```
    deploy_ok = abs(evaluator.deploy_weight - expected_deploy) < 1e-6

    if theory_ok and impl_ok and deploy_ok:
        print("✓ All weights match JSON values")
        return True
    else:
        print("✗ Weights don't match JSON:")
        if not theory_ok:
            print(f"  theory: expected {expected_theory}, got {evaluator.theory_weight}")
        if not impl_ok:
            print(f"  impl: expected {expected_impl}, got {evaluator.impl_weight}")
        if not deploy_ok:
            print(f"  deploy: expected {expected_deploy}, got {evaluator.deploy_weight}")
        return False


def test_weights_sum_to_one():
    """Test that weights sum to 1.0."""
    print("=" * 80)
    print("TEST 2: Weights Sum to 1.0")
    print("=" * 80)
    print()

    evaluator = BaseLayerEvaluator("config/intrinsic_calibration.json")

    total = evaluator.theory_weight + evaluator.impl_weight + evaluator.deploy_weight

    print(f"Weight sum: {total}")
    print()

    if abs(total - 1.0) < 1e-6:
        print("✓ Weights sum to 1.0")
        return True
    else:
        print(f"✗ Weights sum to {total}, not 1.0")
        return False


def test_consistency_with_intrinsic_loader():
    """Test that BaseLayerEvaluator produces same scores as IntrinsicScoreLoader."""
    print("=" * 80)
    print("TEST 3: Consistency with IntrinsicScoreLoader")
    print("=" * 80)
    print()

    base_evaluator = BaseLayerEvaluator("config/intrinsic_calibration.json")
    intrinsic_loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")

    # Test a few methods (only calibrated ones)
    test_methods = [
        "orchestrator.__init__.__getattr__",
        "orchestrator.factory.build_processor",
        "src.saaaaaa.core.orchestrator.executors.D1Q1_Executor.execute",
        "src.saaaaaa.core.orchestrator.executors.D3Q2_Executor.execute",
        "src.saaaaaa.core.orchestrator.executors.D6Q5_Executor.execute",
    ]

    print("Comparing scores (calibrated methods only):")
    print()

    all_match = True
    for method_id in test_methods:
        # Skip if not calibrated
        if not intrinsic_loader.is_calibrated(method_id):
            print(f"⊘ {method_id[:55]:55s} (not calibrated, skipping)")
            continue
        # Get score from BaseLayerEvaluator
        layer_score = base_evaluator.evaluate(method_id)
        base_score = layer_score.score

        # Get score from IntrinsicScoreLoader
        intrinsic_score = intrinsic_loader.get_score(method_id)

        # Compare
        difference = abs(base_score - intrinsic_score)
        match = difference < 0.001

        status = "✓" if match else "✗"

        if not match:
            all_match = False

        print(f"{status} {method_id[:55]:55s}")
        print(f"   BaseLayer:  {base_score:.6f}")
        print(f"   Intrinsic:  {intrinsic_score:.6f}")
        print(f"   Difference: {difference:.6f}")
        print()

    if all_match:
        print("✓ All scores match between BaseLayerEvaluator and IntrinsicScoreLoader")
        return True
    else:
        print("✗ Some scores don't match")
        return False


def test_no_old_hardcoded_weights():
    """Test that old hardcoded weights (0.4, 0.4, 0.2) are NOT used."""
    print("=" * 80)
    print("TEST 4: Old Hardcoded Weights NOT Used")
    print("=" * 80)
    print()

    evaluator = BaseLayerEvaluator("config/intrinsic_calibration.json")

    # Old buggy weights
    old_theory = 0.4
    old_impl = 0.4
    old_deploy = 0.2

    print(f"Old (buggy) hardcoded weights:")
    print(f"  theory: {old_theory}")
    print(f"  impl:   {old_impl}")
    print(f"  deploy: {old_deploy}")
    print()

    print(f"Current weights:")
    print(f"  theory: {evaluator.theory_weight}")
    print(f"  impl:   {evaluator.impl_weight}")
    print(f"  deploy: {evaluator.deploy_weight}")
    print()

    # Check if using old weights
    using_old = (
        abs(evaluator.theory_weight - old_theory) < 1e-6 and
        abs(evaluator.impl_weight - old_impl) < 1e-6 and
        abs(evaluator.deploy_weight - old_deploy) < 1e-6
    )

    if not using_old:
        print("✓ NOT using old hardcoded weights")
        return True
    else:
        print("✗ STILL USING old hardcoded weights (BUG NOT FIXED!)")
        return False
```

```
if __name__ == "__main__":
    print("\nBASE LAYER EVALUATOR WEIGHT FIX VERIFICATION")
    print()

    # Run tests
    test1 = test_weights_loaded_from_json()
    print()
    test2 = test_weights_sum_to_one()
    print()
    test3 = test_consistency_with_intrinsic_loader()
    print()
    test4 = test_no_old_hardcoded_weights()

    # Summary
    print()
    print("=" * 80)
    print("FINAL RESULTS")
    print("=" * 80)
    print(f"Weights loaded from JSON: {'✓ PASS' if test1 else '✗ FAIL'}")
    print(f"Weights sum to 1.0: {'✓ PASS' if test2 else '✗ FAIL'}")
    print(f"Consistency with IntrinsicScoreLoader: {'✓ PASS' if test3 else '✗ FAIL'}")
    print(f"Old weights NOT used: {'✓ PASS' if test4 else '✗ FAIL'}")
    print()

    if all([test1, test2, test3, test4]):
        print("🎉 ALL TESTS PASSED - BUG FIXED!")
        sys.exit(0)
    else:
        print("⚠  SOME TESTS FAILED - BUG NOT FULLY FIXED")
        sys.exit(1)

===== FILE: tests/test_boot_checks_runtime.py =====
"""
Tests for boot checks with runtime configuration integration.

Tests boot check behavior across different runtime modes (PROD, DEV, EXPLORATORY)
and validates proper error handling and fallback behavior.
"""

import pytest
from unittest.mock import patch, MagicMock

from saaaaaa.core.runtime_config import RuntimeConfig, RuntimeMode
from saaaaaa.core.boot_checks import (
    run_boot_checks,
    get_boot_check_summary,
    BootCheckError,
    check_contradiction_module_available,
    check_wiring_validator_available,
    check_spacy_model_available,
    check_networkx_available,
)


class TestBootChecksInProdMode:
    """Test boot checks in PROD mode (strict enforcement)."""

    def test_all_checks_pass_in_prod(self):
        """All checks passing should succeed in PROD mode."""
        config = RuntimeConfig(
            mode=RuntimeMode.PROD,
            allow_contradiction_fallback=False,
            allow_execution_estimates=False,
            allow_dev_ingestion_fallbacks=False,
            allow_aggregation_defaults=False,
            strict_calibration=True,
            allow_validator_disable=False,
            allow_hash_fallback=False,
            preferred_spacy_model="es_core_news_lg"
        )

        # Should not raise if all dependencies available
        results = run_boot_checks(config)

        # All checks should pass
        assert all(result["status"] == "passed" for result in results.values())

        # Summary should show success
        summary = get_boot_check_summary(results)
        assert "passed" in summary.lower()

    def test_missing_critical_module_fails_in_prod(self):
        """Missing critical module should raise BootCheckError in PROD."""
        config = RuntimeConfig(
            mode=RuntimeMode.PROD,
            allow_contradiction_fallback=False,
            allow_execution_estimates=False,
            allow_dev_ingestion_fallbacks=False,
            allow_aggregation_defaults=False,
            strict_calibration=True,
            allow_validator_disable=False,
            allow_hash_fallback=False,
            preferred_spacy_model="es_core_news_lg"
        )

        # Mock contradiction module as unavailable
        with patch('saaaaaa.core.boot_checks.check_contradiction_module_available') as
mock_check:
            mock_check.return_value = {
                "status": "failed",
                "component": "contradiction_module",
                "reason": "Module not found",
                "code": "MODULE_NOT_FOUND"
            }

            # Should raise in PROD mode
            with pytest.raises(BootCheckError) as exc_info:
                run_boot_checks(config)

            assert exc_info.value.component == "contradiction_module"
            assert "MODULE_NOT_FOUND" in exc_info.value.code

    def test_spacy_model_missing_fails_in_prod(self):
        """Missing spaCy model should fail in PROD mode."""
        config = RuntimeConfig(
            mode=RuntimeMode.PROD,
            allow_contradiction_fallback=False,
            allow_execution_estimates=False,
            allow_dev_ingestion_fallbacks=False,
            allow_aggregation_defaults=False,
            strict_calibration=True,
            allow_validator_disable=False,
            allow_hash_fallback=False,
            preferred_spacy_model="es_core_news_lg"
        )

        with patch('saaaaaa.core.boot_checks.check_spacy_model_available') as mock_check:
            mock_check.return_value = {
                "status": "failed",
                "component": "spacy_model",
                "reason": "Model es_core_news_lg not found",
                "code": "SPACY_MODEL_NOT_FOUND"
            }

            with pytest.raises(BootCheckError):
```

```
            run_boot_checks(config)


class TestBootChecksInDevMode:
    """Test boot checks in DEV mode (permissive with warnings)."""

    def test_missing_module_warns_in_dev(self):
        """Missing module should warn but not fail in DEV mode."""
        config = RuntimeConfig(
            mode=RuntimeMode.DEV,
            allow_contradiction_fallback=True,
            allow_execution_estimates=True,
            allow_dev_ingestion_fallbacks=True,
            allow_aggregation_defaults=True,
            strict_calibration=False,
            allow_validator_disable=True,
            allow_hash_fallback=True,
            preferred_spacy_model="es_core_news_lg"
        )

        with patch('saaaaaa.core.boot_checks.check_contradiction_module_available') as
mock_check:
            mock_check.return_value = {
                "status": "warning",
                "component": "contradiction_module",
                "reason": "Module not found but fallback allowed",
                "code": "MODULE_NOT_FOUND_FALLBACK_ALLOWED"
            }

            # Should not raise in DEV mode
            results = run_boot_checks(config)

            # Check should show warning status
            assert results["contradiction_module"]["status"] == "warning"

    def test_networkx_missing_allowed_in_dev(self):
        """NetworkX missing should be allowed in DEV mode."""
        config = RuntimeConfig(
            mode=RuntimeMode.DEV,
            allow_contradiction_fallback=True,
            allow_execution_estimates=True,
            allow_dev_ingestion_fallbacks=True,
            allow_aggregation_defaults=True,
            strict_calibration=False,
            allow_validator_disable=True,
            allow_hash_fallback=True,
            preferred_spacy_model="es_core_news_lg"
        )

        with patch('saaaaaa.core.boot_checks.check_networkx_available') as mock_check:
            mock_check.return_value = {
                "status": "warning",
                "component": "networkx",
                "reason": "NetworkX not available, graph metrics will be skipped",
                "code": "NETWORKX_NOT_FOUND"
            }

            results = run_boot_checks(config)

            # Should complete without raising
            assert results["networkx"]["status"] == "warning"


class TestBootChecksInExploratoryMode:
    """Test boot checks in EXPLORATORY mode (maximum flexibility)."""

    def test_all_fallbacks_allowed_in_exploratory(self):
        """All fallbacks should be allowed in EXPLORATORY mode."""
        config = RuntimeConfig(
            mode=RuntimeMode.EXPLORATORY,
            allow_contradiction_fallback=True,
            allow_execution_estimates=True,
            allow_dev_ingestion_fallbacks=True,
            allow_aggregation_defaults=True,
            strict_calibration=False,
            allow_validator_disable=True,
            allow_hash_fallback=True,
            preferred_spacy_model="es_core_news_sm"  # Can use smaller model
        )

        # Even with multiple missing dependencies, should not fail
        with patch('saaaaaa.core.boot_checks.check_contradiction_module_available') as 
mock1, \
             patch('saaaaaa.core.boot_checks.check_networkx_available') as mock2:

            mock1.return_value = {"status": "warning", "component": 
"contradiction_module",
                                  "reason": "Fallback allowed", "code": "FALLBACK"}
            mock2.return_value = {"status": "warning", "component": "networkx",
                                  "reason": "Fallback allowed", "code": "FALLBACK"}

            results = run_boot_checks(config)

            # Should complete with warnings
            assert results["contradiction_module"]["status"] == "warning"
            assert results["networkx"]["status"] == "warning"


class TestIndividualBootChecks:
    """Test individual boot check functions."""

    def test_check_contradiction_module_available(self):
        """Test contradiction module availability check."""
        result = check_contradiction_module_available()

        assert "status" in result
        assert "component" in result
        assert result["component"] == "contradiction_module"

        # Status should be either passed or failed
        assert result["status"] in ["passed", "failed", "warning"]

    def test_check_wiring_validator_available(self):
        """Test wiring validator availability check."""
        result = check_wiring_validator_available()

        assert "status" in result
        assert "component" in result
        assert result["component"] == "wiring_validator"

    def test_check_spacy_model_with_preferred(self):
        """Test spaCy model check with preferred model."""
        # Test with LG model
        result = check_spacy_model_available("es_core_news_lg")

        assert "status" in result
        assert "component" in result
        assert result["component"] == "spacy_model"

        if result["status"] == "passed":
            assert "model" in result
            assert result["model"] in ["es_core_news_lg", "es_core_news_md", 
"es_core_news_sm"]

    def test_check_networkx_available(self):
        """Test NetworkX availability check."""
        result = check_networkx_available()

        assert "status" in result
        assert "component" in result
        assert result["component"] == "networkx"


class TestBootCheckSummary:
    """Test boot check summary generation."""

    def test_summary_all_passed(self):
        """Test summary when all checks pass."""
        results = {
            "check1": {"status": "passed", "component": "check1"},
            "check2": {"status": "passed", "component": "check2"},
            "check3": {"status": "passed", "component": "check3"},
        }

        summary = get_boot_check_summary(results)

        assert "3/3 passed" in summary or "passed" in summary.lower()
        assert "✓" in summary or "check1" in summary

    def test_summary_with_failures(self):
        """Test summary when some checks fail."""
        results = {
            "check1": {"status": "passed", "component": "check1"},
            "check2": {"status": "failed", "component": "check2", "reason": "Not found"},
            "check3": {"status": "warning", "component": "check3", "reason": "Degraded"},
        }

        summary = get_boot_check_summary(results)

        # Should show failure count
        assert "failed" in summary.lower() or "✗" in summary
        assert "check2" in summary or "Not found" in summary

    def test_summary_with_warnings(self):
        """Test summary with warnings."""
        results = {
            "check1": {"status": "passed", "component": "check1"},
            "check2": {"status": "warning", "component": "check2", "reason": "Degraded"},
        }

        summary = get_boot_check_summary(results)

        assert "warning" in summary.lower() or "⚠" in summary


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_boundaries.py =====
"""Architecture guardrail tests for pure core modules and layering.

This module enforces the architectural guardrails requested by the
refactoring plan:

* Every ``saaaaaa.core`` module must be importable without crashing.
* Pure library modules must stay free from ``__main__`` blocks and direct I/O.
* ``import-linter`` layer contracts must remain satisfied when available.
"""

from __future__ import annotations

import ast
import importlib
import importlib.util
import pkgutil
import subprocess
import sys
from pathlib import Path
from typing import TYPE_CHECKING

import pytest

from saaaaaa.config.paths import PROJECT_ROOT, SRC_DIR

if TYPE_CHECKING:
    from collections.abc import Iterable

# Define the package root directory
REPO_ROOT = PROJECT_ROOT
PACKAGE_ROOT = SRC_DIR

# Modules that must stay pure (no __main__ and no direct I/O).
PURE_MODULE_PATHS: dict[str, Path] = {
    "saaaaaa.processing.embedding_policy": PACKAGE_ROOT / "processing" /
"embedding_policy.py",
}

# Legacy modules still undergoing I/O migration. We record them so that the
# detector can surface the locations without failing the build yet.
LEGACY_IO_MODULES: dict[str, Path] = {
    "saaaaaa.analysis.Analyzer_one": PACKAGE_ROOT / "analysis" / "Analyzer_one.py",
    "saaaaaa.analysis.derek_beach": PACKAGE_ROOT / "analysis" / "derek_beach.py",
    "saaaaaa.analysis.financiero_viabilidad_tablas": PACKAGE_ROOT / "analysis" /
"financiero_viabilidad_tablas.py",
    "saaaaaa.analysis.teoria_cambio": PACKAGE_ROOT / "analysis" / "teoria_cambio.py",
    "saaaaaa.analysis.contradiction_deteccion": PACKAGE_ROOT / "analysis" /
"contradiction_deteccion.py",
    "saaaaaa.processing.semantic_chunking_policy": PACKAGE_ROOT / "processing" /
"semantic_chunking_policy.py",
}

class _IODetector(ast.NodeVisitor):
    """AST visitor that flags direct file/network I/O usage."""

    IO_FUNCTIONS = {
        "open",
        "read",
        "write",
        "load",
        "dump",
        "loads",
        "dumps",
        "read_csv",
        "read_excel",
        "read_json",
        "read_sql",
        "read_parquet",
        "to_csv",
        "to_excel",
        "to_json",
        "to_sql",
        "to_parquet",
    }
    IO_MODULES = {"pickle", "json", "yaml", "toml", "pathlib"}

    def __init__(self) -> None:
        self.matches: list[int] = []

    def visit_Call(self, node: ast.Call) -> None:  # pragma: no cover - simple visitor
        func = node.func
        if isinstance(func, ast.Name):
            if func.id in self.IO_FUNCTIONS:
                self.matches.append(node.lineno)
        elif isinstance(func, ast.Attribute) and isinstance(func.attr, str):
            if func.attr in self.IO_FUNCTIONS or isinstance(func.value, ast.Name) and
func.value.id in self.IO_MODULES:
                self.matches.append(node.lineno)
        self.generic_visit(node)

    def visit_With(self, node: ast.With) -> None:  # pragma: no cover - simple visitor
        for item in node.items:
            ctx = item.context_expr
            if isinstance(ctx, ast.Call) and isinstance(ctx.func, ast.Name):
                if ctx.func.id == "open":
                    self.matches.append(node.lineno)
        self.generic_visit(node)

class _MainDetector(ast.NodeVisitor):
    """AST visitor that flags ``if __name__ == '__main__'`` blocks."""

    def __init__(self) -> None:
        self.locations: list[int] = []

    def visit_If(self, node: ast.If) -> None:  # pragma: no cover - simple visitor
        test = node.test
        if (
            isinstance(test, ast.Compare)
            and isinstance(test.left, ast.Name)
            and test.left.id == "__name__"
        ):
            for comparator in test.comparators:
                if isinstance(comparator, ast.Constant) and comparator.value ==
"__main__":
                    self.locations.append(node.lineno)
        self.generic_visit(node)

def _load_source(path: Path) -> ast.AST:
    with path.open("r", encoding="utf-8") as handle:
        source = handle.read()
    try:
        return ast.parse(source, filename=str(path))
    except SyntaxError as exc:  # pragma: no cover - sanity guard
        pytest.fail(f"Syntax error while parsing {path}: {exc}")

def _iter_core_modules() -> Iterable[str]:
    package_path = PACKAGE_ROOT / "core"
    for module_info in pkgutil.walk_packages([str(package_path)], prefix="saaaaaa.core."):
        if not module_info.ispkg:
            yield module_info.name

@pytest.mark.parametrize("module_name", sorted(_iter_core_modules()))
def test_core_modules_import_cleanly(module_name: str) -> None:
    """Every module inside ``saaaaaa.core`` must be importable."""

    spec = importlib.util.find_spec(module_name)
    if spec is None:
        pytest.fail(f"Cannot find module {module_name} on sys.path")

    try:
        importlib.import_module(module_name)
    except ImportError as exc:  # pragma: no cover - exercised only when failing
        pytest.fail(f"Importing {module_name} failed: {exc}")

@pytest.mark.parametrize("qualified_name, path", sorted(PURE_MODULE_PATHS.items()))
def test_pure_modules_have_no_main_blocks(qualified_name: str, path: Path) -> None:
    tree = _load_source(path)
    detector = _MainDetector()
    detector.visit(tree)
    assert not detector.locations, (
        f"{qualified_name} contains __main__ guards at lines {detector.locations}. "
        "Pure modules must not ship executable entry points."
    )

@pytest.mark.parametrize("qualified_name, path", sorted({
    **PURE_MODULE_PATHS,
    **LEGACY_IO_MODULES,
}.items()))
def test_ast_scanner_reports_io_usage(qualified_name: str, path: Path) -> None:
    """Detect direct I/O in core analysis modules."""

    if not path.exists():
        pytest.skip(f"Module file for {qualified_name} is missing")

    tree = _load_source(path)
    detector = _IODetector()
    detector.visit(tree)
    if qualified_name in LEGACY_IO_MODULES:
        if detector.matches:
            pytest.skip(
                f"{qualified_name} still performs I/O at lines {detector.matches[:10]}. "
                "Track migrations before flipping this test to strict mode."
            )
        return

    assert not detector.matches, (
        f"{qualified_name} contains I/O operations at lines {detector.matches}. "
        "Core libraries must remain pure."
    )

def test_import_linter_layer_contract(tmp_path: Path) -> None:
    """Run a lightweight import-linter contract when the tool is available."""

    if importlib.util.find_spec("importlinter") is None:
        pytest.skip("import-linter is not installed in this environment")

    config = tmp_path / "importlinter.ini"
    config.write_text(
        """
[importlinter]
root_package = saaaaaa

[contract:core-does-not-import-tests]
name = Core package must not import tests
type = forbidden
source_modules =
    saaaaaa.core
forbidden_modules =
    tests
    saaaaaa.tests
        """.strip()
    )

    completed = subprocess.run(
        [sys.executable, "-m", "importlinter", "contracts", "--config", str(config)],
        cwd=str(REPO_ROOT),
        check=False,
        capture_output=True,
        text=True,
    )

    if completed.returncode == 2:
        pytest.skip("import-linter not configured correctly in this environment")

    stdout = completed.stdout + completed.stderr
    assert completed.returncode == 0, (
        "import-linter detected a layering violation:\n" + stdout
    )
```

```
def test_boundary_scanner_tool_exists() -> None:
    scanner_path = REPO_ROOT / "tools" / "scan_boundaries.py"
    assert scanner_path.exists(), "Boundary scanner tool not found"
    _ = _load_source(scanner_path)

===== FILE: tests/test_calibration_completeness.py =====
"""Test calibration completeness - ensure all methods have explicit calibration.

This test suite enforces the requirement that every method referenced by the
pipeline must have an explicit calibration entry. No silent defaults allowed.

Per the refactoring requirements:
- Every method must have explicit calibration indexed by method_fqn
- Missing calibrations must raise MissingCalibrationError
- No generic fallback calibrations permitted

OBSOLETE: This test uses the old calibration_registry API (CALIBRATIONS dict,
get_calibration_hash, etc.) which was refactored to use resolve_calibration().
New calibration tests are in tests/calibration/ subdirectory.
See tests/calibration/test_gap0_complete.py for current implementation.
"""

import pytest

pytestmark = pytest.mark.skip(reason="obsolete - calibration_registry API refactored, see
tests/calibration/")

# Old imports (no longer valid):
# from saaaaaa.core.orchestrator.calibration_registry import (
#     CALIBRATIONS,
#     MissingCalibrationError,
#     resolve_calibration,
#     get_calibration_hash,
#     CALIBRATION_VERSION,
# )


class TestCalibrationCompleteness:
    """Test that all methods have explicit calibrations."""

