — all succeed
[ ] Submit 6th — "Upgrade" prompt appears, analysis blocked
[ ] History shows only last 5 (older ones are "Upgrade to Pro" blurred)
[ ] Executive summary is blurred/preview only
[ ] PDF has watermark
```

## 16.5 Case Library

- [ ] Visit /cases — seed cases render as card grid
- [ ] Filter by material — cards filter correctly
- [ ] Click a card — detail page renders with all sections
- [ ] "Run your own analysis" CTA links to /failure
- [ ] Check /cases/cyanoacrylate-debonding-aluminum-surface-prep — SEO-indexed page

## 16.6 Admin Dashboard

- [ ] Log in as admin — /admin accessible
- [ ] Overview: hero cards show numbers, trend charts render
- [ ] AI Engine: latency and cost data render
- [ ] Engagement: feedback funnel visible, top users table loads
- [ ] Knowledge Moat: calibration accuracy shows, top patterns table loads
- [ ] System Health: endpoint table loads, error log scrollable
- [ ] Log in as regular user — /admin returns 403 redirect

## 16.7 Email Follow-Up

- [ ] Submit analysis, wait 24h (or manually trigger cron)
- [ ] Check email — follow-up received with correct substrates in subject
- [ ] Click "Resolved ✓" link — lands on /feedback/[id]?outcome=resolved
- [ ] Page shows "Recorded: resolved" banner
- [ ] Feedback form available for additional detail

## 16.8 Stats Bar

- [ ] Visit /tool or /failure — stats bar visible below nav
- [ ] Numbers are non-zero and formatted correctly
- [ ] Stats update after new analyses are submitted

## 16.9 Responsive Design

- [ ] Landing page on mobile (375px) — all sections stack, readable
- [ ] Failure form on tablet (768px) — form full-width, results below
- [ ] Pricing on mobile — cards stack vertically, Pro first
- [ ] Admin dashboard on tablet — sidebar collapses
- [ ] Nav hamburger on mobile — menu opens, all links accessible

---

# 17. TEST DATA STRATEGY

## 17.1 Environments

| Environment | Database | AI Engine | Email | Purpose |
|---|---|---|---|---|
| **Local dev** | Local Supabase | Mocked | Console output | Unit + integration tests |
| **CI (GitHub Actions)** | Supabase branch DB | Mocked | Disabled | PR checks |
| **Staging** | Supabase staging project | Real Claude API | Resend sandbox | Full E2E, manual QA |
| **Production** | Supabase production | Real Claude API | Resend production | Smoke tests only |

## 17.2 Test Data Seeding

For staging:

- 6 seed case library entries (from migration 004)
- 1 admin user (your email)
- 50 synthetic analyses with normalized fields
- 20 feedback entries (mixed outcomes)
- Knowledge patterns generated from aggregation run

Script: `scripts/seed_staging.py`

## 17.3 Data Cleanup

- CI: truncate all tables before each test (conftest autouse fixture)
- Staging: reset weekly via `scripts/reset_staging.py`
- Production: never delete. Smoke tests use a dedicated test user.

---

# 18. CI/CD PIPELINE CONFIGURATION

## 18.1 GitHub Actions Workflow

```
# .github/workflows/test.yml

name: Test Suite

on:
  pull_request:
    branches: [main]
  push:
    branches: [main]

jobs:
  unit-and-integration:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:15
        env:
          POSTGRES_DB: gravix_test
          POSTGRES_USER: test
          POSTGRES_PASSWORD: test
        ports: [5432:5432]
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Install dependencies
        run: pip install -r requirements.txt -r requirements-test.txt

      - name: Run migrations
        run: |
          psql $DATABASE_URL < database/migrations/001_v1_1_structured_fields.sql
          psql $DATABASE_URL < database/migrations/002_v1_1_feedback_knowledge.sq
          psql $DATABASE_URL < database/migrations/003_v1_1_observability.sql
```

```
        psql $DATABASE_URL < database/migrations/004_v1_1_seed_cases.sql
      env:
        DATABASE_URL: postgresql://test:test@localhost:5432/gravix_test

    - name: Run unit tests
      run: pytest tests/unit/ -v --timeout=30

    - name: Run integration tests
      run: pytest tests/integration/ -v --timeout=120
      env:
        DATABASE_URL: postgresql://test:test@localhost:5432/gravix_test
        CRON_SECRET: test_cron_secret
        ADMIN_SECRET: test_admin_secret

    - name: Run security tests
      run: pytest tests/security/ -v --timeout=60

e2e:
  runs-on: ubuntu-latest
  needs: unit-and-integration
  if: github.ref == 'refs/heads/main'

  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: "20"

    - name: Install Playwright
      run: npx playwright install --with-deps chromium

    - name: Start services
      run: docker-compose -f docker-compose.test.yml up -d

    - name: Run E2E tests
      run: pytest tests/e2e/ -v --timeout=300

    - name: Upload screenshots on failure
      if: failure()
      uses: actions/upload-artifact@v4
      with:
        name: playwright-screenshots
        path: tests/e2e/screenshots/

ai-quality:
  runs-on: ubuntu-latest
  if: github.event.schedule == '0 3 * * 1'  # Weekly Monday 3am
```

```
    steps:
      - uses: actions/checkout@v4
      - name: Run AI quality tests
        run: pytest tests/ai/ -v --timeout=300 -m slow
        env:
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
```

## 18.2 Test Commands

```
# Run everything locally
pytest tests/ -v

# Unit tests only (fast, no DB needed)
pytest tests/unit/ -v --timeout=30

# Integration tests (needs DB)
pytest tests/integration/ -v --timeout=120

# E2E tests (needs full stack running)
pytest tests/e2e/ -v --timeout=300

# Regression suite (deploy gate)
pytest tests/regression.py -x --timeout=60

# AI quality tests (costs API credits)
pytest tests/ai/ -v -m slow --timeout=300

# Performance tests
pytest tests/performance/ -v --timeout=600

# Security tests
pytest tests/security/ -v --timeout=60

# Single test file
pytest tests/integration/test_self_learning_pipeline.py -v

# With coverage
pytest tests/ --cov=api --cov-report=html
```

## 18.3 Coverage Targets

| Module | Target |
| --- | --- |

| | |
|---|---|
| `utils/normalizer.py` | 95% |
| `utils/classifier.py` | 90% |
| `services/knowledge_context.py` | 90% |
| `services/knowledge_aggregator.py` | 85% |
| `services/ai_telemetry.py` | 80% |
| `routers/feedback.py` | 90% |
| `routers/admin.py` | 80% |
| `middleware/request_logger.py` | 75% |
| **Overall** | **80%** |

---

**END OF GRAVIX TESTING PLAN**