    def test_calibration_version_exists(self):
        """Verify calibration version is defined."""
        assert CALIBRATION_VERSION is not None
        assert isinstance(CALIBRATION_VERSION, str)
        assert len(CALIBRATION_VERSION) > 0

    def test_calibration_hash_deterministic(self):
        """Verify calibration hash is deterministic."""
        hash1 = get_calibration_hash()
        hash2 = get_calibration_hash()

        assert hash1 == hash2
        assert len(hash1) == 64  # SHA256 hex digest

    def test_calibrations_not_empty(self):
        """Verify calibration registry is not empty."""
        assert len(CALIBRATIONS) > 0
        # We know we have 166 calibrations from the generated registry
        assert len(CALIBRATIONS) >= 100

    def test_all_calibrations_have_valid_keys(self):
        """Verify all calibrations use proper (ClassName, method_name) keys."""
        for key in CALIBRATIONS.keys():
            assert isinstance(key, tuple)
            assert len(key) == 2
            class_name, method_name = key
            assert isinstance(class_name, str)
            assert isinstance(method_name, str)
            assert len(class_name) > 0
            assert len(method_name) > 0

    def test_all_calibrations_have_valid_values(self):
        """Verify all calibration values are MethodCalibration instances."""
        from saaaaaa.core.orchestrator.calibration_registry import MethodCalibration

        for key, calib in CALIBRATIONS.items():
            assert isinstance(calib, MethodCalibration), f"Invalid calibration for {key}"

            # Verify required fields
            assert 0.0 <= calib.score_min <= 1.0
            assert 0.0 <= calib.score_max <= 1.0
            assert calib.score_min <= calib.score_max
            assert calib.min_evidence_snippets >= 1
            assert calib.max_evidence_snippets >= calib.min_evidence_snippets
            assert 0.0 <= calib.contradiction_tolerance <= 1.0
            assert 0.0 <= calib.uncertainty_penalty <= 2.0
            assert calib.aggregation_weight >= 0.1
            assert 0.0 <= calib.sensitivity <= 1.0

    def test_resolve_calibration_strict_mode(self):
        """Test that strict mode raises error for missing calibrations."""
        # Should raise for non-existent method
        with pytest.raises(MissingCalibrationError) as exc_info:
            resolve_calibration("NonExistentClass", "nonexistent_method", strict=True)

        assert "NonExistentClass.nonexistent_method" in str(exc_info.value)
        assert "Missing calibration" in str(exc_info.value)

    def test_resolve_calibration_non_strict_mode(self):
        """Test that non-strict mode returns None for missing calibrations."""
        calib = resolve_calibration("NonExistentClass", "nonexistent_method",
strict=False)
        assert calib is None

    def test_resolve_calibration_success(self):
        """Test successful calibration resolution."""
        # Use a known calibration from registry
        calib = resolve_calibration("BayesianEvidenceScorer", "compute_evidence_score",
strict=True)

        assert calib is not None
        assert calib.score_min == 0.0
        assert calib.score_max == 1.0
        assert calib.min_evidence_snippets >= 1

    def test_no_default_like_calibrations_without_flag(self):
        """Verify that default-like calibrations have safe_default_allowed flag."""
        from saaaaaa.core.orchestrator.calibration_registry import MethodCalibration

        for key, calib in CALIBRATIONS.items():
            if calib.is_default_like():
                # If a calibration is "default-like", it should explicitly allow it
                # Currently none are expected to be default-like in strict regime
                # This test documents the expectation
                assert not calib.is_default_like(), (
                    f"Calibration for {key} appears default-like. "
                    f"If intentional, set safe_default_allowed=True"
                )

    def test_missing_calibration_error_attributes(self):
        """Test MissingCalibrationError has proper attributes."""
        error = MissingCalibrationError("TestClass.test_method", {"question_id": "D1Q1"})

        assert error.method_fqn == "TestClass.test_method"
        assert error.context == {"question_id": "D1Q1"}
        assert "TestClass.test_method" in str(error)
        assert "D1Q1" in str(error)


class TestCalibrationIndexing:
    """Test calibration indexing by different dimensions."""

    def test_calibrations_indexed_by_class_and_method(self):
        """Verify calibrations are indexed by (class_name, method_name)."""
        # Get a sample calibration
        sample_key = list(CALIBRATIONS.keys())[0]
        assert isinstance(sample_key, tuple)
        assert len(sample_key) == 2

    def test_method_fqn_construction(self):
        """Test that method FQN is properly constructed."""
        class_name = "TestClass"
        method_name = "test_method"
        expected_fqn = "TestClass.test_method"

        # MissingCalibrationError constructs FQN
        error = MissingCalibrationError(expected_fqn)
        assert error.method_fqn == expected_fqn


class TestCalibrationMetadata:
    """Test calibration metadata and tracking."""

    def test_calibration_has_document_type_field(self):
        """Verify MethodCalibration dataclass has document_type field."""
        from saaaaaa.core.orchestrator.calibration_registry import MethodCalibration

        # Create a calibration with document_type
        calib = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=3,
            max_evidence_snippets=10,
            contradiction_tolerance=0.1,
            uncertainty_penalty=0.2,
            aggregation_weight=1.0,
            sensitivity=0.8,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
            safe_default_allowed=False,
            document_type="plan_desarrollo_municipal"
        )

        assert calib.document_type == "plan_desarrollo_municipal"

    def test_calibration_safe_default_flag(self):
        """Verify safe_default_allowed flag is present."""
        from saaaaaa.core.orchestrator.calibration_registry import MethodCalibration

        calib = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=1,
            max_evidence_snippets=5,
            contradiction_tolerance=0.5,
            uncertainty_penalty=0.5,
            aggregation_weight=1.0,
            sensitivity=0.5,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=False,
            safe_default_allowed=True  # Explicitly allowed
        )
        assert calib.safe_default_allowed is True


# FIXME(CALIBRATION): Add tests for specific question/method mappings
# Once question catalog is integrated, add tests to verify:
# - All questions have method assignments
# - All assigned methods have calibrations
# - Context-aware calibration resolution works for each question

===== FILE: tests/test_calibration_context.py =====
"""Tests for context-aware calibration system."""

import pytest

from src.saaaaaa.core.orchestrator.calibration_registry import (
    MethodCalibration,
    resolve_calibration,
    resolve_calibration_with_context,
)
from src.saaaaaa.core.orchestrator.calibration_context import (
    CalibrationContext,
    CalibrationModifier,
    PolicyArea,
    UnitOfAnalysis,
    resolve_contextual_calibration,
    infer_context_from_question_id,
)


class TestCalibrationContext:
    """Test CalibrationContext creation and manipulation."""

    def test_from_question_id_valid(self):
        """Test creating context from valid question ID."""
        context = CalibrationContext.from_question_id("D1Q1")
        assert context.question_id == "D1Q1"
        assert context.dimension == 1
        assert context.question_num == 1

    def test_from_question_id_various_formats(self):
        """Test parsing various question ID formats."""
        test_cases = [
            ("D6Q3", 6, 3),
            ("d2q5", 2, 5),
            ("D10Q25", 10, 25),
        ]
        for qid, expected_dim, expected_q in test_cases:
            context = CalibrationContext.from_question_id(qid)
            assert context.dimension == expected_dim
            assert context.question_num == expected_q

    def test_from_question_id_invalid(self):
        """Test handling invalid question IDs."""
        context = CalibrationContext.from_question_id("invalid")
        assert context.question_id == "invalid"
        assert context.dimension == 0
        assert context.question_num == 0

    def test_with_policy_area(self):
        """Test updating policy area."""
        context = CalibrationContext.from_question_id("D1Q1")
        new_context = context.with_policy_area(PolicyArea.FISCAL)

        assert new_context.policy_area == PolicyArea.FISCAL
        assert context.policy_area == PolicyArea.UNKNOWN  # Original unchanged
        assert new_context.dimension == context.dimension  # Other fields preserved
    def test_with_unit_of_analysis(self):
        """Test updating unit of analysis."""
        context = CalibrationContext.from_question_id("D1Q1")
        new_context = context.with_unit_of_analysis(UnitOfAnalysis.BASELINE_GAP)

        assert new_context.unit_of_analysis == UnitOfAnalysis.BASELINE_GAP
        assert context.unit_of_analysis == UnitOfAnalysis.UNKNOWN

    def test_with_method_position(self):
        """Test updating method position."""
        context = CalibrationContext.from_question_id("D1Q1")
        new_context = context.with_method_position(2, 5)

        assert new_context.method_position == 2
        assert new_context.total_methods == 5
        assert context.method_position == 0  # Original unchanged


class TestCalibrationModifier:
    """Test CalibrationModifier application."""

    def test_identity_modifier(self):
        """Test that default modifier doesn't change calibration."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=3,
            max_evidence_snippets=18,
            contradiction_tolerance=0.05,
            uncertainty_penalty=0.25,
            aggregation_weight=1.0,
            sensitivity=0.85,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        modifier = CalibrationModifier()
        result = modifier.apply(base)

        assert result.min_evidence_snippets == base.min_evidence_snippets
        assert result.max_evidence_snippets == base.max_evidence_snippets
        assert result.contradiction_tolerance ==
pytest.approx(base.contradiction_tolerance)
        assert result.uncertainty_penalty == pytest.approx(base.uncertainty_penalty)
        assert result.aggregation_weight == pytest.approx(base.aggregation_weight)
        assert result.sensitivity == pytest.approx(base.sensitivity)

    def test_evidence_multiplier(self):
        """Test evidence snippet multipliers."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=10,
            max_evidence_snippets=20,
            contradiction_tolerance=0.05,
            uncertainty_penalty=0.25,
            aggregation_weight=1.0,
            sensitivity=0.85,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        modifier = CalibrationModifier(
            min_evidence_multiplier=1.5,
            max_evidence_multiplier=1.2,
        )
        result = modifier.apply(base)

        assert result.min_evidence_snippets == 15  # 10 * 1.5
        assert result.max_evidence_snippets == 24  # 20 * 1.2

    def test_clamping(self):
        """Test that modifiers clamp values to valid ranges."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=3,
            max_evidence_snippets=18,
            contradiction_tolerance=0.9,
            uncertainty_penalty=0.1,
            aggregation_weight=1.0,
            sensitivity=0.9,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        # Try to push values out of range
        modifier = CalibrationModifier(
            contradiction_tolerance_multiplier=2.0,  # Would be 1.8 > 1.0
            uncertainty_penalty_multiplier=0.1,  # Would be 0.01 < 0.0 (OK)
            sensitivity_multiplier=1.5,  # Would be 1.35 > 1.0
        )
        result = modifier.apply(base)

        assert result.contradiction_tolerance <= 1.0
        assert result.contradiction_tolerance >= 0.0
        assert result.sensitivity <= 1.0
        assert result.sensitivity >= 0.0


class TestContextualCalibration:
    """Test context-aware calibration resolution."""

    def test_no_context_returns_base(self):
        """Test that no context returns base calibration unchanged."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=3,
            max_evidence_snippets=18,
            contradiction_tolerance=0.05,
            uncertainty_penalty=0.25,
            aggregation_weight=1.0,
            sensitivity=0.85,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        result = resolve_contextual_calibration(base, None)
        assert result == base

    def test_dimension_modifier_applied(self):
        """Test dimension-specific modifiers are applied."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=10,
            max_evidence_snippets=20,
            contradiction_tolerance=0.5,
            uncertainty_penalty=0.5,
            aggregation_weight=1.0,
            sensitivity=0.5,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        # D1 has min_evidence_multiplier=1.3
        context = CalibrationContext.from_question_id("D1Q1")
        result = resolve_contextual_calibration(base, context)

        # Should have more evidence requirements for D1 (baseline gaps)
        assert result.min_evidence_snippets > base.min_evidence_snippets

    def test_policy_area_modifier_applied(self):
        """Test policy area modifiers are applied."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=10,
            max_evidence_snippets=20,
            contradiction_tolerance=0.5,
            uncertainty_penalty=0.5,
            aggregation_weight=1.0,
            sensitivity=0.5,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        context = CalibrationContext.from_question_id("D1Q1")
        context = context.with_policy_area(PolicyArea.FISCAL)
        result = resolve_contextual_calibration(base, context)

        # Fiscal policy should increase evidence requirements and sensitivity
        assert result.min_evidence_snippets > base.min_evidence_snippets
        assert result.sensitivity > base.sensitivity

    def test_cumulative_modifiers(self):
        """Test that multiple modifiers are cumulative."""
        base = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=10,
            max_evidence_snippets=20,
            contradiction_tolerance=0.5,
            uncertainty_penalty=0.5,
            aggregation_weight=1.0,
            sensitivity=0.5,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        # Apply dimension + policy area + unit modifiers
        context = CalibrationContext.from_question_id("D9Q1")  # Financial dimension
        context = context.with_policy_area(PolicyArea.FISCAL)
        context = context.with_unit_of_analysis(UnitOfAnalysis.FINANCIAL)
        result = resolve_contextual_calibration(base, context)

        # Should have significantly higher requirements
        # D9, fiscal, and financial all increase evidence needs
        assert result.min_evidence_snippets > base.min_evidence_snippets * 1.5
        assert result.sensitivity > base.sensitivity


class TestInferContext:
    """Test context inference from question ID."""

    def test_infer_dimension_1_baseline_gap(self):
        """Test D1 infers baseline gap unit."""
        context = infer_context_from_question_id("D1Q1")
        assert context.dimension == 1
        assert context.unit_of_analysis == UnitOfAnalysis.BASELINE_GAP

    def test_infer_dimension_2_indicator(self):
        """Test D2 infers indicator unit."""
        context = infer_context_from_question_id("D2Q3")
        assert context.dimension == 2
        assert context.unit_of_analysis == UnitOfAnalysis.INDICATOR

    def test_infer_dimension_9_financial(self):
        """Test D9 infers financial unit."""
        context = infer_context_from_question_id("D9Q1")
        assert context.dimension == 9
        assert context.unit_of_analysis == UnitOfAnalysis.FINANCIAL


class TestIntegrationWithRegistry:
    """Test integration with calibration registry."""

    def test_resolve_calibration_backward_compatible(self):
        """Test that resolve_calibration still works without context."""
        # BayesianEvidenceScorer.compute_evidence_score exists in registry
        calib = resolve_calibration("BayesianEvidenceScorer", "compute_evidence_score")
        assert calib is not None
        assert isinstance(calib, MethodCalibration)

    def test_resolve_with_context_returns_different(self):
        """Test that context changes calibration."""
        base = resolve_calibration("BayesianEvidenceScorer", "compute_evidence_score")
        assert base is not None

        # Resolve with fiscal policy context
        contextual = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D9Q1",
            policy_area="fiscal",
        )
        assert contextual is not None

        # Should be different due to D9 + fiscal modifiers
        assert (
            contextual.min_evidence_snippets != base.min_evidence_snippets
            or contextual.sensitivity != base.sensitivity
        )

    def test_resolve_with_context_different_questions(self):
        """Test that same method gets different calibration for different questions."""
        # D1Q1 (baseline gap)
        d1_calib = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D1Q1",
        )

        # D6Q3 (logical framework)
        d6_calib = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D6Q3",
        )

        assert d1_calib is not None
        assert d6_calib is not None

        # Should have different characteristics
        # D6 has lower contradiction tolerance (stricter)
        assert d6_calib.contradiction_tolerance < d1_calib.contradiction_tolerance

    def test_method_position_affects_calibration(self):
        """Test that method position in sequence affects calibration."""
        # First method in sequence
        first = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D1Q1",
            method_position=0,
            total_methods=5,
        )

        # Last method in sequence
        last = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D1Q1",
            method_position=4,
            total_methods=5,
        )

        assert first is not None
        assert last is not None

        # Early methods should have higher evidence needs
        assert first.min_evidence_snippets >= last.min_evidence_snippets
        # Late methods should have higher aggregation weight (synthesis)
        assert last.aggregation_weight > first.aggregation_weight


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_calibration_integration.py =====
"""
Integration tests for the complete calibration system.

Tests the end-to-end flow:
1. Loading intrinsic_calibration.json
2. Loading method_parameters.json
3. Determining required layers by method type
4. Computing calibration scores via orchestrator
5. Making validation decisions
"""
import sys
from pathlib import Path

# Add project root to path
sys.path.insert(0, str(Path(__file__).parent.parent))

from src.saaaaaa.core.calibration.intrinsic_loader import IntrinsicScoreLoader
from src.saaaaaa.core.calibration.parameter_loader import MethodParameterLoader
from src.saaaaaa.core.calibration.layer_requirements import LayerRequirementsResolver
from src.saaaaaa.core.calibration.orchestrator import CalibrationOrchestrator
from src.saaaaaa.core.calibration.validator import CalibrationValidator
from src.saaaaaa.core.calibration.data_structures import LayerID


class TestIntrinsicLoader:
    """Test intrinsic calibration loader."""

    def test_load_intrinsic_calibration(self):
        """Test loading intrinsic_calibration.json."""
        loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")

        # Should load lazily
        assert not loader._loaded

        # Get statistics (triggers load)
        stats = loader.get_statistics()

        assert loader._loaded
        assert stats["total"] > 0
        assert stats["computed"] > 0
        print(f"✓ Loaded {stats['total']} methods, {stats['computed']} computed")

    def test_get_score_for_calibrated_method(self):
        """Test getting score for a method that exists."""
        loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")

        # Should have some methods calibrated
        stats = loader.get_statistics()
        assert stats["computed"] > 0

        # Get a score (using a method we know exists from exploration)
        # Note: Replace with actual method from your intrinsic_calibration.json
        score = loader.get_score("some.method.name", default=0.5)

        # Score should be in valid range
        assert 0.0 <= score <= 1.0
        print(f"✓ Retrieved score: {score}")

    def test_get_layer_for_method(self):
        """Test getting layer/role for a method."""
        loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")

        # Get layer for a method
        # Note: This will return None for methods not in JSON
        layer = loader.get_layer("some.method.name")

        # Layer should be one of the expected types or None
        if layer:
            expected_layers = {
                "analyzer", "processor", "ingest", "structure",
                "extract", "aggregate", "report", "utility",
                "orchestrator", "meta", "transform", "score", "core"
            }
            assert layer.lower() in expected_layers
            print(f"✓ Method layer: {layer}")


class TestParameterLoader:
    """Test method parameter loader."""

    def test_load_method_parameters(self):
        """Test loading method_parameters.json."""
        loader = MethodParameterLoader("config/method_parameters.json")

        # Should load lazily
        assert not loader._loaded

        # Get statistics (triggers load)
        stats = loader.get_statistics()

        assert loader._loaded
        assert stats["total_executors"] == 30  # All 30 executors configured
        print(f"✓ Loaded {stats['total_executors']} executors, {stats['total_methods']}
methods")

    def test_get_quality_thresholds(self):
        """Test getting global quality thresholds."""
        loader = MethodParameterLoader("config/method_parameters.json")

        thresholds = loader.get_quality_thresholds()
        assert thresholds["excellent"] == 0.85
        assert thresholds["good"] == 0.70
        assert thresholds["acceptable"] == 0.55
        assert thresholds["insufficient"] == 0.0
        print(f"✓ Quality thresholds: {thresholds}")

    def test_get_executor_threshold(self):
        """Test getting executor-specific threshold."""
        loader = MethodParameterLoader("config/method_parameters.json")

        # Test a few executors
        d1q1_threshold = loader.get_executor_threshold("D1Q1_Executor")
        d4q1_threshold = loader.get_executor_threshold("D4Q1_Executor")

        assert 0.0 < d1q1_threshold <= 1.0
        assert 0.0 < d4q1_threshold <= 1.0

        # D4Q1 (financial) should have higher threshold
        assert d4q1_threshold >= d1q1_threshold
        print(f"✓ D1Q1 threshold: {d1q1_threshold}, D4Q1 threshold: {d4q1_threshold}")

    def test_get_validation_threshold_by_role(self):
        """Test getting validation threshold by role type."""
        loader = MethodParameterLoader("config/method_parameters.json")

        analyzer_threshold = loader.get_validation_threshold_for_role("analyzer")
        utility_threshold = loader.get_validation_threshold_for_role("utility")

        # Analyzer should have higher threshold than utility
        assert analyzer_threshold > utility_threshold
        print(f"✓ Analyzer: {analyzer_threshold}, Utility: {utility_threshold}")


class TestLayerRequirements:
    """Test layer requirements resolver."""

    def test_resolver_initialization(self):
        """Test that resolver initializes correctly."""
        intrinsic_loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")
        resolver = LayerRequirementsResolver(intrinsic_loader)

        # All role mappings should include BASE layer
        for role, layers in resolver.ROLE_LAYER_MAP.items():
            assert LayerID.BASE in layers
            print(f"✓ Role '{role}' includes BASE layer")

    def test_get_required_layers_for_analyzer(self):
        """Test that analyzer methods require all 8 layers."""
        intrinsic_loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")
        resolver = LayerRequirementsResolver(intrinsic_loader)

        # Analyzer should have all 8 layers
        analyzer_layers = resolver.ROLE_LAYER_MAP["analyzer"]

        expected_layers = {
            LayerID.BASE,
            LayerID.UNIT,
            LayerID.QUESTION,
            LayerID.DIMENSION,
            LayerID.POLICY,
            LayerID.CONGRUENCE,
            LayerID.CHAIN,
            LayerID.META
        }

        assert analyzer_layers == expected_layers
        print(f"✓ Analyzer requires {len(analyzer_layers)} layers: all 8")
    def test_get_required_layers_for_utility(self):
        """Test that utility methods require minimal layers."""
        intrinsic_loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")
        resolver = LayerRequirementsResolver(intrinsic_loader)

        # Utility should have minimal layers
        utility_layers = resolver.ROLE_LAYER_MAP["utility"]

        # Should have BASE, CHAIN, META (minimal)
        assert LayerID.BASE in utility_layers
        assert LayerID.CHAIN in utility_layers
        assert LayerID.META in utility_layers

        # Should NOT have contextual layers
        assert LayerID.QUESTION not in utility_layers
        assert LayerID.DIMENSION not in utility_layers
        assert LayerID.POLICY not in utility_layers

        print(f"✓ Utility requires {len(utility_layers)} layers (minimal)")


class TestOrchestratorIntegration:
    """Test calibration orchestrator integration."""

    def test_orchestrator_initialization(self):
        """Test that orchestrator initializes with all loaders."""
        orchestrator = CalibrationOrchestrator(
            intrinsic_calibration_path="config/intrinsic_calibration.json"
        )

        # Check that all components are initialized
        assert orchestrator.intrinsic_loader is not None
        assert orchestrator.layer_resolver is not None
        assert orchestrator.base_evaluator is not None
        assert orchestrator.unit_evaluator is not None

        print("✓ Orchestrator initialized with all components")

    def test_orchestrator_has_layer_resolver(self):
        """Test that orchestrator has layer resolver."""
        orchestrator = CalibrationOrchestrator(
            intrinsic_calibration_path="config/intrinsic_calibration.json"
        )

        # Should have layer_resolver attribute
        assert hasattr(orchestrator, 'layer_resolver')
        assert orchestrator.layer_resolver is not None

        # Test layer resolver functionality
        required_layers = orchestrator.layer_resolver.get_required_layers("test.method")
        assert isinstance(required_layers, set)
        assert len(required_layers) > 0

        print(f"✓ Layer resolver working, returns {len(required_layers)} layers for
unknown method")


class TestValidatorIntegration:
    """Test validator integration."""

    def test_validator_initialization(self):
        """Test validator initialization."""
        orchestrator = CalibrationOrchestrator(
            intrinsic_calibration_path="config/intrinsic_calibration.json"
        )
        parameter_loader = MethodParameterLoader("config/method_parameters.json")
        validator = CalibrationValidator(
            orchestrator=orchestrator,
            parameter_loader=parameter_loader
        )

        assert validator.orchestrator is not None
        assert validator.parameter_loader is not None
        assert validator.intrinsic_loader is not None

        print("✓ Validator initialized successfully")

    def test_get_threshold_for_executor(self):
        """Test threshold determination for executors."""
        orchestrator = CalibrationOrchestrator(
            intrinsic_calibration_path="config/intrinsic_calibration.json"
        )
        parameter_loader = MethodParameterLoader("config/method_parameters.json")
        validator = CalibrationValidator(
            orchestrator=orchestrator,
            parameter_loader=parameter_loader
        )

        # Get threshold for an executor
        threshold = validator._get_threshold_for_method("D1Q1_Executor")

        assert 0.0 < threshold <= 1.0
        assert threshold >= 0.60  # Executors should have high thresholds

        print(f"✓ D1Q1_Executor threshold: {threshold}")

    def test_validate_method_flow(self):
        """Test full validation flow (skipped - requires PDT)."""
        # This would test the full flow but requires a valid PDT structure
        # and context, which are complex to mock
        print("  (Skipped - requires full PDT structure)")


class TestEndToEndFlow:
    """Test the complete end-to-end flow."""

    def test_complete_system_initialization(self):
        """Test that all components can be initialized together."""
        # 1. Load intrinsic calibration
        intrinsic_loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")
        intrinsic_stats = intrinsic_loader.get_statistics()
        print(f"✓ Step 1: Loaded intrinsic calibration ({intrinsic_stats['computed']}
methods)")

        # 2. Load method parameters
        parameter_loader = MethodParameterLoader("config/method_parameters.json")
        param_stats = parameter_loader.get_statistics()
        print(f"✓ Step 2: Loaded method parameters ({param_stats['total_executors']}
executors)")

        # 3. Initialize layer resolver
        layer_resolver = LayerRequirementsResolver(intrinsic_loader)
        print("✓ Step 3: Initialized layer requirements resolver")

        # 4. Initialize orchestrator
        orchestrator = CalibrationOrchestrator(
            intrinsic_calibration_path="config/intrinsic_calibration.json"
        )
        print("✓ Step 4: Initialized calibration orchestrator")

        # 5. Initialize validator
        validator = CalibrationValidator(
            orchestrator=orchestrator,
            parameter_loader=parameter_loader
        )
        print("✓ Step 5: Initialized validator")

        # Verify all components are connected
        assert validator.orchestrator.intrinsic_loader is not None
        assert validator.orchestrator.layer_resolver is not None
        assert validator.parameter_loader is not None

        print("\n✓ Complete system initialized successfully!")

    def test_executor_configuration_complete(self):
        """Test that all 30 executors are configured."""
        parameter_loader = MethodParameterLoader("config/method_parameters.json")

        # Check all 30 executors
        all_configured = True
        missing = []

        for d in range(1, 7):
            for q in range(1, 6):
                executor_name = f"D{d}Q{q}_Executor"
                threshold = parameter_loader.get_executor_threshold(executor_name,
default=None)

                if threshold is None:
                    all_configured = False
                    missing.append(executor_name)

        if all_configured:
            print("✓ All 30 executors are configured with thresholds")
        else:
            print(f"✗ Missing configurations for: {missing}")

        assert all_configured, f"Missing executor configurations: {missing}"

    def test_layer_mapping_completeness(self):
        """Test that all role types have layer mappings."""
        intrinsic_loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")
        resolver = LayerRequirementsResolver(intrinsic_loader)

        expected_roles = {
            "analyzer", "processor", "ingest", "structure",
            "extract", "aggregate", "report", "utility",
            "orchestrator", "meta", "transform"
        }

        for role in expected_roles:
            assert role in resolver.ROLE_LAYER_MAP
            layers = resolver.ROLE_LAYER_MAP[role]
            assert LayerID.BASE in layers  # All must include BASE

        print(f"✓ All {len(expected_roles)} role types have layer mappings")


if __name__ == "__main__":
    """Run tests with detailed output."""
    print("=" * 80)
    print("CALIBRATION SYSTEM INTEGRATION TESTS")
    print("=" * 80)
    print()

    # Run each test class
    test_classes = [
        TestIntrinsicLoader,
        TestParameterLoader,
        TestLayerRequirements,
        TestOrchestratorIntegration,
        TestValidatorIntegration,
        TestEndToEndFlow
    ]

    for test_class in test_classes:
        print(f"\n{test_class.__name__}:")
        print("-" * 80)

        test_instance = test_class()
        for method_name in dir(test_instance):
            if method_name.startswith("test_"):
                try:
                    method = getattr(test_instance, method_name)
                    print(f"\n  {method_name}:")
                    method()
                except Exception as e:
                    print(f"  ✗ FAILED: {e}")

    print("\n" + "=" * 80)
    print("TESTS COMPLETE")
    print("=" * 80)

===== FILE: tests/test_calibration_stability.py =====
"""Test calibration stability - verify deterministic execution.

This test suite enforces that:
- Same calibration + same seed → same results
- Calibration changes → detectable via hash
- ExecutorConfig drives deterministic behavior

Per the refactoring requirements:
- Calibrations must be versioned and hashed
- Same config + seed must produce deterministic results
- Config changes must be traceable via hash

OBSOLETE: This test uses old calibration_registry API (get_calibration_hash,
CALIBRATION_VERSION) which no longer exist. See tests/calibration/ for current tests.
"""

import pytest

pytestmark = pytest.mark.skip(reason="obsolete - calibration_registry API refactored, see
tests/calibration/")

# Old imports (no longer valid):
# from saaaaaa.core.orchestrator.calibration_registry import (
#     get_calibration_hash,
#     CALIBRATION_VERSION,
#     resolve_calibration,
# )
# from saaaaaa.core.orchestrator.executor_config import ExecutorConfig


class TestCalibrationVersioning:
    """Test calibration versioning and hashing."""

    def test_calibration_version_stable(self):
        """Verify calibration version is stable across calls."""
        version1 = CALIBRATION_VERSION
        version2 = CALIBRATION_VERSION

        assert version1 == version2
        assert isinstance(version1, str)

    def test_calibration_hash_stable(self):
        """Verify calibration hash is stable across calls."""
        hash1 = get_calibration_hash()
        hash2 = get_calibration_hash()
        assert hash1 == hash2
        assert isinstance(hash1, str)
        assert len(hash1) == 64  # SHA256

    def test_calibration_hash_changes_with_data(self):
        """Verify hash would change if calibrations change.

        This is a documentation test - we verify hash is computed from
        calibration data, not just a constant.
        """
        hash_value = get_calibration_hash()

        # Hash should be non-trivial
        assert hash_value != "0" * 64
        assert hash_value != "f" * 64

        # Hash should be deterministic hex
        assert all(c in "0123456789abcdef" for c in hash_value)


class TestExecutorConfigDeterminism:
    """Test ExecutorConfig produces deterministic behavior."""

    def test_executor_config_hash_deterministic(self):
        """Verify ExecutorConfig.compute_hash() is deterministic."""
        config1 = ExecutorConfig(
            max_tokens=2048,
            temperature=0.0,
            timeout_s=30.0,
            retry=2,
            seed=42
        )

        config2 = ExecutorConfig(
            max_tokens=2048,
            temperature=0.0,
            timeout_s=30.0,
            retry=2,
            seed=42
        )

        hash1 = config1.compute_hash()
        hash2 = config2.compute_hash()

        assert hash1 == hash2

    def test_executor_config_hash_changes_with_params(self):
        """Verify ExecutorConfig hash changes when parameters change."""
        config1 = ExecutorConfig(seed=42)
        config2 = ExecutorConfig(seed=43)

        hash1 = config1.compute_hash()
        hash2 = config2.compute_hash()

        assert hash1 != hash2

    def test_executor_config_zero_temperature_deterministic(self):
        """Verify temperature=0.0 is documented as deterministic."""
        config = ExecutorConfig(temperature=0.0, seed=42)

        # temperature=0.0 should be deterministic per docstring
        assert config.temperature == 0.0

        # Seed should also be set for full determinism
        assert config.seed == 42

    def test_executor_config_seed_range(self):
        """Verify seed is in valid range for reproducibility."""
        # Valid seed
        config = ExecutorConfig(seed=42)
        assert 0 <= config.seed <= 2147483647

        # Boundary values
        config_min = ExecutorConfig(seed=0)
        assert config_min.seed == 0

        config_max = ExecutorConfig(seed=2147483647)
        assert config_max.seed == 2147483647

        # Out of range should raise validation error
        with pytest.raises(Exception):  # Pydantic validation error
            ExecutorConfig(seed=-1)

        with pytest.raises(Exception):  # Pydantic validation error
            ExecutorConfig(seed=2147483648)


class TestCalibrationStability:
    """Test that calibration resolution is stable."""

    def test_resolve_same_calibration_multiple_times(self):
        """Verify resolving same calibration yields identical results."""
        calib1 = resolve_calibration("BayesianEvidenceScorer", "compute_evidence_score",
strict=False)
        calib2 = resolve_calibration("BayesianEvidenceScorer", "compute_evidence_score",
strict=False)

        if calib1 is not None:
            assert calib1 == calib2
            # Verify immutability
            assert id(calib1) == id(calib2)  # Should be same frozen object

    def test_calibration_immutability(self):
        """Verify MethodCalibration is immutable (frozen dataclass)."""
        from saaaaaa.core.orchestrator.calibration_registry import MethodCalibration

        calib = MethodCalibration(
            score_min=0.0,
            score_max=1.0,
            min_evidence_snippets=3,
            max_evidence_snippets=10,
            contradiction_tolerance=0.1,
            uncertainty_penalty=0.2,
            aggregation_weight=1.0,
            sensitivity=0.8,
            requires_numeric_support=False,
            requires_temporal_support=False,
            requires_source_provenance=True,
        )

        # Frozen dataclass should not allow mutation
        with pytest.raises(Exception):  # FrozenInstanceError or AttributeError
            calib.score_min = 0.5  # type: ignore


class TestCalibrationContextDeterminism:
    """Test context-aware calibration is deterministic."""

    def test_context_resolution_deterministic(self):
        """Verify context-aware calibration resolution is deterministic."""
        from saaaaaa.core.orchestrator.calibration_registry import
resolve_calibration_with_context

        # Same inputs should yield same calibration
        calib1 = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D1Q1",
            policy_area="fiscal",
            unit_of_analysis="baseline_gap",
            method_position=0,
            total_methods=3,
        )

        calib2 = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D1Q1",
            policy_area="fiscal",
            unit_of_analysis="baseline_gap",
            method_position=0,
            total_methods=3,
        )

        if calib1 is not None:
            assert calib1 == calib2

    def test_context_changes_yield_different_calibrations(self):
        """Verify different contexts yield different calibrations."""
        from saaaaaa.core.orchestrator.calibration_registry import
resolve_calibration_with_context

        # D1 vs D9 should have different modifiers
        calib_d1 = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D1Q1",
        )

        calib_d9 = resolve_calibration_with_context(
            "BayesianEvidenceScorer",
            "compute_evidence_score",
            question_id="D9Q1",
        )

        if calib_d1 is not None and calib_d9 is not None:
            # D9 (financial) should have stricter requirements than D1
            # This is based on dimension modifiers in calibration_context.py
            assert calib_d9.min_evidence_snippets >= calib_d1.min_evidence_snippets


class TestCalibrationDocumentation:
    """Test that calibrations are properly documented."""

    def test_calibration_context_has_document_type(self):
        """Verify CalibrationContext includes document_type dimension."""
        from saaaaaa.core.orchestrator.calibration_context import CalibrationContext,
DocumentType

        context = CalibrationContext(
            question_id="D1Q1",
            dimension=1,
            question_num=1,
            document_type=DocumentType.PLAN_DESARROLLO_MUNICIPAL
        )

        assert context.document_type == DocumentType.PLAN_DESARROLLO_MUNICIPAL

    def test_document_type_enum_exists(self):
        """Verify DocumentType enum is defined."""
        from saaaaaa.core.orchestrator.calibration_context import DocumentType

        # Verify expected document types
        assert hasattr(DocumentType, "PLAN_DESARROLLO_MUNICIPAL")
        assert hasattr(DocumentType, "POLITICA_PUBLICA")
        assert hasattr(DocumentType, "UNKNOWN")

    def test_document_type_modifiers_exist(self):
        """Verify document type modifiers are defined."""
        from saaaaaa.core.orchestrator import calibration_context

        # Internal variable should exist
        assert hasattr(calibration_context, "_DOCUMENT_TYPE_MODIFIERS")

        modifiers = calibration_context._DOCUMENT_TYPE_MODIFIERS
        assert len(modifiers) > 0


# FIXME(CALIBRATION): Add integration tests once pipeline is wired
# Test that actual pipeline execution:
# - Uses ExecutorConfig for all runtime decisions
# - Produces same results with same config + seed
# - Exposes calibration_hash in artifacts
# - Blocks execution when calibration missing

===== FILE: tests/test_calibration_system.py =====
"""
Tests for Three-Pillar Calibration System

These tests validate the core calibration functionality according to
the SUPERPROMPT specification.

OBSOLETE: This test module uses the old calibration API that was refactored.
The new calibration system is tested in tests/calibration/ subdirectory.
See tests/calibration/test_gap0_complete.py for current calibration tests.
"""

import pytest

pytestmark = pytest.mark.skip(reason="obsolete - calibration API refactored, see
tests/calibration/")

# Old imports (no longer valid):
# from calibration import (
#     calibrate, CalibrationEngine, validate_config_files,
#     Context, ComputationGraph, EvidenceStore,
#     LayerType, MethodRole
# )


# Test constants
TEST_METHOD_SCORE = "src.saaaaaa.flux.phases.run_score"
TEST_METHOD_AGGREGATE = "src.saaaaaa.flux.phases.run_aggregate"
TEST_METHOD_NORMALIZE = "src.saaaaaa.flux.phases.run_normalize"


class TestConfigValidation:
    """Test configuration file validation"""

    def test_config_files_exist_and_valid(self):
        """Test that all three pillar configs exist and pass validation"""
        is_valid, errors = validate_config_files()

        if not is_valid:
            print("\nValidation errors:")
            for error in errors:
                print(f"  - {error}")

        assert is_valid, f"Config validation failed: {errors}"

    def test_intrinsic_calibration_structure(self):
        """Test intrinsic calibration config structure"""
        engine = CalibrationEngine()
        config = engine.intrinsic_config

        # Check metadata
        assert "_metadata" in config
        assert "version" in config["_metadata"]

        # Check base weights
        assert "_base_weights" in config
        weights = config["_base_weights"]
        assert "w_th" in weights
        assert "w_imp" in weights
        assert "w_dep" in weights

        # Verify normalization
        weight_sum = weights["w_th"] + weights["w_imp"] + weights["w_dep"]
        assert abs(weight_sum - 1.0) < 1e-9, f"Weights don't sum to 1.0: {weight_sum}"

        # Check methods
        assert "methods" in config
        assert len(config["methods"]) > 0

    def test_contextual_parametrization_structure(self):
        """Test contextual parametrization config structure"""
        engine = CalibrationEngine()
        config = engine.contextual_config

        # Check all layer sections exist
        assert "layer_chain" in config
        assert "layer_unit_of_analysis" in config
        assert "layer_question" in config
        assert "layer_dimension" in config
        assert "layer_policy" in config
        assert "layer_interplay" in config
        assert "layer_meta" in config

        # Check anti-universality constraint
        assert "anti_universality_constraint" in config

    def test_fusion_specification_structure(self):
        """Test fusion specification config structure"""
        engine = CalibrationEngine()
        config = engine.fusion_config

        # Check metadata
        assert "_metadata" in config
        assert "_fusion_formula" in config

        # Check role parameters
        assert "role_fusion_parameters" in config
        roles = config["role_fusion_parameters"]

        # Verify all 8 roles have parameters
        expected_roles = {
            "INGEST_PDM", "STRUCTURE", "EXTRACT", "SCORE_Q",
            "AGGREGATE", "REPORT", "META_TOOL", "TRANSFORM"
        }
        assert expected_roles.issubset(set(roles.keys()))

        # Check each role has required fields
        for role_name, params in roles.items():
            assert "required_layers" in params
            assert "linear_weights" in params
            # interaction_weights is optional


class TestDataStructures:
    """Test core data structures"""
    def test_context_creation(self):
        """Test Context dataclass"""
        ctx = Context(
            question_id="Q001",
            dimension_id="DIM01",
            policy_id="PA01",
            unit_quality=0.85
        )

        assert ctx.question_id == "Q001"
        assert ctx.dimension_id == "DIM01"
        assert ctx.policy_id == "PA01"
        assert ctx.unit_quality == 0.85

    def test_context_validation(self):
        """Test Context validation"""
        # Unit quality must be in [0,1]
        with pytest.raises(ValueError):
            Context(unit_quality=1.5)

        with pytest.raises(ValueError):
            Context(unit_quality=-0.1)

    def test_computation_graph_dag_validation(self):
        """Test computation graph DAG validation"""
        # Valid DAG
        graph = ComputationGraph(
            nodes={"A", "B", "C"},
            edges=[("A", "B"), ("B", "C")]
        )
        assert graph.validate_dag() is True

        # Cycle detection
        graph_with_cycle = ComputationGraph(
            nodes={"A", "B", "C"},
            edges=[("A", "B"), ("B", "C"), ("C", "A")]
        )
        assert graph_with_cycle.validate_dag() is False


class TestLayerComputation:
    """Test individual layer computation functions"""

    @pytest.fixture
    def engine(self):
        """Create calibration engine"""
        return CalibrationEngine()

    def test_base_layer_computation(self, engine):
        """Test base layer (@b) computation"""
        from saaaaaa.core.calibration.layer_computers import compute_base_layer

        # Use a method that exists in config
        score = compute_base_layer(TEST_METHOD_SCORE, engine.intrinsic_config)

        assert 0.0 <= score <= 1.0
        assert score > 0.0  # Should have some positive score

    def test_chain_layer_computation(self, engine):
        """Test chain layer (@chain) computation"""
        from saaaaaa.core.calibration.layer_computers import compute_chain_layer

        graph = ComputationGraph(
            nodes={"node1"},
            edges=[],
            node_signatures={"node1": {"required_inputs": []}}
        )
        score = compute_chain_layer("node1", graph, engine.contextual_config)
        assert 0.0 <= score <= 1.0

    def test_unit_layer_computation(self, engine):
        """Test unit layer (@u) computation"""
        from saaaaaa.core.calibration.layer_computers import compute_unit_layer

        # Test identity function (INGEST_PDM)
        score = compute_unit_layer(
            "test_method",
            MethodRole.INGEST_PDM,
            0.75,
            engine.contextual_config
        )
        assert abs(score - 0.75) < 1e-9  # Should return U directly

        # Test constant function (AGGREGATE)
        score = compute_unit_layer(
            "test_method",
            MethodRole.AGGREGATE,
            0.5,
            engine.contextual_config
        )
        assert abs(score - 1.0) < 1e-9  # Should return 1.0


class TestCalibrationEngine:
    """Test main calibration engine"""

    def test_calibrate_basic(self):
        """Test basic calibration flow"""
        # Create simple test setup
        ctx = Context(
            question_id="Q001",
            dimension_id="DIM01",
            policy_id="PA01",
            unit_quality=0.85
        )

        graph = ComputationGraph(
            nodes={"node1"},
            edges=[],
            node_signatures={"node1": {}}
        )

        evidence = EvidenceStore(
            runtime_metrics={"runtime_ms": 500}
        )

        # Calibrate using a method that exists in intrinsic config
        certificate = calibrate(
            method_id=TEST_METHOD_SCORE,
            node_id="node1",
            graph=graph,
            context=ctx,
            evidence_store=evidence
        )

        # Validate certificate
        assert certificate is not None
        assert certificate.method_id == TEST_METHOD_SCORE
        assert certificate.node_id == "node1"
        assert 0.0 <= certificate.calibrated_score <= 1.0
        assert 0.0 <= certificate.intrinsic_score <= 1.0

    def test_certificate_structure(self):
        """Test that certificate contains all required fields"""
        ctx = Context()
        graph = ComputationGraph(nodes={"n1"})
        evidence = EvidenceStore()

        certificate = calibrate(
            method_id="src.saaaaaa.flux.phases.run_score",
            node_id="n1",
            graph=graph,
            context=ctx,
            evidence_store=evidence
        )

        # Check required fields
        assert hasattr(certificate, 'instance_id')
        assert hasattr(certificate, 'method_id')
        assert hasattr(certificate, 'layer_scores')
        assert hasattr(certificate, 'calibrated_score')
        assert hasattr(certificate, 'fusion_formula')
        assert hasattr(certificate, 'parameter_provenance')
        assert hasattr(certificate, 'evidence_trail')
        assert hasattr(certificate, 'config_hash')
        assert hasattr(certificate, 'graph_hash')

        # Check layer scores
        assert LayerType.BASE.value in certificate.layer_scores
        assert LayerType.CHAIN.value in certificate.layer_scores
        assert LayerType.META.value in certificate.layer_scores

    def test_fusion_formula_structure(self):
        """Test fusion formula details"""
        ctx = Context()
        graph = ComputationGraph(nodes={"n1"})
        evidence = EvidenceStore()

        certificate = calibrate(
            method_id=TEST_METHOD_SCORE,
            node_id="n1",
            graph=graph,
            context=ctx,
            evidence_store=evidence
        )

        fusion = certificate.fusion_formula

        # Check structure
        assert "symbolic" in fusion
        assert "linear_terms" in fusion
        assert "interaction_terms" in fusion
        assert "linear_sum" in fusion
        assert "interaction_sum" in fusion
        assert "total" in fusion

        # Verify totals match
        assert abs(fusion["total"] - certificate.calibrated_score) < 1e-9

    def test_determinism(self):
        """Test that calibration is deterministic"""
        ctx = Context(unit_quality=0.75)
        graph = ComputationGraph(nodes={"n1"})
        evidence = EvidenceStore(runtime_metrics={"runtime_ms": 300})

        # Run calibration twice
        cert1 = calibrate(
            method_id=TEST_METHOD_SCORE,
            node_id="n1",
            graph=graph,
            context=ctx,
            evidence_store=evidence
        )

        cert2 = calibrate(
            method_id=TEST_METHOD_SCORE,
            node_id="n1",
            graph=graph,
            context=ctx,
            evidence_store=evidence
        )

        # Scores should be identical
        assert cert1.calibrated_score == cert2.calibrated_score
        assert cert1.layer_scores == cert2.layer_scores
        assert cert1.intrinsic_score == cert2.intrinsic_score

    def test_boundedness_property(self):
        """Test P1: Boundedness - Cal(I) ∈ [0,1]"""
        ctx = Context()
        graph = ComputationGraph(nodes={"n1"})
        evidence = EvidenceStore()

        # Test with different methods
        for method_id in [TEST_METHOD_SCORE, TEST_METHOD_AGGREGATE,
TEST_METHOD_NORMALIZE]:
            certificate = calibrate(
                method_id=method_id,
                node_id="n1",
                graph=graph,
                context=ctx,
                evidence_store=evidence
            )

            # All scores must be bounded
            assert 0.0 <= certificate.calibrated_score <= 1.0
            for layer, score in certificate.layer_scores.items():
                assert 0.0 <= score <= 1.0, f"Layer {layer} out of bounds: {score}"


class TestValidators:
    """Test validation functions"""

    def test_fusion_weight_validation(self):
        """Test fusion weight validation"""
        from saaaaaa.core.calibration.validators import CalibrationValidator

        validator = CalibrationValidator()

        # Valid weights
        valid_params = {
            "linear_weights": {"@b": 0.5, "@chain": 0.3},
            "interaction_weights": {"(@b, @chain)": 0.2}
        }
        is_valid, errors = validator.validate_fusion_weights(valid_params, "TEST")
        assert is_valid

        # Invalid: don't sum to 1.0
        invalid_params = {
            "linear_weights": {"@b": 0.5, "@chain": 0.3},
            "interaction_weights": {"(@b, @chain)": 0.1}
        }
        is_valid, errors = validator.validate_fusion_weights(invalid_params, "TEST")
        assert not is_valid
        assert len(errors) > 0

    def test_boundedness_validation(self):
        """Test boundedness validation"""
        from saaaaaa.core.calibration.validators import CalibrationValidator
        validator = CalibrationValidator()

        # Valid scores
        valid_layers = {"@b": 0.5, "@chain": 0.8}
        valid_cal = 0.75
        is_valid, errors = validator.validate_boundedness(valid_layers, valid_cal)
        assert is_valid

        # Invalid: out of bounds
        invalid_layers = {"@b": 1.5, "@chain": 0.8}
        is_valid, errors = validator.validate_boundedness(invalid_layers, 0.75)
        assert not is_valid


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_calibration_system_regression.py =====
"""
FASE 5.6: Comprehensive regression test suite.

This test verifies that the calibration system works correctly after
all Phase 1-5 changes, with NO behavioral regressions.

Test Coverage:
1. All 1,995 methods load correctly
2. All 30 executors calibrate with 8 layers
3. Weights correctly loaded from JSON (0.4, 0.35, 0.25)
4. Quality thresholds configurable
5. Layer requirements work for all role types
6. No hardcoded critical values in use
7. End-to-end calibration produces valid scores
"""
import sys
from pathlib import Path

# Add project root to path
sys.path.insert(0, str(Path(__file__).parent.parent))

from src.saaaaaa.core.calibration.orchestrator import CalibrationOrchestrator
from src.saaaaaa.core.calibration.intrinsic_loader import IntrinsicScoreLoader
from src.saaaaaa.core.calibration.layer_requirements import LayerRequirementsResolver
from src.saaaaaa.core.calibration.parameter_loader import MethodParameterLoader
from src.saaaaaa.core.calibration.base_layer import BaseLayerEvaluator
from src.saaaaaa.core.calibration.data_structures import ContextTuple, LayerID
from src.saaaaaa.core.calibration.pdt_structure import PDTStructure


def test_intrinsic_loader_functional():
    """Verify IntrinsicScoreLoader works correctly."""
    print("=" * 80)
    print("TEST 1: IntrinsicScoreLoader Functional")
    print("=" * 80)
    print()

    loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")

    # Check basic functionality
    stats = loader.get_statistics()

    print(f"Total methods: {stats['total']}")
    print(f"Computed: {stats['computed']}")
    print(f"Excluded: {stats['excluded']}")
    print()

    # Verify expected counts
    if stats['total'] != 1995:
        print(f"✗ Expected 1995 methods, got {stats['total']}")
        return False

    if stats['computed'] != 1467:
        print(f"✗ Expected 1467 computed, got {stats['computed']}")
        return False

    print("✓ Method counts correct")

    # Verify weights loaded
    if loader.w_theory != 0.4 or loader.w_impl != 0.35 or loader.w_deploy != 0.25:
        print(f"✗ Weights incorrect: {loader.w_theory}, {loader.w_impl},
{loader.w_deploy}")
        return False

    print("✓ Weights correct (0.4, 0.35, 0.25)")

    # Test score retrieval
    score = loader.get_score("orchestrator.__init__.__getattr__")
    if not (0.0 <= score <= 1.0):
        print(f"✗ Invalid score: {score}")
        return False

    print(f"✓ Score retrieval works ({score:.3f})")

    return True


def test_layer_requirements_functional():
    """Verify LayerRequirementsResolver works correctly."""
    print("=" * 80)
    print("TEST 2: LayerRequirementsResolver Functional")
    print("=" * 80)
    print()

    loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")
    resolver = LayerRequirementsResolver(loader)

    # Test executor detection
    is_exec =
resolver.is_executor("src.saaaaaa.core.orchestrator.executors.D1Q1_Executor.execute")
    if not is_exec:
        print("✗ Executor not detected")
        return False
    print("✓ Executor detection works")

    # Test executor gets 8 layers
    layers = resolver.get_required_layers("src.saaaaaa.core.orchestrator.executors.D1Q1_Ex
ecutor.execute")
    if len(layers) != 8:
        print(f"✗ Executor has {len(layers)} layers, expected 8")
        return False
    print("✓ Executor gets 8 layers")

    # Test utility gets 3 layers
    util_layers =
resolver.get_required_layers("src.saaaaaa.utils.adapters._deprecation_warning")
    if len(util_layers) != 3:
        print(f"✗ Utility has {len(util_layers)} layers, expected 3")
        return False
    print("✓ Utility gets 3 layers")

    # Test analyzer gets 8 layers
    analyzer_layers =
resolver.get_required_layers("src.saaaaaa.analysis.Analyzer_one.BatchProcessor.__init__")
    if len(analyzer_layers) != 8:
        print(f"✗ Analyzer has {len(analyzer_layers)} layers, expected 8")
        return False
    print("✓ Analyzer gets 8 layers")
    return True


def test_base_layer_weights():
    """Verify BaseLayerEvaluator uses correct weights."""
    print("=" * 80)
    print("TEST 3: BaseLayerEvaluator Weights")
    print("=" * 80)
    print()

    evaluator = BaseLayerEvaluator("config/intrinsic_calibration.json")

    # Check weights
    if evaluator.theory_weight != 0.4:
        print(f"✗ theory_weight = {evaluator.theory_weight}, expected 0.4")
        return False

    if evaluator.impl_weight != 0.35:
        print(f"✗ impl_weight = {evaluator.impl_weight}, expected 0.35")
        return False

    if evaluator.deploy_weight != 0.25:
        print(f"✗ deploy_weight = {evaluator.deploy_weight}, expected 0.25")
        return False

    print("✓ Weights correct (0.4, 0.35, 0.25)")

    # Verify score consistency with IntrinsicScoreLoader
    loader = IntrinsicScoreLoader("config/intrinsic_calibration.json")

    method_id = "orchestrator.__init__.__getattr__"
    base_score = evaluator.evaluate(method_id).score
    intrinsic_score = loader.get_score(method_id)

    if abs(base_score - intrinsic_score) > 0.001:
        print(f"✗ Score mismatch: base={base_score:.6f}, intrinsic={intrinsic_score:.6f}")
        return False

    print(f"✓ Scores consistent ({base_score:.3f})")

    return True


def test_parameter_loader_functional():
    """Verify MethodParameterLoader works correctly."""
    print("=" * 80)
    print("TEST 4: MethodParameterLoader Functional")
    print("=" * 80)
    print()

    loader = MethodParameterLoader("config/method_parameters.json")

    # Test quality thresholds
    quality = loader.get_quality_thresholds()
    if quality['excellent'] != 0.85 or quality['good'] != 0.70:
        print(f"✗ Quality thresholds incorrect: {quality}")
        return False
    print("✓ Overall quality thresholds correct")

    # Test base layer quality thresholds
    base_quality = loader.get_base_layer_quality_thresholds()
    if base_quality['excellent'] != 0.8 or base_quality['good'] != 0.6:
        print(f"✗ Base layer thresholds incorrect: {base_quality}")
        return False
    print("✓ Base layer quality thresholds correct")

    # Test executor threshold
    threshold = loader.get_executor_threshold("D4Q1_Executor")
    if threshold != 0.80:
        print(f"✗ D4Q1 threshold = {threshold}, expected 0.80")
        return False
    print("✓ Executor thresholds load correctly")

    # Test role-based threshold
    analyzer_threshold = loader.get_validation_threshold_for_role("analyzer")
    if analyzer_threshold != 0.70:
        print(f"✗ Analyzer threshold = {analyzer_threshold}, expected 0.70")
        return False
    print("✓ Role-based thresholds load correctly")

    return True


def test_executor_calibration():
    """Verify all 30 executors calibrate with 8 layers."""
    print("=" * 80)
    print("TEST 5: All 30 Executors Calibrate with 8 Layers")
    print("=" * 80)
    print()

    orchestrator = CalibrationOrchestrator(
        intrinsic_calibration_path="config/intrinsic_calibration.json"
    )

    # Create test context and PDT
    context = ContextTuple(
        question_id="Q001",
        dimension="DIM01",
        policy_area="PA01",
        unit_quality=0.75
    )

    pdt = PDTStructure(
        full_text="Plan de Desarrollo Territorial 2024-2027.",
        total_tokens=1000,
        blocks_found={
            "Diagnóstico": {"text": "Test", "tokens": 250, "numbers_count": 5},
            "Parte Estratégica": {"text": "Test", "tokens": 250, "numbers_count": 10},
            "PPI": {"text": "Test", "tokens": 250, "numbers_count": 50},
            "Seguimiento": {"text": "Test", "tokens": 250, "numbers_count": 15}
        },
        indicator_matrix_present=True,
        ppi_matrix_present=True
    )

    # Sample 5 executors
    test_executors = [
        "src.saaaaaa.core.orchestrator.executors.D1Q1_Executor.execute",
        "src.saaaaaa.core.orchestrator.executors.D2Q3_Executor.execute",
        "src.saaaaaa.core.orchestrator.executors.D4Q2_Executor.execute",
        "src.saaaaaa.core.orchestrator.executors.D5Q4_Executor.execute",
        "src.saaaaaa.core.orchestrator.executors.D6Q5_Executor.execute",
    ]

    print(f"Testing {len(test_executors)} executors (sample)...")

    for executor_id in test_executors:
        result = orchestrator.calibrate(
            method_id=executor_id,
            method_version="1.0.0",
            context=context,
            pdt_structure=pdt
        )

        if len(result.layer_scores) != 8:
            print(f"✗ {executor_id} has {len(result.layer_scores)} layers, expected 8")
            return False

        # Verify all 8 layer IDs present
        expected_layers = {
            LayerID.BASE, LayerID.UNIT, LayerID.QUESTION,
            LayerID.DIMENSION, LayerID.POLICY, LayerID.CONGRUENCE,
            LayerID.CHAIN, LayerID.META
        }
        actual_layers = set(result.layer_scores.keys())

        if expected_layers != actual_layers:
            print(f"✗ {executor_id} missing layers: {expected_layers - actual_layers}")
            return False

    print(f"✓ All {len(test_executors)} sampled executors have 8 layers")
    print("✓ All layer IDs present")

    return True


def test_no_hardcoded_critical_values():
    """Verify no critical hardcoded values in use."""
    print("=" * 80)
    print("TEST 6: No Critical Hardcoded Values")
    print("=" * 80)
    print()

    # Test 1: BaseLayerEvaluator doesn't use old weights (0.4, 0.4, 0.2)
    evaluator = BaseLayerEvaluator("config/intrinsic_calibration.json")

    old_weights = (0.4, 0.4, 0.2)
    actual_weights = (evaluator.theory_weight, evaluator.impl_weight,
evaluator.deploy_weight)

    if actual_weights == old_weights:
        print("✗ CRITICAL: Still using old hardcoded weights!")
        return False

    print("✓ Not using old hardcoded weights")

    # Test 2: Weights sum to 1.0
    weight_sum = sum(actual_weights)
    if abs(weight_sum - 1.0) > 1e-6:
        print(f"✗ Weights don't sum to 1.0: {weight_sum}")
        return False

    print("✓ Weights sum to 1.0")

    # Test 3: Quality thresholds loaded from config
    if not hasattr(evaluator, 'excellent_threshold'):
        print("✗ Quality thresholds not loaded as instance variables")
        return False

    print("✓ Quality thresholds are instance variables")

    return True


def test_end_to_end_calibration():
    """Verify end-to-end calibration produces valid results."""
    print("=" * 80)
    print("TEST 7: End-to-End Calibration")
    print("=" * 80)
    print()

    orchestrator = CalibrationOrchestrator(
        intrinsic_calibration_path="config/intrinsic_calibration.json"
    )

    context = ContextTuple(
        question_id="Q001",
        dimension="DIM01",
        policy_area="PA01",
        unit_quality=0.75
    )

    pdt = PDTStructure(
        full_text="Plan de Desarrollo Territorial 2024-2027.",
        total_tokens=1000,
        blocks_found={
            "Diagnóstico": {"text": "Test", "tokens": 250, "numbers_count": 5},
            "Parte Estratégica": {"text": "Test", "tokens": 250, "numbers_count": 10},
            "PPI": {"text": "Test", "tokens": 250, "numbers_count": 50},
            "Seguimiento": {"text": "Test", "tokens": 250, "numbers_count": 15}
        },
        indicator_matrix_present=True,
        ppi_matrix_present=True
    )

    # Calibrate one executor
    result = orchestrator.calibrate(
        method_id="src.saaaaaa.core.orchestrator.executors.D1Q1_Executor.execute",
        method_version="1.0.0",
        context=context,
        pdt_structure=pdt
    )

    # Verify result structure
    if not hasattr(result, 'final_score'):
        print("✗ Result missing final_score")
        return False

    if not hasattr(result, 'layer_scores'):
        print("✗ Result missing layer_scores")
        return False

    print("✓ Result structure valid")

    # Verify final score in valid range
    if not (0.0 <= result.final_score <= 1.0):
        print(f"✗ Invalid final score: {result.final_score}")
        return False

    print(f"✓ Final score valid ({result.final_score:.3f})")

    # Verify all layer scores in valid range
    for layer_id, layer_score in result.layer_scores.items():
        if not (0.0 <= layer_score.score <= 1.0):
            print(f"✗ Invalid layer score for {layer_id}: {layer_score.score}")
            return False

    print("✓ All layer scores valid")

    return True


if __name__ == "__main__":
    print("\nFASE 5.6: COMPREHENSIVE REGRESSION TEST SUITE")
    print("=" * 80)
    print()

    # Run all tests
    test1 = test_intrinsic_loader_functional()
    print()
    test2 = test_layer_requirements_functional()
    print()
    test3 = test_base_layer_weights()
    print()
    test4 = test_parameter_loader_functional()
    print()
    test5 = test_executor_calibration()
    print()
    test6 = test_no_hardcoded_critical_values()
    print()
    test7 = test_end_to_end_calibration()

    # Summary
    print()
    print("=" * 80)
    print("FINAL RESULTS - REGRESSION TEST SUITE")
    print("=" * 80)
    print(f"IntrinsicScoreLoader: {'✓ PASS' if test1 else '✗ FAIL'}")
    print(f"LayerRequirementsResolver: {'✓ PASS' if test2 else '✗ FAIL'}")
    print(f"BaseLayerEvaluator Weights: {'✓ PASS' if test3 else '✗ FAIL'}")
    print(f"MethodParameterLoader: {'✓ PASS' if test4 else '✗ FAIL'}")
    print(f"Executor Calibration: {'✓ PASS' if test5 else '✗ FAIL'}")
    print(f"No Hardcoded Values: {'✓ PASS' if test6 else '✗ FAIL'}")
    print(f"End-to-End Calibration: {'✓ PASS' if test7 else '✗ FAIL'}")
    print()

    if all([test1, test2, test3, test4, test5, test6, test7]):
        print("🎉 ALL REGRESSION TESTS PASSED!")
        print()
        print("✓ System behavior UNCHANGED after Phase 1-5 modifications")
        print("✓ All critical values now loaded from config")
        print("✓ No hardcoded weights or thresholds in use")
        print("✓ 1,995 methods load correctly")
        print("✓ 30 executors calibrate with 8 layers")
        print("✓ Scores computed with correct weights (0.4, 0.35, 0.25)")
        print()
        sys.exit(0)
    else:
        print("⚠  SOME REGRESSION TESTS FAILED")
        print("System behavior may have changed!")
        sys.exit(1)

===== FILE: tests/test_chunk_execution.py =====
"""
Integration tests for chunk-aware execution.

Tests the SPC exploitation features including chunk routing,
chunk-scoped execution, and verification that chunk mode
produces equivalent results to flat mode.
"""

import pytest

# Import the components we're testing
try:
    from saaaaaa.core.orchestrator.core import ChunkData, PreprocessedDocument
    from saaaaaa.core.orchestrator.chunk_router import ChunkRouter
    HAS_IMPORTS = True
except ImportError:
    HAS_IMPORTS = False
    pytest.skip("Required modules not available", allow_module_level=True)


class TestChunkRouting:
    """Test chunk routing functionality."""

    def test_chunk_router_initialization(self):
        """Test that ChunkRouter initializes correctly."""
        router = ChunkRouter()
        assert router is not None
        assert hasattr(router, 'ROUTING_TABLE')
        assert isinstance(router.ROUTING_TABLE, dict)

    def test_chunk_routing_coverage(self):
        """Verify all chunk types have executors mapped."""
        router = ChunkRouter()

        chunk_types = ["diagnostic", "activity", "indicator", "resource", "temporal",
"entity"]

        for chunk_type in chunk_types:
            mock_chunk = ChunkData(
                id=0,
                text="test chunk",
                chunk_type=chunk_type,
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=100,
                confidence=0.9,
            )

            route = router.route_chunk(mock_chunk)

            # Each chunk type should have at least one executor
            assert route.executor_class != "", f"No executor for chunk type {chunk_type}"
            assert route.skip_reason is None, f"Chunk type {chunk_type} skipped:
{route.skip_reason}"
            assert route.chunk_type == chunk_type

    def test_chunk_routing_unknown_type(self):
        """Test router behavior for unknown chunk type."""
        router = ChunkRouter()

        # Test get_relevant_executors with an unknown type
        executors = router.get_relevant_executors("unknown_type")
        assert executors == []

        # We can't create a ChunkData with invalid chunk_type due to Literal type
constraint
        # So we test the router's behavior directly with get_relevant_executors

    def test_get_relevant_executors(self):
        """Test getting relevant executors for a chunk type."""
        router = ChunkRouter()

        # Diagnostic chunks should map to baseline/gap executors
        executors = router.get_relevant_executors("diagnostic")
        assert len(executors) > 0
        assert "D1Q1" in executors or "D1Q2" in executors

        # Activity chunks should map to intervention executors
        executors = router.get_relevant_executors("activity")
        assert len(executors) > 0
        assert any(ex.startswith("D2") for ex in executors)


class TestChunkDataStructure:
    """Test ChunkData and PreprocessedDocument structures."""

    def test_chunk_data_creation(self):
        """Test creating ChunkData objects."""
        chunk = ChunkData(
            id=1,
            text="Test chunk text",
            chunk_type="diagnostic",
            sentences=[0, 1, 2],
            tables=[0],
            start_pos=0,
            end_pos=50,
            confidence=0.85,
            edges_out=[2, 3],
            edges_in=[0],
        )

        assert chunk.id == 1
        assert chunk.chunk_type == "diagnostic"
        assert len(chunk.sentences) == 3
        assert len(chunk.edges_out) == 2
        assert chunk.confidence == 0.85

    def test_preprocessed_document_chunked_mode(self):
        """Test PreprocessedDocument in chunked mode."""
        chunks = [
            ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[i],
                tables=[],
                start_pos=i*10,
                end_pos=(i+1)*10,
                confidence=0.9,
            )
            for i in range(3)
        ]

        doc = PreprocessedDocument(
            document_id="test_doc",
            raw_text="Test document text",
            sentences=[{"text": "Sentence 1"}, {"text": "Sentence 2"}],
            tables=[],
            metadata={},
            chunks=chunks,
            chunk_index={"sent_0": 0, "sent_1": 1},
            chunk_graph={"nodes": [], "edges": []},
            processing_mode="chunked",
        )

        assert doc.processing_mode == "chunked"
        assert len(doc.chunks) == 3
        assert doc.chunk_index["sent_0"] == 0
        assert doc.chunk_graph is not None

    def test_preprocessed_document_flat_mode_compatibility(self):
        """Test that PreprocessedDocument still works in flat mode."""
        doc = PreprocessedDocument(
            document_id="test_doc",
            raw_text="Test document text",
            sentences=[{"text": "Sentence 1"}],
            tables=[],
            metadata={},
        )

        # Default mode should be flat
        assert doc.processing_mode == "flat"
        assert len(doc.chunks) == 0
        assert len(doc.chunk_index) == 0


class TestSPCCausalBridge:
    """Test SPC causal bridge functionality."""

    def test_spc_causal_bridge_initialization(self):
        """Test SPCCausalBridge initializes correctly."""
        try:
            from saaaaaa.analysis.spc_causal_bridge import SPCCausalBridge

            bridge = SPCCausalBridge()
            assert bridge is not None
            assert hasattr(bridge, 'CAUSAL_WEIGHTS')
            assert bridge.CAUSAL_WEIGHTS["dependency"] >
bridge.CAUSAL_WEIGHTS["sequential"]
        except ImportError:
            pytest.skip("SPCCausalBridge not available")

    def test_causal_weight_mapping(self):
        """Test that edge types map to appropriate causal weights."""
        try:
            from saaaaaa.analysis.spc_causal_bridge import SPCCausalBridge

            bridge = SPCCausalBridge()

            # Strong causal relationships should have high weights
            assert bridge._compute_causal_weight("dependency") > 0.8
            assert bridge._compute_causal_weight("hierarchical") > 0.6

            # Weak causal relationships should have lower weights
            assert bridge._compute_causal_weight("sequential") < 0.5

            # Unknown types should return 0
            assert bridge._compute_causal_weight("unknown_type") == 0.0
        except ImportError:
            pytest.skip("SPCCausalBridge not available")

    def test_build_causal_graph_from_spc(self):
        """Test building causal graph from chunk graph."""
        try:
            from saaaaaa.analysis.spc_causal_bridge import SPCCausalBridge
            import networkx as nx

            bridge = SPCCausalBridge()

            chunk_graph = {
                "nodes": [
                    {"id": 0, "type": "diagnostic", "text": "Node 0", "confidence": 0.9},
                    {"id": 1, "type": "activity", "text": "Node 1", "confidence": 0.8},
                    {"id": 2, "type": "indicator", "text": "Node 2", "confidence": 0.85},
                ],
                "edges": [
                    {"source": 0, "target": 1, "type": "sequential"},
                    {"source": 1, "target": 2, "type": "dependency"},
                ],
            }

            G = bridge.build_causal_graph_from_spc(chunk_graph)

            assert G is not None
            assert G.number_of_nodes() == 3
            assert G.number_of_edges() == 2
            assert nx.is_directed_acyclic_graph(G)

        except ImportError:
            pytest.skip("NetworkX or SPCCausalBridge not available")


class TestChunkMetricsCalculation:
    """Test chunk metrics calculation for verification manifest."""

    def test_chunk_metrics_for_chunked_mode(self):
        """Test that chunk metrics are calculated correctly."""
        # This would require running the actual pipeline
        # For now, we just verify the structure
        chunks = [
            ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type=["diagnostic", "activity", "indicator"][i % 3],
                sentences=[i],
                tables=[],
                start_pos=i*10,
                end_pos=(i+1)*10,
                confidence=0.9,
            )
            for i in range(6)
        ]

        # Verify chunk type distribution
        chunk_types = {}
        for chunk in chunks:
            chunk_types[chunk.chunk_type] = chunk_types.get(chunk.chunk_type, 0) + 1

        assert chunk_types["diagnostic"] == 2
        assert chunk_types["activity"] == 2
        assert chunk_types["indicator"] == 2


# Integration test markers
@pytest.mark.integration
class TestChunkVsFlatEquivalence:
    """
    Test that chunk mode produces equivalent results to flat mode.

    Note: This would require running the full pipeline with real data.
    For now, we document the expected behavior.
    """

    def test_structure_preservation(self):
        """
        Verify that chunk mode preserves semantic structure.

        This test would:
        1. Run pipeline on same input in both modes
        2. Compare macro scores (should be within 5% relative tolerance)
        3. Verify chunk mode executes fewer operations
        4. Confirm chunk mode maintains quality
        """
        pytest.skip("Requires full pipeline execution with real data")


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_circuit_breaker_async.py =====
"""Test async circuit breaker safety in executors.py."""
import asyncio
import pytest


@pytest.mark.asyncio
async def test_circuit_breaker_increment_failures():
    """Test that circuit breaker increments failures safely."""
    from saaaaaa.core.orchestrator.executors import CircuitBreakerState

    cb = CircuitBreakerState()

    assert cb.failures == 0
    assert not await cb.is_open()

    await cb.increment_failures()
    assert cb.failures == 1
    assert not await cb.is_open()

    await cb.increment_failures()
    assert cb.failures == 2
    assert not await cb.is_open()

    await cb.increment_failures()
    assert cb.failures == 3
    assert await cb.is_open()


@pytest.mark.asyncio
async def test_circuit_breaker_concurrent_increments():
    """Test that circuit breaker handles concurrent increments correctly."""
    from saaaaaa.core.orchestrator.executors import CircuitBreakerState

    cb = CircuitBreakerState()

    # Run 10 concurrent increment operations
    tasks = [cb.increment_failures() for _ in range(10)]
    await asyncio.gather(*tasks)

    # Should have 10 failures
    assert cb.failures == 10
    assert await cb.is_open()


@pytest.mark.asyncio
async def test_circuit_breaker_reset():
    """Test that circuit breaker reset works correctly."""
    from saaaaaa.core.orchestrator.executors import CircuitBreakerState

    cb = CircuitBreakerState()

    # Increment to open circuit
    await cb.increment_failures()
    await cb.increment_failures()
    await cb.increment_failures()

    assert await cb.is_open()

    # Reset
    await cb.reset()

    assert cb.failures == 0
    assert not await cb.is_open()


@pytest.mark.asyncio
async def test_circuit_breaker_race_condition():
    """Test that circuit breaker is safe under race conditions."""
    from saaaaaa.core.orchestrator.executors import CircuitBreakerState

    cb = CircuitBreakerState()

    async def increment_multiple():
        for _ in range(100):
            await cb.increment_failures()

    # Run multiple tasks concurrently
    tasks = [increment_multiple() for _ in range(10)]
    await asyncio.gather(*tasks)

    # Should have exactly 1000 failures
    assert cb.failures == 1000
    assert await cb.is_open()
```

```
===== FILE: tests/test_circuit_breaker_stress.py =====
"""Stress tests for circuit breaker under high concurrency."""

import asyncio
import time
from concurrent.futures import ThreadPoolExecutor
from unittest.mock import Mock, patch
import pytest

# Add src to path for imports
import sys
from pathlib import Path

from saaaaaa.core.orchestrator.signals import (
    SignalClient,
    SignalPack,
    CircuitBreakerError,
    SignalUnavailableError,
)


def test_circuit_breaker_basic_state_tracking():
    """Test that circuit breaker tracks state changes."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=3,
    )

    # Initially no state changes
    assert len(client.get_state_history()) == 0
    assert client.get_metrics()["circuit_open"] is False
    assert client.get_metrics()["failure_count"] == 0

    # Simulate failures by calling _record_failure
    client._record_failure()
    assert client.get_metrics()["failure_count"] == 1
    assert client.get_metrics()["circuit_open"] is False

    client._record_failure()
    assert client.get_metrics()["failure_count"] == 2
    assert client.get_metrics()["circuit_open"] is False

    # Third failure should open circuit
    client._record_failure()
    assert client.get_metrics()["failure_count"] == 3
    assert client.get_metrics()["circuit_open"] is True

    # Should have one state change (closed -> open)
    history = client.get_state_history()
    assert len(history) == 1
    assert history[0]["from_open"] is False
    assert history[0]["to_open"] is True
    assert history[0]["failures"] == 3


def test_circuit_breaker_state_history_trimming():
    """Test that state history is trimmed to max_history."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=1,
    )

    # Force many state changes by opening/closing circuit
    for i in range(150):
        # Open circuit
        client._record_failure()

        # Manually close it to create more state changes
        client._circuit_open = False
        client._failure_count = 0

    # Should be trimmed to max_history (100)
    history = client.get_state_history()
    assert len(history) <= 100


def test_circuit_breaker_metrics_after_state_changes():
    """Test that metrics reflect state changes correctly."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=2,
    )

    # Record failures
    client._record_failure()
    metrics1 = client.get_metrics()
    assert metrics1["failure_count"] == 1
    assert metrics1["state_change_count"] == 0

    client._record_failure()
    metrics2 = client.get_metrics()
    assert metrics2["failure_count"] == 2
    assert metrics2["state_change_count"] == 1  # Circuit opened
    assert metrics2["circuit_open"] is True
    assert metrics2["last_failure_time"] is not None


@pytest.mark.asyncio
async def test_circuit_breaker_high_concurrency():
    """Stress test: Multiple concurrent failure recordings."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=100,
    )

    async def record_failures(count: int):
        """Record multiple failures sequentially."""
        for _ in range(count):
            client._record_failure()
            await asyncio.sleep(0.001)  # Simulate work

    # Run 10 concurrent tasks, each recording 10 failures
    tasks = [asyncio.create_task(record_failures(10)) for _ in range(10)]
    await asyncio.gather(*tasks)

    # Should have exactly 100 failures
    assert client.get_metrics()["failure_count"] == 100
    assert client.get_metrics()["circuit_open"] is True

    # Should have one state change (when threshold reached)
    history = client.get_state_history()
    assert len(history) == 1
    assert history[0]["to_open"] is True


@pytest.mark.asyncio
async def test_circuit_breaker_race_condition():
    """Test for race condition when crossing threshold."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=10,
    )

    async def increment_failures():
        """Increment failure count."""
        for _ in range(5):
            client._record_failure()
            await asyncio.sleep(0.001)

    # Start multiple tasks simultaneously
    tasks = [asyncio.create_task(increment_failures()) for _ in range(3)]
    await asyncio.gather(*tasks)

    # Should be exactly 15 failures (3 * 5)
    assert client.get_metrics()["failure_count"] == 15
    assert client.get_metrics()["circuit_open"] is True


def test_circuit_breaker_thread_safety():
    """Test thread safety with ThreadPoolExecutor."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=200,
    )

    def record_failure():
        """Record a single failure."""
        client._record_failure()

    with ThreadPoolExecutor(max_workers=20) as executor:
        futures = [executor.submit(record_failure) for _ in range(100)]
        for future in futures:
            future.result()

    # Should have exactly 100 failures
    assert client.get_metrics()["failure_count"] == 100
    assert client.get_metrics()["circuit_open"] is False  # Below threshold


def test_circuit_breaker_cooldown_and_reset():
    """Test circuit breaker cooldown and reset behavior."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=2,
        circuit_breaker_cooldown_s=0.1,  # Short cooldown for testing
    )

    # Open circuit
    client._record_failure()
    client._record_failure()
    assert client.get_metrics()["circuit_open"] is True

    initial_changes = len(client.get_state_history())

    # Wait for cooldown
    time.sleep(0.15)

    # Simulate successful request by resetting
    old_open = client._circuit_open
    client._circuit_open = False
    client._failure_count = 0

    # Manually record state change (simulating what _fetch_from_http does)
    client._state_changes.append({
        'timestamp': time.time(),
        'from_open': old_open,
        'to_open': False,
        'failures': 0,
    })

    # Should have one more state change
    assert len(client.get_state_history()) == initial_changes + 1
    assert client.get_metrics()["circuit_open"] is False
    assert client.get_metrics()["failure_count"] == 0
```

```python
@pytest.mark.asyncio
async def test_circuit_breaker_under_sustained_load():
    """Test circuit breaker behavior under sustained concurrent load."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=50,
    )

    async def sustained_failures(duration_s: float):
        """Record failures for a duration."""
        end_time = time.time() + duration_s
        count = 0
        while time.time() < end_time:
            client._record_failure()
            count += 1
            await asyncio.sleep(0.01)
        return count

    # Run for 0.5 seconds with 5 concurrent workers
    tasks = [asyncio.create_task(sustained_failures(0.5)) for _ in range(5)]
    results = await asyncio.gather(*tasks)

    total_failures = sum(results)
    assert client.get_metrics()["failure_count"] == total_failures
    assert client.get_metrics()["circuit_open"] is True  # Should exceed threshold


def test_circuit_breaker_state_history_timestamps():
    """Test that state history includes proper timestamps."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=2,
    )

    start_time = time.time()

    client._record_failure()
    client._record_failure()  # Opens circuit

    history = client.get_state_history()
    assert len(history) == 1

    change = history[0]
    assert "timestamp" in change
    assert "from_open" in change
    assert "to_open" in change
    assert "failures" in change

    # Timestamp should be recent
    assert change["timestamp"] >= start_time
    assert change["timestamp"] <= time.time()
    assert change["failures"] == 2


@pytest.mark.asyncio
async def test_circuit_breaker_many_state_changes():
    """Test multiple open/close cycles."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=2,
        circuit_breaker_cooldown_s=0.05,
    )

    for cycle in range(5):
        # Open circuit
        client._record_failure()
        client._record_failure()
        assert client.get_metrics()["circuit_open"] is True

        # Wait for cooldown
        await asyncio.sleep(0.06)

        # Manually close (simulating successful request after cooldown)
        old_open = client._circuit_open
        client._circuit_open = False
        client._failure_count = 0
        client._state_changes.append({
            'timestamp': time.time(),
            'from_open': old_open,
            'to_open': False,
            'failures': 0,
        })

    # Should have 10 state changes (5 opens + 5 closes)
    history = client.get_state_history()
    assert len(history) == 10

    # Verify alternating pattern
    for i, change in enumerate(history):
        if i % 2 == 0:
            # Even indices should be open transitions
            assert change["from_open"] is False
            assert change["to_open"] is True
        else:
            # Odd indices should be close transitions
            assert change["from_open"] is True
            assert change["to_open"] is False


def test_circuit_breaker_state_after_many_failures():
    """Test circuit breaker state after many failures beyond threshold."""
    client = SignalClient(
        base_url="memory://",
        circuit_breaker_threshold=5,
    )

    # Record way more failures than threshold
    for _ in range(50):
        client._record_failure()

    # Should still only have one state change (when threshold was crossed)
    history = client.get_state_history()
    assert len(history) == 1
    assert history[0]["failures"] == 5  # Recorded when threshold was reached

    # But failure count should be 50
    assert client.get_metrics()["failure_count"] == 50
    assert client.get_metrics()["circuit_open"] is True

===== FILE: tests/test_class_registry_paths.py =====
"""Test that class registry paths are correctly configured.

This test validates that the import path fix described in the problem statement
has been correctly applied. It checks that all 22 classes have the proper
saaaaaa. prefix in their import paths.
"""

import pytest


# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="Registry system refactored")

def test_class_registry_paths_have_saaaaaa_prefix():
    """Verify all class paths use absolute imports with saaaaaa. prefix."""
    from saaaaaa.core.orchestrator.class_registry import get_class_paths

    paths = get_class_paths()

    # All paths should start with "saaaaaa."
    for class_name, import_path in paths.items():
        assert import_path.startswith("saaaaaa."), \
            f"{class_name} has invalid path: {import_path} (should start with 'saaaaaa.')"

def test_class_registry_has_all_expected_classes():
    """Verify all 22 classes from problem statement are registered."""
    from saaaaaa.core.orchestrator.class_registry import get_class_paths

    paths = get_class_paths()

    # Expected classes by category (from problem statement)
    expected_classes = {
        # Derek Beach (5 classes)
        "CDAFFramework",
        "CausalExtractor",
        "OperationalizationAuditor",
        "FinancialAuditor",
        "BayesianMechanismInference",

        # Contradiction Detection (3 classes)
        "PolicyContradictionDetector",
        "TemporalLogicVerifier",
        "BayesianConfidenceCalculator",

        # Analyzer_one (4 classes)
        "SemanticAnalyzer",
        "PerformanceAnalyzer",
        "TextMiningEngine",
        "MunicipalOntology",

        # Theory of Change (2 classes)
        "TeoriaCambio",
        "AdvancedDAGValidator",

        # Embedding Policy (3 classes + 1 alias)
        "BayesianNumericalAnalyzer",
        "PolicyAnalysisEmbedder",
        "AdvancedSemanticChunker",
        "SemanticChunker",  # Alias

        # Financial Analysis (1 class)
        "PDETMunicipalPlanAnalyzer",

        # Policy Processor (3 classes)
        "IndustrialPolicyProcessor",
        "PolicyTextProcessor",
        "BayesianEvidenceScorer",
    }

    # Check all expected classes are present
    actual_classes = set(paths.keys())
    missing_classes = expected_classes - actual_classes
    extra_classes = actual_classes - expected_classes

    assert not missing_classes, f"Missing classes: {missing_classes}"
    assert not extra_classes, f"Unexpected classes: {extra_classes}"
    assert len(paths) == 22, f"Expected 22 classes, got {len(paths)}"

def test_class_registry_paths_match_expected_modules():
    """Verify classes are mapped to the correct analysis/processing modules."""
    from saaaaaa.core.orchestrator.class_registry import get_class_paths
    paths = get_class_paths()

    # Derek Beach classes should be in saaaaaa.analysis.derek_beach
    derek_beach_classes = [
        "CDAFFramework", "CausalExtractor", "OperationalizationAuditor",
        "FinancialAuditor", "BayesianMechanismInference"
    ]
    for class_name in derek_beach_classes:
        assert paths[class_name].startswith("saaaaaa.analysis.derek_beach."), \
            f"{class_name} should be in saaaaaa.analysis.derek_beach"

    # Contradiction detection classes should be in
saaaaaa.analysis.contradiction_deteccion
    contradiction_classes = [
        "PolicyContradictionDetector", "TemporalLogicVerifier",
"BayesianConfidenceCalculator"
    ]
    for class_name in contradiction_classes:
        assert paths[class_name].startswith("saaaaaa.analysis.contradiction_deteccion."),
\
            f"{class_name} should be in saaaaaa.analysis.contradiction_deteccion"

    # Analyzer_one classes should be in saaaaaa.analysis.Analyzer_one
    analyzer_classes = [
        "SemanticAnalyzer", "PerformanceAnalyzer", "TextMiningEngine", "MunicipalOntology"
    ]
    for class_name in analyzer_classes:
        assert paths[class_name].startswith("saaaaaa.analysis.Analyzer_one."), \
            f"{class_name} should be in saaaaaa.analysis.Analyzer_one"

    # Theory of Change classes should be in saaaaaa.analysis.teoria_cambio
    teoria_classes = ["TeoriaCambio", "AdvancedDAGValidator"]
    for class_name in teoria_classes:
        assert paths[class_name].startswith("saaaaaa.analysis.teoria_cambio."), \
            f"{class_name} should be in saaaaaa.analysis.teoria_cambio"

    # Financial class should be in saaaaaa.analysis.financiero_viabilidad_tablas
    assert paths["PDETMunicipalPlanAnalyzer"].startswith(
        "saaaaaa.analysis.financiero_viabilidad_tablas."
    ), "PDETMunicipalPlanAnalyzer should be in
saaaaaa.analysis.financiero_viabilidad_tablas"

    # Embedding policy classes should be in saaaaaa.processing.embedding_policy
    embedding_classes = [
        "BayesianNumericalAnalyzer", "PolicyAnalysisEmbedder",
        "AdvancedSemanticChunker", "SemanticChunker"
    ]
    for class_name in embedding_classes:
        assert paths[class_name].startswith("saaaaaa.processing.embedding_policy."), \
            f"{class_name} should be in saaaaaa.processing.embedding_policy"

    # Policy processor classes should be in saaaaaa.processing.policy_processor
    processor_classes = [
        "IndustrialPolicyProcessor", "PolicyTextProcessor", "BayesianEvidenceScorer"
    ]
    for class_name in processor_classes:
        assert paths[class_name].startswith("saaaaaa.processing.policy_processor."), \
            f"{class_name} should be in saaaaaa.processing.policy_processor"

def test_class_registry_import_structure():
    """Test that class registry can be imported and has correct structure."""
    from saaaaaa.core.orchestrator.class_registry import (
        ClassRegistryError,
        build_class_registry,
        get_class_paths,
    )

    # Verify functions exist
    assert callable(build_class_registry)
    assert callable(get_class_paths)

    # Verify exception exists
    assert issubclass(ClassRegistryError, RuntimeError)

    # Verify get_class_paths returns a mapping
    paths = get_class_paths()
    assert isinstance(paths, dict) or hasattr(paths, '__getitem__')
    assert len(paths) > 0

def test_semantic_chunker_alias():
    """Verify SemanticChunker is an alias for AdvancedSemanticChunker."""
    from saaaaaa.core.orchestrator.class_registry import get_class_paths

    paths = get_class_paths()

    # Both should point to the same class
    assert paths["SemanticChunker"] == paths["AdvancedSemanticChunker"], \
        "SemanticChunker should be an alias for AdvancedSemanticChunker"
    assert paths["SemanticChunker"] ==
"saaaaaa.processing.embedding_policy.AdvancedSemanticChunker"

if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_concurrency.py =====
#!/usr/bin/env python3
"""
Tests for concurrency module.

Tests verify:
- Deterministic execution
- No race conditions
- Proper max_workers control
- Backoff and retry behavior
- Abortability
- Instrumentation and logging
"""

import time
import unittest
from concurrent.futures import ThreadPoolExecutor

from saaaaaa.concurrency import (
    TaskExecutionError,
    TaskStatus,
    WorkerPool,
    WorkerPoolConfig,
)

class TestWorkerPoolBasics(unittest.TestCase):
    """Test basic WorkerPool functionality."""

    def test_pool_initialization(self):
        """Test that pool initializes with correct config."""
        config = WorkerPoolConfig(
            max_workers=10,
            max_retries=2,
            task_timeout_seconds=60.0
        )
        pool = WorkerPool(config)

        self.assertEqual(pool.config.max_workers, 10)
        self.assertEqual(pool.config.max_retries, 2)
        self.assertEqual(pool.config.task_timeout_seconds, 60.0)

        pool.shutdown()
    def test_simple_task_submission(self):
        """Test submitting and executing a simple task."""
        pool = WorkerPool()

        def simple_task(x):
            return x * 2

        task_id = pool.submit_task("double_5", simple_task, args=(5,))
        result = pool.get_task_result(task_id)

        self.assertTrue(result.success)
        self.assertEqual(result.result, 10)
        self.assertIsNotNone(result.metrics)
        self.assertEqual(result.metrics.status, TaskStatus.COMPLETED)

        pool.shutdown()

    def test_multiple_tasks(self):
        """Test executing multiple tasks in parallel."""
        config = WorkerPoolConfig(max_workers=5)
        pool = WorkerPool(config)

        def square(x):
            time.sleep(0.01)  # Simulate work
            return x * x

        # Submit 10 tasks
        task_ids = []
        for i in range(10):
            task_id = pool.submit_task(f"square_{i}", square, args=(i,))
            task_ids.append(task_id)

        # Wait for all tasks
        results = pool.wait_for_all()

        self.assertEqual(len(results), 10)
        self.assertTrue(all(r.success for r in results))

        # Verify results (order may vary due to parallelism)
        result_values = sorted([r.result for r in results])
        expected = [i * i for i in range(10)]
        self.assertEqual(result_values, expected)

        pool.shutdown()

    def test_context_manager(self):
        """Test WorkerPool as context manager."""
        def dummy_task():
            return 42

        with WorkerPool() as pool:
            task_id = pool.submit_task("dummy", dummy_task)
            result = pool.get_task_result(task_id)
            self.assertTrue(result.success)
            self.assertEqual(result.result, 42)

        # Pool should be shutdown after context exit
        self.assertTrue(pool._is_shutdown)

class TestWorkerPoolRetry(unittest.TestCase):
    """Test retry and backoff behavior."""

    def test_successful_task_no_retry(self):
        """Test that successful tasks don't retry."""
        config = WorkerPoolConfig(max_retries=3)
        pool = WorkerPool(config)
        def always_succeed():
            return "success"

        task_id = pool.submit_task("no_retry", always_succeed)
        result = pool.get_task_result(task_id)

        self.assertTrue(result.success)
        self.assertEqual(result.result, "success")
        self.assertEqual(result.metrics.retries_used, 0)

        pool.shutdown()

    def test_task_retry_on_failure(self):
        """Test that failing tasks are retried."""
        config = WorkerPoolConfig(max_retries=2, backoff_base_seconds=0.01)
        pool = WorkerPool(config)

        # Counter to track attempts
        attempts = {"count": 0}

        def fail_twice_then_succeed():
            attempts["count"] += 1
            if attempts["count"] < 3:
                raise ValueError(f"Attempt {attempts['count']} failed")
            return "success"

        task_id = pool.submit_task("retry_task", fail_twice_then_succeed)
        result = pool.get_task_result(task_id, timeout=5.0)

        self.assertTrue(result.success)
        self.assertEqual(result.result, "success")
        self.assertEqual(result.metrics.retries_used, 2)
        self.assertEqual(attempts["count"], 3)  # Initial + 2 retries

        pool.shutdown()

    def test_task_fails_after_max_retries(self):
        """Test that tasks fail after exhausting retries."""
        config = WorkerPoolConfig(max_retries=2, backoff_base_seconds=0.01)
        pool = WorkerPool(config)

        def always_fail():
            raise ValueError("Always fails")

        task_id = pool.submit_task("fail_task", always_fail)
        result = pool.get_task_result(task_id, timeout=5.0)

        self.assertFalse(result.success)
        self.assertIsInstance(result.error, TaskExecutionError)
        self.assertEqual(result.metrics.status, TaskStatus.FAILED)
        self.assertEqual(result.metrics.retries_used, 2)

        pool.shutdown()

    def test_exponential_backoff(self):
        """Test that backoff increases exponentially."""
        config = WorkerPoolConfig(
            max_retries=3,
            backoff_base_seconds=0.1,
            backoff_max_seconds=1.0
        )
        pool = WorkerPool(config)

        # Test backoff calculation
        self.assertEqual(pool._calculate_backoff_delay(0), 0.1)
        self.assertEqual(pool._calculate_backoff_delay(1), 0.2)
        self.assertEqual(pool._calculate_backoff_delay(2), 0.4)
        self.assertEqual(pool._calculate_backoff_delay(3), 0.8)
        # Should be capped at max
        self.assertEqual(pool._calculate_backoff_delay(10), 1.0)

        pool.shutdown()

class TestWorkerPoolAbort(unittest.TestCase):
    """Test abort and cancellation functionality."""

    def test_abort_pending_tasks(self):
        """Test aborting pending tasks."""
        config = WorkerPoolConfig(max_workers=2)
        pool = WorkerPool(config)

        def slow_task():
            time.sleep(1.0)
            return "done"

        # Submit many tasks to queue them up
        task_ids = []
        for i in range(10):
            task_id = pool.submit_task(f"slow_{i}", slow_task)
            task_ids.append(task_id)

        # Give some time for a few to start
        time.sleep(0.1)

        # Abort pending tasks
        cancelled_count = pool.abort_pending_tasks()

        # Should have cancelled some tasks
        self.assertGreater(cancelled_count, 0)

        pool.shutdown(wait=False, cancel_futures=True)

class TestWorkerPoolMetrics(unittest.TestCase):
    """Test metrics and instrumentation."""

    def test_task_metrics_collection(self):
        """Test that metrics are collected for each task."""
        pool = WorkerPool()

        def task_with_delay(delay):
            time.sleep(delay)
            return "done"

        task_id = pool.submit_task("delayed", task_with_delay, args=(0.05,))
        result = pool.get_task_result(task_id)

        self.assertTrue(result.success)
        self.assertIsNotNone(result.metrics)
        self.assertEqual(result.metrics.task_id, task_id)
        self.assertEqual(result.metrics.task_name, "delayed")
        self.assertEqual(result.metrics.status, TaskStatus.COMPLETED)
        self.assertGreater(result.metrics.execution_time_ms, 0)
        self.assertIsNotNone(result.metrics.worker_id)

        pool.shutdown()

    def test_summary_metrics(self):
        """Test summary metrics aggregation."""
        config = WorkerPoolConfig(max_workers=5, max_retries=1)
        pool = WorkerPool(config)

        def success_task():
            return "ok"

        def fail_task():
            raise ValueError("fail")

        # Submit mix of successful and failing tasks
        for i in range(5):
            pool.submit_task(f"success_{i}", success_task)
        for i in range(3):
            pool.submit_task(f"fail_{i}", fail_task)

        # Wait for all
        results = pool.wait_for_all(timeout=10.0)

        # Assert results: 5 successes, 3 failures
        success_count = 0
        fail_count = 0
        for result in results:
            if isinstance(result, Exception):
                fail_count += 1
                self.assertIsInstance(result, TaskExecutionError)
            else:
                success_count += 1
                self.assertEqual(result, "ok")

        self.assertEqual(success_count, 5)
        self.assertEqual(fail_count, 3)

        # Validate results
        successful = [r for r in results if r.success]
        failed = [r for r in results if not r.success]
        self.assertEqual(len(successful), 5)
        self.assertEqual(len(failed), 3)

        # Get summary
        summary = pool.get_summary_metrics()

        self.assertEqual(summary["total_tasks"], 8)
        self.assertEqual(summary["completed"], 5)
        self.assertEqual(summary["failed"], 3)
        self.assertGreater(summary["avg_execution_time_ms"], 0)
        # Failing tasks should have retried once each
        self.assertEqual(summary["total_retries"], 3)

        pool.shutdown()

class TestWorkerPoolThreadSafety(unittest.TestCase):
    """Test thread safety and no race conditions."""

    def test_concurrent_submissions(self):
        """Test submitting tasks from multiple threads."""
        pool = WorkerPool()

        task_ids = []
        lock = ThreadPoolExecutor(max_workers=5)

        def submit_tasks(start, end):
            for i in range(start, end):
                task_id = pool.submit_task(
                    f"task_{i}",
                    lambda x, val=i: val * 2,
                    args=(i,)
                )
                task_ids.append(task_id)

        # Submit from multiple threads
        futures = []
        for i in range(5):
            future = lock.submit(submit_tasks, i * 10, (i + 1) * 10)
            futures.append(future)
        # Wait for submissions
        for future in futures:
            future.result()

        lock.shutdown()

        # All 50 tasks should be submitted
        self.assertEqual(len(task_ids), 50)

        # Wait for all tasks
        results = pool.wait_for_all(timeout=10.0)
        self.assertEqual(len(results), 50)
        self.assertTrue(all(r.success for r in results))

        pool.shutdown()

    def test_no_race_conditions_in_metrics(self):
        """Test that metrics are updated atomically without races."""
        config = WorkerPoolConfig(max_workers=10)
        pool = WorkerPool(config)

        def increment_shared(shared_dict, key):
            # Simulate work
            time.sleep(0.001)
            return key

        shared = {}

        # Submit many tasks
        for i in range(100):
            pool.submit_task(f"task_{i}", increment_shared, args=(shared, i))

        # Wait for all
        results = pool.wait_for_all(timeout=20.0)

        # Verify all tasks completed
        self.assertEqual(len(results), 100)

        # Check metrics consistency
        metrics = pool.get_metrics()
        self.assertEqual(len(metrics), 100)

        # All should be completed or failed (no partial states)
        for metric in metrics.values():
            self.assertIn(
                metric.status,
                [TaskStatus.COMPLETED, TaskStatus.FAILED]
            )

        pool.shutdown()

class TestWorkerPoolDeterminism(unittest.TestCase):
    """Test deterministic behavior."""

    def test_consistent_results(self):
        """Test that same tasks produce consistent results."""
        def deterministic_task(x):
            # Pure function - always produces same output for same input
            return x * x + 2 * x + 1

        # Run twice with same inputs
        results1 = []
        results2 = []

        for run in range(2):
            pool = WorkerPool()

            for i in range(20):
                pool.submit_task(f"task_{i}", deterministic_task, args=(i,))

            results = pool.wait_for_all()
            if run == 0:
                results1 = sorted([r.result for r in results if r.success])
            else:
                results2 = sorted([r.result for r in results if r.success])

            pool.shutdown()

        # Results should be identical
        self.assertEqual(results1, results2)

    def test_task_execution_order_within_constraints(self):
        """Test that tasks respect max_workers constraint."""
        config = WorkerPoolConfig(max_workers=2)
        pool = WorkerPool(config)

        execution_log = []

        def logged_task(task_num):
            execution_log.append(("start", task_num, time.time()))
            time.sleep(0.1)
            execution_log.append(("end", task_num, time.time()))
            return task_num

        # Submit 6 tasks
        for i in range(6):
            pool.submit_task(f"task_{i}", logged_task, args=(i,))

        pool.wait_for_all(timeout=10.0)

        # Check that at most 2 tasks were running simultaneously
        running = []
        max_concurrent = 0

        for event, task_num, _timestamp in execution_log:
            if event == "start":
                running.append(task_num)
                max_concurrent = max(max_concurrent, len(running))
            elif event == "end":
                running.remove(task_num)

        # Should not exceed max_workers
        self.assertLessEqual(max_concurrent, 2)

        pool.shutdown()

def run_tests():
    """Run all tests."""
    unittest.main(verbosity=2)

if __name__ == "__main__":
    run_tests()

===== FILE: tests/test_contract_runtime.py =====
"""
Tests for runtime contract validation using Pydantic models.

These tests ensure that:
1. Valid contracts pass validation
2. Invalid contracts fail validation with appropriate errors
3. Schema versioning is enforced
4. Field constraints are properly validated
"""

import pytest
```

```
# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="Runtime contracts now in
test_contracts_comprehensive.py")

from pydantic import ValidationError

from saaaaaa.utils.contracts_runtime import (
    CDAFFrameworkInputModel,
    CDAFFrameworkOutputModel,
    ContradictionDetectorInputModel,
    ContradictionDetectorOutputModel,
    EmbeddingPolicyInputModel,
    EmbeddingPolicyOutputModel,
    PDETAnalyzerInputModel,
    PDETAnalyzerOutputModel,
    PolicyProcessorInputModel,
    PolicyProcessorOutputModel,
    SemanticAnalyzerInputModel,
    SemanticAnalyzerOutputModel,
    SemanticChunkingInputModel,
    SemanticChunkingOutputModel,
    TeoriaCambioInputModel,
)

class TestSemanticAnalyzerContracts:
    """Test SemanticAnalyzer contract validation."""

    def test_valid_input_minimal(self):
        """Valid input with minimal required fields."""
        model = SemanticAnalyzerInputModel(
            text="Sample municipal plan text",
            schema_version="sem-1.0"
        )
        assert model.text == "Sample municipal plan text"
        assert model.schema_version == "sem-1.0"
        assert model.segments == []
        assert model.ontology_params == {}

    def test_valid_input_full(self):
        """Valid input with all fields populated."""
        model = SemanticAnalyzerInputModel(
            text="El plan de desarrollo municipal...",
            segments=["Segment 1", "Segment 2"],
            ontology_params={"domain": "municipal"},
            schema_version="sem-1.1"
        )
        assert len(model.segments) == 2
        assert model.ontology_params["domain"] == "municipal"

    def test_invalid_empty_text(self):
        """Empty text should fail validation."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerInputModel(
                text="",
                schema_version="sem-1.0"
            )
        assert "text" in str(exc_info.value)

    def test_invalid_whitespace_text(self):
        """Whitespace-only text should fail validation."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerInputModel(
                text="   ",
                schema_version="sem-1.0"
            )
        assert "text" in str(exc_info.value)

    def test_invalid_schema_version(self):
        """Invalid schema version format should fail."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerInputModel(
                text="Sample text",
                schema_version="v1.0"  # Wrong format
            )
        assert "schema_version" in str(exc_info.value)

    def test_unknown_field_rejected(self):
        """Unknown fields should be rejected (strict mode)."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerInputModel(
                text="Sample text",
                schema_version="sem-1.0",
                unknown_field="value"
            )
        assert "Extra inputs are not permitted" in str(exc_info.value)

    def test_valid_output(self):
        """Valid output contract."""
        model = SemanticAnalyzerOutputModel(
            semantic_cube={"key": "value"},
            coherence_score=0.85,
            complexity_score=2.5,
            domain_classification={"municipal": 0.8, "economic": 0.2},
            schema_version="sem-1.0"
        )
        assert model.coherence_score == 0.85
        assert 0.0 <= model.coherence_score <= 1.0

    def test_invalid_coherence_score(self):
        """Coherence score outside [0, 1] should fail."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerOutputModel(
                semantic_cube={},
                coherence_score=1.5,  # Invalid: > 1
                complexity_score=1.0,
                domain_classification={},
                schema_version="sem-1.0"
            )
        assert "coherence_score" in str(exc_info.value)

    def test_invalid_domain_probabilities(self):
        """Domain probabilities outside [0, 1] should fail."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerOutputModel(
                semantic_cube={},
                coherence_score=0.5,
                complexity_score=1.0,
                domain_classification={"domain1": 1.5},  # Invalid
                schema_version="sem-1.0"
            )
        assert "Probability" in str(exc_info.value)

class TestCDAFFrameworkContracts:
    """Test CDAF Framework contract validation."""

    def test_valid_input(self):
        """Valid CDAF input."""
        model = CDAFFrameworkInputModel(
            document_text="Plan document text",
            plan_metadata={"author": "Municipality"},
            schema_version="sem-1.0"
        )
        assert model.document_text == "Plan document text"

    def test_valid_output(self):
        """Valid CDAF output."""
        model = CDAFFrameworkOutputModel(
            causal_mechanisms=[{"type": "mechanism1"}],
            evidential_tests={"test1": "result"},
            bayesian_inference={"posterior": 0.8},
            audit_results={"status": "passed"},
            schema_version="sem-1.0"
        )
        assert len(model.causal_mechanisms) == 1

class TestPDETAnalyzerContracts:
    """Test PDET Analyzer contract validation."""

    def test_valid_input(self):
        """Valid PDET input."""
        model = PDETAnalyzerInputModel(
            document_content="Financial document",
            extract_tables=True,
            schema_version="sem-1.0"
        )
        assert model.extract_tables is True

    def test_valid_output(self):
        """Valid PDET output."""
        model = PDETAnalyzerOutputModel(
            extracted_tables=[{"table1": "data"}],
            financial_indicators={"indicator1": 100.0},
            viability_score=0.75,
            quality_scores={"quality1": 0.9},
            schema_version="sem-1.0"
        )
        assert 0.0 <= model.viability_score <= 1.0

    def test_invalid_viability_score(self):
        """Viability score outside [0, 1] should fail."""
        with pytest.raises(ValidationError) as exc_info:
            PDETAnalyzerOutputModel(
                extracted_tables=[],
                financial_indicators={},
                viability_score=1.2,  # Invalid
                quality_scores={},
                schema_version="sem-1.0"
            )
        assert "viability_score" in str(exc_info.value)

class TestContradictionDetectorContracts:
    """Test Contradiction Detector contract validation."""

    def test_valid_input(self):
        """Valid contradiction detector input."""
        model = ContradictionDetectorInputModel(
            text="Policy document text",
            plan_name="Municipal Plan 2024",
            dimension="economic",
            schema_version="sem-1.0"
        )
        assert model.plan_name == "Municipal Plan 2024"

    def test_valid_output(self):
        """Valid contradiction detector output."""
        model = ContradictionDetectorOutputModel(
            contradictions=[{"id": 1, "description": "Contradiction A"}],
            confidence_scores={"contradiction_1": 0.9},
            temporal_conflicts=[],
            severity_scores={"contradiction_1": 0.8},
            schema_version="sem-1.0"
        )
        assert len(model.contradictions) == 1
```

```
class TestEmbeddingPolicyContracts:
    """Test Embedding Policy contract validation."""

    def test_valid_input(self):
        """Valid embedding policy input."""
        model = EmbeddingPolicyInputModel(
            text="Policy text",
            dimensions=["economic", "social"],
            schema_version="sem-1.0"
        )
        assert len(model.dimensions) == 2

    def test_valid_output(self):
        """Valid embedding policy output."""
        model = EmbeddingPolicyOutputModel(
            embeddings=[[0.1, 0.2, 0.3]],
            similarity_scores={"score1": 0.8},
            bayesian_evaluation={},
            policy_metrics={},
            schema_version="sem-1.0"
        )
        assert len(model.embeddings) == 1

class TestSemanticChunkingContracts:
    """Test Semantic Chunking contract validation."""

    def test_valid_input(self):
        """Valid semantic chunking input."""
        model = SemanticChunkingInputModel(
            text="Document to chunk",
            preserve_structure=True,
            schema_version="sem-1.0"
        )
        assert model.preserve_structure is True

    def test_valid_output(self):
        """Valid semantic chunking output."""
        model = SemanticChunkingOutputModel(
            chunks=[{"id": 1, "text": "Chunk 1"}],
            causal_dimensions={},
            key_excerpts={},
            summary={},
            schema_version="sem-1.0"
        )
        assert len(model.chunks) == 1

class TestPolicyProcessorContracts:
    """Test Policy Processor contract validation."""

    def test_valid_input(self):
        """Valid policy processor input."""
        model = PolicyProcessorInputModel(
            data={"key": "value"},
            text="Policy text",
            sentences=["Sentence 1", "Sentence 2"],
            schema_version="sem-1.0"
        )
        assert len(model.sentences) == 2

    def test_valid_output(self):
        """Valid policy processor output."""
        model = PolicyProcessorOutputModel(
            processed_data={"result": "data"},
            evidence_bundles=[],
            bayesian_scores={},
            matched_patterns=[],
            schema_version="sem-1.0"
        )
        assert model.processed_data["result"] == "data"

class TestSchemaVersioning:
    """Test schema versioning across all contracts."""

    @pytest.mark.parametrize("model_class", [
        SemanticAnalyzerInputModel,
        CDAFFrameworkInputModel,
        PDETAnalyzerInputModel,
        TeoriaCambioInputModel,
        ContradictionDetectorInputModel,
        EmbeddingPolicyInputModel,
        SemanticChunkingInputModel,
        PolicyProcessorInputModel,
    ])
    def test_default_schema_version(self, model_class):
        """All contracts default to sem-1.0."""
        # Create minimal valid instance
        kwargs = {}
        if hasattr(model_class.model_fields.get('text'), 'annotation'):
            kwargs['text'] = "Sample text"
        if hasattr(model_class.model_fields.get('document_text'), 'annotation'):
            kwargs['document_text'] = "Sample document"
        if hasattr(model_class.model_fields.get('document_content'), 'annotation'):
            kwargs['document_content'] = "Sample content"
        if hasattr(model_class.model_fields.get('plan_name'), 'annotation'):
            kwargs['plan_name'] = "Plan"
        if hasattr(model_class.model_fields.get('data'), 'annotation'):
            kwargs['data'] = {}
            kwargs['text'] = "Sample text"

        model = model_class(**kwargs)
        assert model.schema_version == "sem-1.0"

    @pytest.mark.parametrize("version", ["sem-1.0", "sem-1.1", "sem-2.0", "sem-10.5"])
    def test_valid_version_formats(self, version):
        """Test various valid version formats."""
        model = SemanticAnalyzerInputModel(
            text="Test",
            schema_version=version
        )
        assert model.schema_version == version

    @pytest.mark.parametrize("invalid_version", ["v1.0", "1.0", "sem-1", "sem-a.b"])
    def test_invalid_version_formats(self, invalid_version):
        """Test that invalid version formats are rejected."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerInputModel(
                text="Test",
                schema_version=invalid_version
            )
        assert "schema_version" in str(exc_info.value)

class TestStrictMode:
    """Test that strict mode rejects unknown fields."""

    def test_reject_extra_input_fields(self):
        """Extra fields in input contracts should be rejected."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerInputModel(
                text="Test",
                schema_version="sem-1.0",
                extra_field="not allowed"
            )
        assert "Extra inputs are not permitted" in str(exc_info.value)

    def test_reject_extra_output_fields(self):
        """Extra fields in output contracts should be rejected."""
        with pytest.raises(ValidationError) as exc_info:
            SemanticAnalyzerOutputModel(
                semantic_cube={},
                coherence_score=0.5,
                complexity_score=1.0,
                domain_classification={},
                schema_version="sem-1.0",
                extra_result="not allowed"
            )
        assert "Extra inputs are not permitted" in str(exc_info.value)

===== FILE: tests/test_contract_snapshots.py =====
"""Snapshot tests that guard contract schemas."""

from __future__ import annotations

import json
from pathlib import Path

import pytest

from saaaaaa.utils import contracts as core_contracts


# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="Snapshot testing replaced by deterministic
fingerprints")

SNAPSHOT_PATH = Path(__file__).parent / "data" / "contract_snapshots.json"

def _format_type(annotation: object) -> str:
    text = repr(annotation)
    return text.replace("typing.", "")

def _collect_contracts() -> dict[str, dict[str, str]]:
    members: dict[str, dict[str, str]] = {}
    for name in dir(core_contracts):
        if not name.endswith("Contract"):
            continue
        obj = getattr(core_contracts, name)
        annotations = getattr(obj, "__annotations__", None)
        if not isinstance(annotations, dict):
            continue
        members[name] = {
            field: _format_type(annotation)
            for field, annotation in sorted(annotations.items())
        }
    return dict(sorted(members.items()))

def test_contract_snapshots_are_stable() -> None:
    assert SNAPSHOT_PATH.exists(), (
        "Contract snapshot missing. Run the governance tests to regenerate "
        "or update tests/data/contract_snapshots.json."
    )

    current = _collect_contracts()
    stored = json.loads(SNAPSHOT_PATH.read_text(encoding="utf-8"))
    assert current == stored, (
        "Core contract schema changed. Update tests/data/contract_snapshots.json "
        "after stakeholder approval."
    )

===== FILE: tests/test_contracts.py =====
"""
CONTRACT TESTS - API Boundary Validation
=========================================

Tests that verify API contracts are maintained across module boundaries.
```

```
Every public function/class gets a test that validates documented input/output shapes.

If a signature or schema drifts, the contract test breaks before production does.

NOTE: This test file is OUTDATED. Use test_contracts_comprehensive.py instead.
"""

from typing import Any

import pytest

# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="outdated - use test_contracts_comprehensive.py")

from saaaaaa.contracts import (
    MISSING,
    AnalysisInputV1,
    AnalysisOutputV1,
    DocumentMetadataV1,
    ProcessedTextV1,
    SentenceCollection,
    TextDocument,
    ensure_hashable,
    ensure_iterable_not_string,
    validate_contract,
    validate_mapping_keys,
)

class TestContractValidation:
    """Test runtime contract validation helpers."""

    def test_validate_contract_success(self) -> None:
        """Validate contract passes for correct type."""
        validate_contract(
            "hello",
            str,
            parameter="text",
            producer="test",
            consumer="validator",
        )

    def test_validate_contract_failure(self) -> None:
        """Validate contract raises TypeError for wrong type."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH"):
            validate_contract(
                123,
                str,
                parameter="text",
                producer="test",
                consumer="validator",
            )

    def test_validate_mapping_keys_success(self) -> None:
        """Validate mapping with all required keys."""
        mapping: dict[str, Any] = {"key1": "val1", "key2": "val2"}
        validate_mapping_keys(
            mapping,
            ["key1", "key2"],
            producer="test",
            consumer="validator",
        )

    def test_validate_mapping_keys_failure(self) -> None:
        """Validate mapping raises KeyError for missing keys."""
        mapping: dict[str, Any] = {"key1": "val1"}
        with pytest.raises(KeyError, match="ERR_CONTRACT_MISMATCH.*missing_keys"):
            validate_mapping_keys(
                mapping,
                ["key1", "key2"],
                producer="test",
                consumer="validator",
            )

    def test_ensure_iterable_not_string_success(self) -> None:
        """Validate iterable that is not string passes."""
        ensure_iterable_not_string(
            [1, 2, 3],
            parameter="items",
            producer="test",
            consumer="validator",
        )

    def test_ensure_iterable_not_string_rejects_string(self) -> None:
        """Validate iterable check rejects strings."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH.*not str/bytes"):
            ensure_iterable_not_string(
                "string",
                parameter="items",
                producer="test",
                consumer="validator",
            )

    def test_ensure_iterable_not_string_rejects_bool(self) -> None:
        """Validate iterable check rejects non-iterables like bool."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH"):
            ensure_iterable_not_string(
                True,
                parameter="items",
                producer="test",
                consumer="validator",
            )

    def test_ensure_hashable_success(self) -> None:
        """Validate hashable check passes for hashable types."""
        ensure_hashable(
            "string",
            parameter="key",
            producer="test",
            consumer="validator",
        )
        ensure_hashable(
            123,
            parameter="key",
            producer="test",
            consumer="validator",
        )
        ensure_hashable(
            (1, 2, 3),
            parameter="key",
            producer="test",
            consumer="validator",
        )

    def test_ensure_hashable_rejects_dict(self) -> None:
        """Validate hashable check rejects dicts."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH.*unhashable"):
            ensure_hashable(
                {"key": "value"},
                parameter="key",
                producer="test",
                consumer="validator",
            )

    def test_ensure_hashable_rejects_list(self) -> None:
        """Validate hashable check rejects lists."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH.*unhashable"):
            ensure_hashable(
                [1, 2, 3],
                parameter="key",
                producer="test",
                consumer="validator",
            )

class TestValueObjects:
    """Test value objects that prevent type confusion."""

    def test_text_document_creation(self) -> None:
        """TextDocument can be created with valid inputs."""
        doc = TextDocument(
            text="Hello world",
            document_id="doc123",
            metadata={},
        )
        assert doc.text == "Hello world"
        assert doc.document_id == "doc123"

    def test_text_document_rejects_non_string_text(self) -> None:
        """TextDocument rejects non-string text."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH.*text must be str"):
            TextDocument(
                text=123,  # type: ignore[arg-type]
                document_id="doc123",
                metadata={},
            )

    def test_text_document_rejects_empty_text(self) -> None:
        """TextDocument rejects empty text."""
        with pytest.raises(ValueError, match="ERR_CONTRACT_MISMATCH.*cannot be empty"):
            TextDocument(
                text="",
                document_id="doc123",
                metadata={},
            )

    def test_text_document_is_frozen(self) -> None:
        """TextDocument is immutable."""
        doc = TextDocument(text="Hello", document_id="doc123", metadata={})
        with pytest.raises(AttributeError):
            doc.text = "World"  # type: ignore[misc]

    def test_sentence_collection_creation(self) -> None:
        """SentenceCollection can be created with valid sentences."""
        sentences = SentenceCollection(sentences=("Hello", "World"))
        assert len(sentences) == 2
        assert list(sentences) == ["Hello", "World"]

    def test_sentence_collection_rejects_non_strings(self) -> None:
        """SentenceCollection rejects non-string items."""
        with pytest.raises(TypeError, match="ERR_CONTRACT_MISMATCH.*must be strings"):
            SentenceCollection(sentences=(123, 456))  # type: ignore[arg-type]

    def test_sentence_collection_is_hashable(self) -> None:
        """SentenceCollection can be used in sets and as dict keys."""
        s1 = SentenceCollection(sentences=("Hello",))
        s2 = SentenceCollection(sentences=("World",))

        # Can add to set
        sentence_set = {s1, s2}
        assert len(sentence_set) == 2

        # Can use as dict key
        mapping = {s1: "first", s2: "second"}
        assert mapping[s1] == "first"
```

```
class TestSentinelValues:
    """Test MISSING sentinel for optional parameters."""

    def test_missing_sentinel_identity(self) -> None:
        """MISSING has identity semantics."""
        from saaaaaa.contracts import MISSING as MISSING2
        assert MISSING is MISSING2

    def test_missing_sentinel_not_none(self) -> None:
        """MISSING is distinguishable from None."""
        assert MISSING is not None
        assert MISSING != None  # noqa: E711

    def test_missing_sentinel_repr(self) -> None:
        """MISSING has readable repr."""
        assert repr(MISSING) == "<MISSING>"

class TestTypedDictContracts:
    """Test TypedDict definitions for data shapes."""

    def test_document_metadata_v1_required_fields(self) -> None:
        """DocumentMetadataV1 requires all fields."""
        metadata: DocumentMetadataV1 = {
            "file_path": "/path/to/file.pdf",
            "file_name": "file.pdf",
            "num_pages": 10,
            "file_size_bytes": 1024,
            "file_hash": "abc123",
        }
        assert metadata["file_path"] == "/path/to/file.pdf"
        assert metadata["num_pages"] == 10

    def test_processed_text_v1_shape(self) -> None:
        """ProcessedTextV1 has correct shape."""
        text: ProcessedTextV1 = {
            "raw_text": "Hello world",
            "normalized_text": "hello world",
            "language": "es",
            "encoding": "utf-8",
        }
        assert text["language"] == "es"

    def test_analysis_input_v1_keyword_only_semantics(self) -> None:
        """AnalysisInputV1 is designed for keyword-only usage."""
        # Should be used with keyword args
        input_data: AnalysisInputV1 = {
            "text": "Sample text",
            "document_id": "doc123",
        }
        assert input_data["text"] == "Sample text"

    def test_analysis_output_v1_shape(self) -> None:
        """AnalysisOutputV1 has correct output shape."""
        output: AnalysisOutputV1 = {
            "dimension": "D1",
            "category": "insumos",
            "confidence": 0.85,
            "matches": ["palabra1", "palabra2"],
        }
        assert output["confidence"] == 0.85
        assert len(output["matches"]) == 2

@pytest.mark.contract
class TestDocumentIngestionContracts:
    """Contract tests for document ingestion module boundaries."""

    def test_document_loader_protocol_signature(self) -> None:
        """DocumentLoader.load_pdf must use keyword-only params."""
        import inspect

        from saaaaaa.contracts import DocumentLoaderProtocol

        # Check protocol signature
        sig = inspect.signature(DocumentLoaderProtocol.load_pdf)
        params = list(sig.parameters.values())

        # First param is self, second should be KEYWORD_ONLY
        assert len(params) >= 2
        # pdf_path should be keyword-only
        pdf_path_param = [p for p in params if p.name == "pdf_path"][0]
        assert pdf_path_param.kind == inspect.Parameter.KEYWORD_ONLY

@pytest.mark.contract
class TestAnalyzerContracts:
    """Contract tests for analyzer module boundaries."""

    def test_analyzer_protocol_keyword_only(self) -> None:
        """Analyzer.analyze must use keyword-only params."""
        import inspect

        from saaaaaa.contracts import AnalyzerProtocol

        sig = inspect.signature(AnalyzerProtocol.analyze)
        params = list(sig.parameters.values())

        # All params except self should be keyword-only
        for param in params[1:]:  # Skip self
            assert param.kind in (
                inspect.Parameter.KEYWORD_ONLY,
                inspect.Parameter.VAR_KEYWORD,
            )

if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_contracts_comprehensive.py =====
"""
Comprehensive Contract Tests - API Boundary Validation
======================================================

Tests that verify API contracts are maintained across all module boundaries.
Validates:
- Preconditions: Input validation before execution
- Postconditions: Output validation after execution
- Invariants: State consistency throughout execution
- Type safety: All inputs/outputs match declared types
- Error handling: Proper exceptions for invalid inputs

Modules tested:
- scoring: All 6 scoring modalities (TYPE_A through TYPE_F)
- aggregation: Dimension, Area, Cluster, Macro aggregators
- concurrency: WorkerPool, task submission, metrics
- recommendation_engine: Rule evaluation, template rendering
- seed_factory: Deterministic seed generation
"""

from pathlib import Path

import pytest

# ============================================================================
# SCORING MODULE CONTRACTS
# ============================================================================

class TestScoringContracts:
    """Test scoring module contract enforcement."""
    def test_scoring_precondition_evidence_dict(self):
        """Scoring requires evidence to be a dictionary."""
        from saaaaaa.scoring.scoring import ModalityConfig, ScoringModality, score_type_a

        config = ModalityConfig(
            modality=ScoringModality.TYPE_A,
            score_range=(0.0, 3.0),
            threshold=0.7,
            required_keys=["elements", "confidence"],
        )

        # Valid precondition
        evidence = {"elements": [1, 2, 3, 4], "confidence": 0.9}
        score, metadata = score_type_a(evidence, config)
        assert isinstance(score, float)
        assert isinstance(metadata, dict)

        # Invalid precondition - not a dict
        with pytest.raises((TypeError, AttributeError, KeyError)):
            score_type_a("not a dict", config)  # type: ignore

    def test_scoring_postcondition_score_range(self):
        """Scoring postcondition: score must be within declared range."""
        from saaaaaa.scoring.scoring import ModalityConfig, ScoringModality, score_type_a

        config = ModalityConfig(
            modality=ScoringModality.TYPE_A,
            score_range=(0.0, 3.0),
            threshold=0.7,
            required_keys=["elements", "confidence"],
        )

        # Test various evidence, ensure score always in range
        test_cases = [
            {"elements": [], "confidence": 0.0},
            {"elements": [1], "confidence": 0.5},
            {"elements": [1, 2], "confidence": 0.75},
            {"elements": [1, 2, 3, 4], "confidence": 1.0},
        ]

        for evidence in test_cases:
            score, metadata = score_type_a(evidence, config)
            assert 0.0 <= score <= 3.0, f"Score {score} out of range for {evidence}"

    def test_scoring_invariant_determinism(self):
        """Scoring invariant: same input produces same output."""
        from saaaaaa.scoring.scoring import ModalityConfig, ScoringModality, score_type_a

        config = ModalityConfig(
            modality=ScoringModality.TYPE_A,
            score_range=(0.0, 3.0),
            threshold=0.7,
            required_keys=["elements", "confidence"],
        )

        evidence = {"elements": [1, 2, 3], "confidence": 0.85}

        score1, metadata1 = score_type_a(evidence, config)
        score2, metadata2 = score_type_a(evidence, config)

        assert score1 == score2, "Same input must produce same score"
        assert metadata1 == metadata2, "Same input must produce same metadata"

# ============================================================================
# AGGREGATION MODULE CONTRACTS
# ============================================================================
class TestAggregationContracts:
    """Test aggregation module contract enforcement."""

    def test_dimension_aggregator_precondition_monolith(self):
        """DimensionAggregator requires valid monolith structure."""
        from saaaaaa.core.aggregation import DimensionAggregator

        # Valid monolith (minimal structure)
        valid_monolith = {
            "questions": [],
            "rubric": {
                "dimension": {
                    "thresholds": {
                        "EXCELENTE": 0.85,
                        "BUENO": 0.70,
                        "ACEPTABLE": 0.55,
                        "INSUFICIENTE": 0.0,
                    }
                }
            }
        }

        aggregator = DimensionAggregator(valid_monolith, abort_on_insufficient=False)
        assert aggregator.monolith == valid_monolith

        # Invalid monolith - not a dict
        with pytest.raises(Exception):
            DimensionAggregator("not a dict", abort_on_insufficient=False)  # type: ignore

    def test_dimension_aggregator_postcondition_score_range(self):
        """Dimension aggregation postcondition: score in [0, 3]."""
        from saaaaaa.core.aggregation import DimensionAggregator, ScoredResult

        monolith = {
            "questions": [],
            "rubric": {
                "dimension": {
                    "thresholds": {
                        "EXCELENTE": 0.85,
                        "BUENO": 0.70,
                        "ACEPTABLE": 0.55,
                        "INSUFICIENTE": 0.0,
                    }
                }
            }
        }

        aggregator = DimensionAggregator(monolith, abort_on_insufficient=False)

        # Create scored results
        scored_results = [
            ScoredResult(
                question_global=i,
                base_slot=f"P1-D1-Q{i:03d}",
                policy_area="P1",
                dimension="D1",
                score=2.5,
                quality_level="BUENO",
                evidence={},
                raw_results={},
            )
            for i in range(1, 6)
        ]

        result = aggregator.aggregate_dimension(
            dimension_id="D1",
            area_id="P1",
            scored_results=scored_results,
        )

        assert 0.0 <= result.score <= 3.0, f"Dimension score {result.score} out of range"

    def test_aggregation_invariant_weights_sum_to_one(self):
        """Aggregation invariant: weights must sum to 1.0."""
        from saaaaaa.core.aggregation import DimensionAggregator

        monolith = {
            "questions": [],
            "rubric": {
                "dimension": {
                    "thresholds": {
                        "EXCELENTE": 0.85,
                        "BUENO": 0.70,
                        "ACEPTABLE": 0.55,
                        "INSUFICIENTE": 0.0,
                    }
                }
            }
        }

        aggregator = DimensionAggregator(monolith, abort_on_insufficient=False)

        # Valid weights
        weights = [0.2, 0.2, 0.2, 0.2, 0.2]
        valid, msg = aggregator.validate_weights(weights)
        assert valid, f"Valid weights rejected: {msg}"

        # Invalid weights (don't sum to 1.0)
        weights = [0.3, 0.3, 0.3]
        valid, msg = aggregator.validate_weights(weights)
        assert not valid, "Invalid weights accepted"

# ============================================================================
# CONCURRENCY MODULE CONTRACTS
# ============================================================================

class TestConcurrencyContracts:
    """Test concurrency module contract enforcement."""

    def test_worker_pool_precondition_max_workers(self):
        """WorkerPool requires max_workers >= 1."""
        from saaaaaa.concurrency.concurrency import WorkerPool, WorkerPoolConfig

        # Valid precondition
        config = WorkerPoolConfig(max_workers=4, max_retries=3, backoff_factor=2.0)
        pool = WorkerPool(config)
        assert pool.config.max_workers == 4

        # Invalid precondition
        with pytest.raises(Exception):
            WorkerPoolConfig(max_workers=0, max_retries=3, backoff_factor=2.0)

    def test_worker_pool_postcondition_result_type(self):
        """WorkerPool postcondition: submit returns TaskResult."""
        from saaaaaa.concurrency.concurrency import TaskResult, WorkerPool,
WorkerPoolConfig

        config = WorkerPoolConfig(max_workers=2, max_retries=1, backoff_factor=1.0)

        def simple_task():
            return "success"

        with WorkerPool(config) as pool:
            result = pool.submit(simple_task, task_id="test-task")
            assert isinstance(result, TaskResult)
            assert result.result == "success"
    def test_worker_pool_invariant_determinism(self):
        """WorkerPool invariant: deterministic execution with same seed."""
        import random

        from saaaaaa.concurrency.concurrency import WorkerPool, WorkerPoolConfig

        config = WorkerPoolConfig(
            max_workers=2,
            max_retries=1,
            backoff_factor=1.0,
            deterministic_seed=12345,
        )

        def random_task():
            return random.random()

        # First execution
        with WorkerPool(config) as pool:
            pool.submit(random_task, task_id="random-1")

        # Second execution with same seed
        config2 = WorkerPoolConfig(
            max_workers=2,
            max_retries=1,
            backoff_factor=1.0,
            deterministic_seed=12345,
        )

        with WorkerPool(config2) as pool:
            pool.submit(random_task, task_id="random-1")

        # With deterministic seed, should get same result
        # Note: This may not work if random() is called elsewhere
        # assert value1 == value2, "Deterministic seed should produce same results"

# ============================================================================
# SEED FACTORY CONTRACTS
# ============================================================================

class TestSeedFactoryContracts:
    """Test seed factory contract enforcement."""

    def test_seed_factory_precondition_correlation_id(self):
        """SeedFactory requires non-empty correlation_id."""
        from saaaaaa.core.seed_factory import SeedFactory

        factory = SeedFactory()

        # Valid precondition
        seed = factory.create_deterministic_seed("run-001")
        assert isinstance(seed, int)
        assert 0 <= seed < 2**32

        # Invalid precondition - empty correlation_id
        # Note: Current implementation doesn't enforce this, should it?
        seed = factory.create_deterministic_seed("")
        assert isinstance(seed, int)  # Still returns a seed

    def test_seed_factory_postcondition_range(self):
        """SeedFactory postcondition: seed is 32-bit unsigned integer."""
        from saaaaaa.core.seed_factory import SeedFactory

        factory = SeedFactory()

        for i in range(10):
            seed = factory.create_deterministic_seed(f"run-{i}")
            assert isinstance(seed, int)
        assert 0 <= seed < 2**32, f"Seed {seed} out of 32-bit range"

    def test_seed_factory_invariant_determinism(self):
        """SeedFactory invariant: same input produces same seed."""
        from saaaaaa.core.seed_factory import SeedFactory

        factory = SeedFactory()

        # Same correlation_id
        seed1 = factory.create_deterministic_seed("run-123")
        seed2 = factory.create_deterministic_seed("run-123")
        assert seed1 == seed2, "Same correlation_id must produce same seed"

        # Same correlation_id and context
        seed3 = factory.create_deterministic_seed(
            "run-123",
            context={"question": "P1-D1-Q001", "area": "P1"}
        )
        seed4 = factory.create_deterministic_seed(
            "run-123",
            context={"question": "P1-D1-Q001", "area": "P1"}
        )
        assert seed3 == seed4, "Same input must produce same seed"

        # Different context produces different seed
        seed5 = factory.create_deterministic_seed(
            "run-123",
            context={"question": "P1-D1-Q002", "area": "P1"}
        )
        assert seed3 != seed5, "Different context must produce different seed"

# ============================================================================
# RECOMMENDATION ENGINE CONTRACTS
# ============================================================================

class TestRecommendationEngineContracts:
    """Test recommendation engine contract enforcement."""

    def test_recommendation_precondition_rules_schema(self):
        """RecommendationEngine requires valid rules schema."""

        # Valid rules (minimal structure)

        # Should not raise
        # Note: Need to check if constructor validates this

    def test_recommendation_postcondition_output_structure(self):
        """RecommendationEngine postcondition: output has required fields."""
        # This would test that generated recommendations have:
        # - intervention_id
        # - level
        # - trigger
        # - action
        # - expected_impact
        # etc.
        pass

# ============================================================================
# INTER-MODULE CONTRACT TESTS
# ============================================================================

class TestInterModuleContracts:
    """Test contracts between modules."""

    def test_scoring_to_aggregation_contract(self):
        """Test that scoring output matches aggregation input contract."""
        from saaaaaa.scoring.scoring import ScoredResult
        # Create ScoredResult (scoring output)
        scored = ScoredResult(
            question_global=1,
            base_slot="P1-D1-Q001",
            policy_area="P1",
            dimension="D1",
            score=2.5,
            quality_level="BUENO",
            evidence={"elements": [1, 2, 3], "confidence": 0.85},
            raw_results={},
        )

        # Verify it has all required fields for aggregation
        assert hasattr(scored, "question_global")
        assert hasattr(scored, "base_slot")
        assert hasattr(scored, "policy_area")
        assert hasattr(scored, "dimension")
        assert hasattr(scored, "score")
        assert hasattr(scored, "quality_level")
        assert hasattr(scored, "evidence")

        # Verify types
        assert isinstance(scored.score, (int, float))
        assert isinstance(scored.quality_level, str)
        assert isinstance(scored.evidence, dict)

if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_core_expected_counts.py =====
"""Test environment-configurable expected counts in core.py."""
import os
import pytest



# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="Count validation moved to structure_verification")

def test_expected_counts_default():
    """Test that default expected counts are loaded correctly."""
    # Import after setting env vars
    from saaaaaa.core.orchestrator.core import EXPECTED_QUESTION_COUNT,
EXPECTED_METHOD_COUNT

    # Default values should be loaded
    assert EXPECTED_QUESTION_COUNT == 305
    assert EXPECTED_METHOD_COUNT == 416


def test_expected_counts_custom():
    """Test that custom expected counts can be set via environment."""
    # Set custom env vars
    os.environ["EXPECTED_QUESTION_COUNT"] = "500"
    os.environ["EXPECTED_METHOD_COUNT"] = "600"

    try:
        # Re-import to pick up new values
        import importlib
        from saaaaaa.core.orchestrator import core
        importlib.reload(core)

        assert core.EXPECTED_QUESTION_COUNT == 500
        assert core.EXPECTED_METHOD_COUNT == 600
    finally:
        # Clean up
        os.environ.pop("EXPECTED_QUESTION_COUNT", None)
        os.environ.pop("EXPECTED_METHOD_COUNT", None)
        # Reload again to restore defaults
        importlib.reload(core)


def test_phase_timeout_default():
    """Test that default phase timeout is loaded correctly."""
    from saaaaaa.core.orchestrator.core import PHASE_TIMEOUT_DEFAULT

    # Default value should be 300 seconds
    assert PHASE_TIMEOUT_DEFAULT == 300


def test_phase_timeout_custom():
    """Test that custom phase timeout can be set via environment."""
    os.environ["PHASE_TIMEOUT_SECONDS"] = "600"

    try:
        import importlib
        from saaaaaa.core.orchestrator import core
        importlib.reload(core)

        assert core.PHASE_TIMEOUT_DEFAULT == 600
    finally:
        os.environ.pop("PHASE_TIMEOUT_SECONDS", None)
        importlib.reload(core)

===== FILE: tests/test_core_monolith_hash.py =====
"""Test stable content-based hash for monolith data in core.py."""
import hashlib
import json

import pytest



# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="Hash validation now part of
questionnaire_validation")

def test_monolith_hash_reproducibility():
    """Test that same monolith data produces same hash across runs."""
    monolith = {
        "version": "1.0.0",
        "blocks": {
            "micro_questions": [
                {"question_id": "Q1", "question_global": "Q1 Global", "base_slot":
"slot1"}
            ],
            "meso_questions": [],
            "macro_question": {}
        }
    }

    # Compute hash twice
    hash1 = hashlib.sha256(
        json.dumps(monolith, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
    ).hexdigest()

    hash2 = hashlib.sha256(
        json.dumps(monolith, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
    ).hexdigest()

    assert hash1 == hash2, "Hash should be reproducible for identical data"
```

```
def test_monolith_hash_changes_with_mutation():
    """Test that hash changes when data is mutated."""
    monolith1 = {
        "version": "1.0.0",
        "blocks": {
            "micro_questions": [
                {"question_id": "Q1", "question_global": "Q1 Global", "base_slot":
"slot1"}
            ]
        }
    }

    monolith2 = {
        "version": "1.0.0",
        "blocks": {
            "micro_questions": [
                {"question_id": "Q2", "question_global": "Q2 Global", "base_slot":
"slot2"}
            ]
        }
    }

    hash1 = hashlib.sha256(
        json.dumps(monolith1, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
    ).hexdigest()

    hash2 = hashlib.sha256(
        json.dumps(monolith2, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
    ).hexdigest()

    assert hash1 != hash2, "Hash should change when data is mutated"


def test_monolith_hash_key_order_independent():
    """Test that hash is independent of key order in dict."""
    monolith1 = {
        "version": "1.0.0",
        "blocks": {"micro_questions": []},
        "schema_version": "1.0"
    }

    monolith2 = {
        "schema_version": "1.0",
        "version": "1.0.0",
        "blocks": {"micro_questions": []}
    }

    hash1 = hashlib.sha256(
        json.dumps(monolith1, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
    ).hexdigest()

    hash2 = hashlib.sha256(
        json.dumps(monolith2, sort_keys=True, ensure_ascii=False, separators=(",",
":")).encode("utf-8")
    ).hexdigest()

    assert hash1 == hash2, "Hash should be independent of key order"

===== FILE: tests/test_cpp_table_extraction_none_handling.py =====
"""
Tests for CPP table extraction None value handling.

This test module validates the fixes for handling None values in table cells
during KPI and budget extraction.
```

```
OBSOLETE: cpp_ingestion.tables module no longer exists.
Table extraction functionality has been refactored/removed in SPC migration.
"""

import pytest

pytestmark = pytest.mark.skip(reason="obsolete - cpp_ingestion.tables module removed
during SPC refactor")

# Old import (no longer valid):
# from saaaaaa.processing.cpp_ingestion.tables import TableExtractor


class TestTableExtractionNoneHandling:
    """Test suite for None value handling in table extraction."""

    @pytest.fixture
    def extractor(self):
        """Create a TableExtractor instance."""
        return TableExtractor()

    def test_extract_kpis_with_none_values(self, extractor):
        """Test KPI extraction with None values in cells."""
        table = {
            "table_id": "test_table_1",
            "page": 1,
            "headers": ["indicador", "línea base", "meta", "unidad", "año"],
            "data_rows": [
                ["Tasa de cobertura", "85%", "95%", "%", "2028"],
                [None, "100", "150", None, "2027"],  # None values
                ["Población atendida", None, "5000", "personas", None],  # More None
values
                [None, None, None, None, None],  # All None
            ],
        }

        kpis = extractor._extract_kpis(table)

        # Should extract KPIs even with None values
        assert len(kpis) >= 2, "Should extract at least 2 KPIs with valid data"

        # First row - all values present
        kpi1 = kpis[0]
        assert kpi1["indicator"] == "Tasa de cobertura"
        assert kpi1["unit"] == "%"

        # Second row - None indicator should use first non-empty cell or "Unknown"
        kpi2 = kpis[1]
        assert kpi2["indicator"] in ["100", "Unknown"]  # Should handle None gracefully

        # Third row - None baseline and year should not cause errors
        kpi3 = kpis[2]
        assert kpi3["indicator"] == "Población atendida"
        assert kpi3.get("unit", "") == "personas"

    def test_extract_budgets_with_none_values(self, extractor):
        """Test budget extraction with None values in cells."""
        table = {
            "table_id": "budget_table_1",
            "page": 2,
            "headers": ["fuente", "uso", "monto", "año"],
            "data_rows": [
                ["SGP Educación", "Infraestructura", "$5,000,000,000", "2024"],
                [None, "Docentes", "$2,000,000,000", None],  # None values
                ["Regalías", None, None, "2025"],  # None use and amount
                [None, None, None, None],  # All None
            ],
        }
        budgets = extractor._extract_budgets(table)

        # Should extract budgets even with None values
        assert len(budgets) >= 2, "Should extract at least 2 budgets with valid data"

        # First row - all values present
        budget1 = budgets[0]
        assert budget1["source"] == "SGP Educación"
        assert budget1["use"] == "Infraestructura"

        # Second row - None source should default to "Unknown"
        budget2 = budgets[1]
        assert budget2["source"] == "Unknown"
        assert budget2["use"] == "Docentes"

        # Third row - None values should not cause AttributeError
        budget3 = budgets[2]
        assert budget3["source"] == "Regalías"
        # None use should be handled gracefully

    def test_extract_kpis_empty_cells_in_row(self, extractor):
        """Test KPI extraction with empty strings and None mixed."""
        table = {
            "table_id": "mixed_table",
            "page": 1,
            "headers": ["indicador", "baseline", "target"],
            "data_rows": [
                ["", None, "100"],  # Empty string and None
                [None, "", None],  # None and empty string
                ["Valid Indicator", None, None],  # Valid indicator with None values
            ],
        }

        kpis = extractor._extract_kpis(table)

        # Should handle mixed empty/None values without errors
        assert isinstance(kpis, list)

        # Should extract at least the row with valid indicator
        valid_kpis = [kpi for kpi in kpis if kpi.get("indicator") == "Valid Indicator"]
        assert len(valid_kpis) == 1

    def test_extract_budgets_no_strip_error(self, extractor):
        """Test that None values don't cause AttributeError: 'NoneType' object has no
attribute 'strip'."""
        table = {
            "table_id": "none_test",
            "page": 1,
            "headers": ["fuente", "uso", "monto"],
            "data_rows": [
                [None, None, None],
                [None, "Valid Use", "$1000"],
                ["Valid Source", None, "$2000"],
            ],
        }

        # This should not raise AttributeError
        try:
            budgets = extractor._extract_budgets(table)
            assert isinstance(budgets, list)
            # Should have extracted some budgets with valid data
            assert len(budgets) >= 1
        except AttributeError as e:
            if "'NoneType' object has no attribute 'strip'" in str(e):
                pytest.fail("None value handling failed - AttributeError on .strip()")
            raise
    def test_year_extraction_with_none(self, extractor):
        """Test year extraction when cell value is None."""
        table = {
            "table_id": "year_test",
            "page": 1,
            "headers": ["indicador", "año"],
            "data_rows": [
                ["Indicator 1", "2024"],  # Valid year
                ["Indicator 2", None],  # None year
                ["Indicator 3", ""],  # Empty year
                ["Indicator 4", "No year here"],  # Invalid year format
            ],
        }

        kpis = extractor._extract_kpis(table)

        # Should handle None/empty/invalid years without errors
        assert len(kpis) == 4

        # First should have year
        assert kpis[0].get("year") == 2024

        # Others should not have year or should handle gracefully
        for i in range(1, 4):
            # Should not raise error and year should be None or not present
            year = kpis[i].get("year")
            assert year is None or isinstance(year, int)

    def test_parse_numeric_with_none(self, extractor):
        """Test _parse_numeric helper with None input."""
        # Should handle None without errors
        result = extractor._parse_numeric(None)
        assert result is None or result == 0.0

        # Should handle empty string
        result = extractor._parse_numeric("")
        assert result is None or result == 0.0

        # Should still parse valid numbers
        result = extractor._parse_numeric("85%")
        assert result is not None
        assert result > 0

    def test_parse_currency_with_none(self, extractor):
        """Test _parse_currency helper with None input."""
        # Should handle None without errors
        result = extractor._parse_currency(None)
        assert result is None

        # Should handle empty string
        result = extractor._parse_currency("")
        assert result is None

        # Should still parse valid currency
        result = extractor._parse_currency("$5,000,000")
        assert result is not None
        assert result > 0


class TestTableExtractionIntegration:
    """Integration tests for table extraction with real-world scenarios."""

    @pytest.fixture
    def extractor(self):
        """Create a TableExtractor instance."""
        return TableExtractor()

    def test_extract_from_raw_objects_with_none_cells(self, extractor):
        """Test extract method with raw_objects containing None values."""
        raw_objects = {
            "pages": [
                {
                    "page_number": 1,
                    "tables": [
                        {
                            "table_id": "table_1",
                            "page": 1,
                            "headers": ["Indicador", "Meta"],
                            "rows": [
                                ["Indicador", "Meta"],  # Header row
                                ["Cobertura", "95%"],
                                [None, "100"],  # None cell
                                ["Población", None],  # None cell
                            ],
                        }
                    ],
                }
            ]
        }

        # This should not raise AttributeError
        try:
            result = extractor.extract(raw_objects)
            assert isinstance(result, dict)
            assert "tables" in result
            assert "kpis" in result
            assert "budgets" in result
        except AttributeError as e:
            if "'NoneType' object has no attribute 'strip'" in str(e):
                pytest.fail("Integration test failed - None handling issue")
            raise

===== FILE: tests/test_dashboard_static.py =====
"""Static contract tests for the AtroZ dashboard HTML."""

import re
from html.parser import HTMLParser
from pathlib import Path


ROOT = Path(__file__).resolve().parent.parent
HTML_PATH = ROOT / "src" / "saaaaaa" / "api" / "static" / "index.html"


class IdCollector(HTMLParser):
    def __init__(self) -> None:
        super().__init__()
        self.ids: list[str] = []

    def handle_starttag(self, tag, attrs):  # type: ignore[override]
        for key, value in attrs:
            if key == "id" and value:
                self.ids.append(value)


def _load_html() -> str:
    assert HTML_PATH.exists(), "Dashboard HTML should exist"
    return HTML_PATH.read_text(encoding="utf-8")


def test_dashboard_surfaces_expert_sections():
    parser = IdCollector()
    parser.feed(_load_html())

    expected_ids = {
        "designBlueprints",
        "eliteGallery",
        "metricStack",
        "latencySpark",
        "managementLayer",
        "stealthEntry",
        "conceptTribunal",
        "tribunalVoices",
        "integralConcept",
        "conceptNarrative",
        "tribunalTags",
        "conceptText",
    }
    missing = expected_ids.difference(parser.ids)
    assert not missing, f"Missing IDs in dashboard: {missing}"


def test_concept_tribunal_declares_integral_view():
    content = _load_html()

    for name in ["Doris Salcedo", "O. de Sagazan", "Adorno"]:
        assert name in content, f"Tribunal voice missing: {name}"

    assert "Concepto integral del dashboard" in content, "Integral concept statement
should be present"


def test_reservoir_and_blueprints_have_depth():
    content = _load_html()
    reservoir_block = re.search(r"const graphReservoir = \[(.*?)\];", content, flags=re.S)
    assert reservoir_block, "Graph reservoir should be declared"
    reservoir_names = re.findall(r"name:\s*'", reservoir_block.group(1))
    assert len(reservoir_names) >= 24, "Reservoir should expose dozens of graph options"

    blueprint_block = re.search(r"const designBlueprints = \[(.*?)\];", content,
flags=re.S)
    assert blueprint_block, "Design blueprints must be present"
    blueprint_items = re.findall(r"title:\s*'", blueprint_block.group(1))
    assert len(blueprint_items) >= 5, "Need multiple expert-level blueprints"


def test_latency_history_seeded_for_metrics():
    content = _load_html()
    history_seed = re.search(r"latency: Array.from\(\{ length: (\d+) \}", content)
    assert history_seed, "Telemetry history seed must exist"
    assert int(history_seed.group(1)) >= 32, "Telemetry history should start with a robust
 baseline"


===== FILE: tests/test_defensive_signatures.py =====
"""
Test defensive function signatures for argument mismatch resilience.

This test suite validates that the three functions identified in the problem statement
can gracefully handle unexpected or missing arguments without raising errors.
"""

import pytest

class TestDefensiveSignatures:
    """Test suite for defensive function signature implementations."""

    def test_is_likely_header_with_unexpected_kwargs(self):
        """Test _is_likely_header accepts and ignores unexpected keyword arguments."""
        # Import late to avoid dependency issues in test environment
        try:
            from saaaaaa.analysis.financiero_viabilidad_tablas import
PDETMunicipalPlanAnalyzer
        except ImportError:
            pytest.skip("financiero_viabilidad_tablas module not available")

        # Mock the dependencies to avoid initialization issues
        with pytest.raises(RuntimeError):
            # We expect this to fail during init due to missing spacy model
            # but we're testing the signature, not the initialization
            PDETMunicipalPlanAnalyzer(use_gpu=False)

    def test_is_likely_header_signature_accepts_kwargs(self):
        """Test that _is_likely_header method signature accepts **kwargs."""
        try:
            import inspect

            from saaaaaa.analysis.financiero_viabilidad_tablas import
PDETMunicipalPlanAnalyzer

            # Get the signature
            sig = inspect.signature(PDETMunicipalPlanAnalyzer._is_likely_header)
            params = sig.parameters

            # Verify it has **kwargs
            assert 'kwargs' in params, "_is_likely_header should accept **kwargs"
            assert params['kwargs'].kind == inspect.Parameter.VAR_KEYWORD, \
                "kwargs should be VAR_KEYWORD type"
        except ImportError:
            pytest.skip("financiero_viabilidad_tablas module not available")

    def test_analyze_causal_dimensions_signature_optional_sentences(self):
        """Test that _analyze_causal_dimensions has optional sentences parameter."""
        try:
            import inspect

            from saaaaaa.processing.policy_processor import IndustrialPolicyProcessor

            # Get the signature
            sig = inspect.signature(IndustrialPolicyProcessor._analyze_causal_dimensions)
            params = sig.parameters

            # Verify sentences is optional
            assert 'sentences' in params, "_analyze_causal_dimensions should have
sentences parameter"
            assert params['sentences'].default is not inspect.Parameter.empty, \
                "sentences parameter should have a default value (None)"
        except ImportError:
            pytest.skip("policy_processor module not available")

    def test_bayesian_mechanism_inference_init_accepts_kwargs(self):
        """Test that BayesianMechanismInference.__init__ accepts **kwargs."""
        try:
            import inspect

            from saaaaaa.analysis.derek_beach import BayesianMechanismInference

            # Get the signature
            sig = inspect.signature(BayesianMechanismInference.__init__)
            params = sig.parameters

            # Verify it has **kwargs
            assert 'kwargs' in params, "BayesianMechanismInference.__init__ should accept
**kwargs"
            assert params['kwargs'].kind == inspect.Parameter.VAR_KEYWORD, \
                "kwargs should be VAR_KEYWORD type"
        except ImportError:
            pytest.skip("derek_beach module not available")

    def test_defensive_warning_logging(self, caplog):
        """Test that unexpected arguments trigger warning logs."""
        try:
            from saaaaaa.analysis.financiero_viabilidad_tablas import
PDETMunicipalPlanAnalyzer

            # This test would require mocking the entire initialization chain
            # For now, we verify the signature accepts the pattern
            pytest.skip("Full integration test requires mocked dependencies")
        except ImportError:
            pytest.skip("financiero_viabilidad_tablas module not available")

class TestSignatureDocumentation:
    """Test that defensive signatures are properly documented."""

    def test_is_likely_header_docstring_mentions_kwargs(self):
        """Verify _is_likely_header docstring explains **kwargs handling."""
        try:
            from saaaaaa.analysis.financiero_viabilidad_tablas import
PDETMunicipalPlanAnalyzer

            docstring = PDETMunicipalPlanAnalyzer._is_likely_header.__doc__
            assert docstring is not None, "Method should have a docstring"
            assert '**kwargs' in docstring or 'kwargs' in docstring, \
                "Docstring should document **kwargs parameter"
            assert 'backward compatibility' in docstring.lower() or 'ignored' in
docstring.lower(), \
                "Docstring should explain that extra kwargs are ignored"
        except ImportError:
            pytest.skip("financiero_viabilidad_tablas module not available")

    def test_analyze_causal_dimensions_docstring_explains_optional(self):
        """Verify _analyze_causal_dimensions docstring explains optional sentences."""
        try:
            from saaaaaa.processing.policy_processor import IndustrialPolicyProcessor

            docstring = IndustrialPolicyProcessor._analyze_causal_dimensions.__doc__
            assert docstring is not None, "Method should have a docstring"
            assert 'optional' in docstring.lower() or 'Optional' in docstring, \
                "Docstring should mention sentences is optional"
            assert 'auto' in docstring.lower() or 'extract' in docstring.lower(), \
                "Docstring should explain auto-extraction behavior"
        except ImportError:
            pytest.skip("policy_processor module not available")

    def test_bayesian_init_docstring_mentions_kwargs(self):
        """Verify BayesianMechanismInference.__init__ docstring explains **kwargs."""
        try:
            from saaaaaa.analysis.derek_beach import BayesianMechanismInference

            docstring = BayesianMechanismInference.__init__.__doc__
            assert docstring is not None, "Method should have a docstring"
            assert '**kwargs' in docstring or 'kwargs' in docstring, \
                "Docstring should document **kwargs parameter"
            assert 'backward compatibility' in docstring.lower() or 'ignored' in
docstring.lower(), \
                "Docstring should explain that extra kwargs are ignored"
        except ImportError:
            pytest.skip("derek_beach module not available")

if __name__ == "__main__":
    pytest.main([__file__, "-v", "--tb=short"])

===== FILE: tests/test_dependency_lockdown.py =====
"""Tests for dependency lockdown enforcement.

Tests verify that:
1. HF_ONLINE environment variable controls online model access
2. Offline mode is enforced when HF_ONLINE=0 or not set
3. Clear errors are raised when online models are attempted offline
4. No silent fallback or "best effort" behavior
```

```
"""

import os
import pytest
from unittest import mock

from saaaaaa.core.dependency_lockdown import (
    DependencyLockdown,
    DependencyLockdownError,
    get_dependency_lockdown,
    reset_dependency_lockdown,
)


@pytest.fixture
def clean_env():
    """Fixture to clean up environment variables before and after each test."""
    # Clean before test
    for key in ["HF_ONLINE", "HF_HUB_OFFLINE", "TRANSFORMERS_OFFLINE"]:
        if key in os.environ:
            del os.environ[key]
    reset_dependency_lockdown()

    yield

    # Clean after test
    for key in ["HF_ONLINE", "HF_HUB_OFFLINE", "TRANSFORMERS_OFFLINE"]:
        if key in os.environ:
            del os.environ[key]
    reset_dependency_lockdown()


class TestDependencyLockdown:
    """Test dependency lockdown enforcement."""

    def test_offline_mode_default(self, clean_env):
        """Test that offline mode is enforced by default (HF_ONLINE not set)."""
        # Ensure HF_ONLINE is not set
        if "HF_ONLINE" in os.environ:
            del os.environ["HF_ONLINE"]

        lockdown = DependencyLockdown()

        assert lockdown.hf_allowed is False
        assert os.getenv("HF_HUB_OFFLINE") == "1"
        assert os.getenv("TRANSFORMERS_OFFLINE") == "1"

    def test_offline_mode_explicit_zero(self, clean_env):
        """Test that offline mode is enforced when HF_ONLINE=0."""
        os.environ["HF_ONLINE"] = "0"

        lockdown = DependencyLockdown()

        assert lockdown.hf_allowed is False
        assert os.getenv("HF_HUB_OFFLINE") == "1"
        assert os.getenv("TRANSFORMERS_OFFLINE") == "1"

    def test_online_mode_enabled(self, clean_env):
        """Test that online mode is enabled when HF_ONLINE=1."""
        os.environ["HF_ONLINE"] = "1"

        lockdown = DependencyLockdown()

        assert lockdown.hf_allowed is True
        # HF_HUB_OFFLINE should NOT be set to "1" in online mode
        # (it might be unset or set to something else)

    def test_check_online_model_access_offline_raises(self, clean_env):
        """Test that checking online model access raises error when offline."""
        os.environ["HF_ONLINE"] = "0"

        lockdown = DependencyLockdown()

        with pytest.raises(DependencyLockdownError) as exc_info:
            lockdown.check_online_model_access(
                model_name="test-model",
                operation="test operation"
            )

        assert "Online model download disabled" in str(exc_info.value)
        assert "test-model" in str(exc_info.value)
        assert "HF_ONLINE=1" in str(exc_info.value)
        assert "No fallback" in str(exc_info.value)

    def test_check_online_model_access_online_succeeds(self, clean_env):
        """Test that checking online model access succeeds when online."""
        os.environ["HF_ONLINE"] = "1"

        lockdown = DependencyLockdown()

        # Should not raise
        lockdown.check_online_model_access(
            model_name="test-model",
            operation="test operation"
        )

    def test_check_critical_dependency_missing_raises(self, clean_env):
        """Test that missing critical dependency raises error."""
        lockdown = DependencyLockdown()

        with pytest.raises(DependencyLockdownError) as exc_info:
            lockdown.check_critical_dependency(
                module_name="nonexistent_module_xyz",
                pip_package="nonexistent-package",
                phase="test_phase"
            )

        assert "Critical dependency" in str(exc_info.value)
        assert "nonexistent_module_xyz" in str(exc_info.value)
        assert "test_phase" in str(exc_info.value)
        assert "No degraded mode" in str(exc_info.value)

    def test_check_critical_dependency_present_succeeds(self, clean_env):
        """Test that present critical dependency check succeeds."""
        lockdown = DependencyLockdown()

        # Should not raise (os is always available)
        lockdown.check_critical_dependency(
            module_name="os",
            pip_package="builtin",
            phase="test_phase"
        )

    def test_check_optional_dependency_missing_returns_false(self, clean_env):
        """Test that missing optional dependency returns False and logs warning."""
        lockdown = DependencyLockdown()

        result = lockdown.check_optional_dependency(
            module_name="nonexistent_optional_xyz",
            pip_package="nonexistent-optional",
            feature="test_feature"
        )

        assert result is False

    def test_check_optional_dependency_present_returns_true(self, clean_env):
        """Test that present optional dependency returns True."""
        lockdown = DependencyLockdown()

        # os is always available
        result = lockdown.check_optional_dependency(
            module_name="os",
            pip_package="builtin",
            feature="test_feature"
        )

        assert result is True

    def test_get_mode_description(self, clean_env):
        """Test mode description contains expected keys."""
        os.environ["HF_ONLINE"] = "0"

        lockdown = DependencyLockdown()
        mode_desc = lockdown.get_mode_description()

        assert "hf_online_allowed" in mode_desc
        assert "hf_hub_offline" in mode_desc
        assert "transformers_offline" in mode_desc
        assert "mode" in mode_desc
        assert mode_desc["hf_online_allowed"] is False
        assert mode_desc["mode"] == "offline_enforced"

    def test_singleton_get_dependency_lockdown(self, clean_env):
        """Test that get_dependency_lockdown returns singleton instance."""
        os.environ["HF_ONLINE"] = "0"

        lockdown1 = get_dependency_lockdown()
        lockdown2 = get_dependency_lockdown()

        assert lockdown1 is lockdown2

    def test_reset_dependency_lockdown(self, clean_env):
        """Test that reset creates new instance."""
        os.environ["HF_ONLINE"] = "0"

        lockdown1 = get_dependency_lockdown()
        reset_dependency_lockdown()
        lockdown2 = get_dependency_lockdown()

        assert lockdown1 is not lockdown2


class TestEmbeddingPolicyIntegration:
    """Test that embedding policy respects dependency lockdown."""

    def test_embedding_model_init_offline_no_cache_raises(self, clean_env):
        """Test that embedding model init raises error when offline and model not
cached."""
        os.environ["HF_ONLINE"] = "0"

        from saaaaaa.processing.embedding_policy import (
            PolicyEmbeddingConfig,
            PolicyAnalysisEmbedder,
        )

        config = PolicyEmbeddingConfig(
            embedding_model="fake-model-that-does-not-exist"
        )

        # Mock _is_model_cached to return False
        with mock.patch(
            "saaaaaa.core.dependency_lockdown._is_model_cached",
            return_value=False
        ):
            with pytest.raises(DependencyLockdownError) as exc_info:
                PolicyAnalysisEmbedder(config)

            assert "Online model download disabled" in str(exc_info.value)
            assert "fake-model-that-does-not-exist" in str(exc_info.value)

    def test_cross_encoder_init_offline_no_cache_raises(self, clean_env):
        """Test that cross encoder init raises error when offline and model not cached."""
        os.environ["HF_ONLINE"] = "0"

        from saaaaaa.processing.embedding_policy import PolicyCrossEncoderReranker

        # Mock _is_model_cached to return False
        with mock.patch(
            "saaaaaa.core.dependency_lockdown._is_model_cached",
            return_value=False
        ):
            with pytest.raises(DependencyLockdownError) as exc_info:
                PolicyCrossEncoderReranker(
                    model_name="fake-cross-encoder-model"
                )

            assert "Online model download disabled" in str(exc_info.value)
            assert "fake-cross-encoder-model" in str(exc_info.value)


class TestOrchestratorIntegration:
    """Test that orchestrator initializes dependency lockdown."""

    def test_orchestrator_initializes_lockdown(self, clean_env):
        """Test that Orchestrator initializes dependency lockdown on construction."""
        os.environ["HF_ONLINE"] = "0"

        from saaaaaa.core.orchestrator import Orchestrator

        # Create minimal orchestrator
        orchestrator = Orchestrator()

        # Verify lockdown is initialized
        assert hasattr(orchestrator, "dependency_lockdown")
        assert orchestrator.dependency_lockdown is not None
        assert orchestrator.dependency_lockdown.hf_allowed is False

    def test_orchestrator_respects_hf_online(self, clean_env):
        """Test that Orchestrator respects HF_ONLINE setting."""
        os.environ["HF_ONLINE"] = "1"

        from saaaaaa.core.orchestrator import Orchestrator

        orchestrator = Orchestrator()

        assert orchestrator.dependency_lockdown.hf_allowed is True

===== FILE: tests/test_dependency_management.py =====
"""
Test dependency management system.

Verifies that the dependency management infrastructure works correctly.
"""

import json
import subprocess
import sys
from pathlib import Path

import pytest

from saaaaaa.config.paths import PROJECT_ROOT
```

```
class TestDependencyManagement:
    """Test suite for dependency management system."""

    @pytest.fixture
    def project_root(self):
        """Get project root directory."""
        return PROJECT_ROOT

    def test_requirements_files_exist(self, project_root):
        """Verify all required dependency files exist."""
        required_files = [
            "requirements-core.txt",
            "requirements-optional.txt",
            "requirements-dev.txt",
            "requirements-docs.txt",
            "requirements-all.txt",
            "constraints-new.txt",
        ]

        for filename in required_files:
            filepath = project_root / filename
            assert filepath.exists(), f"Missing required file: {filename}"

    def test_dependency_scripts_exist(self, project_root):
        """Verify all dependency management scripts exist and are executable."""
        import os

        required_scripts = [
            "scripts/audit_dependencies.py",
            "scripts/verify_importability.py",
            "scripts/generate_dependency_files.py",
            "scripts/compare_freeze_lock.py",
            "scripts/check_version_pins.py",
        ]

        for script_path in required_scripts:
            filepath = project_root / script_path
            assert filepath.exists(), f"Missing required script: {script_path}"

            # Check executable permission (cross-platform)
            assert os.access(filepath, os.R_OK), f"Script not readable: {script_path}"

    def test_documentation_exists(self, project_root):
        """Verify dependency documentation exists."""
        docs = [
            "DEPENDENCIES_AUDIT.md",
            "DEPENDENCIES_QUICKSTART.md",
        ]

        for doc in docs:
            filepath = project_root / doc
            assert filepath.exists(), f"Missing documentation: {doc}"

            # Check that documentation is not empty
            content = filepath.read_text()
            assert len(content) > 1000, f"Documentation too short: {doc}"

    def test_core_requirements_version_constraints(self, project_root):
        """Verify core requirements use appropriate version constraints.

        Most packages should use exact pins (==) for reproducibility,
        but certain packages with complex dependency chains (like ML/NLP libraries)
        may use constrained ranges (>=X,<Y) to allow pip to resolve dependencies.
        """
        requirements_file = project_root / "requirements-core.txt"
        # Packages allowed to have constrained ranges due to complex dependency chains
        allowed_ranges = {
            'fastapi', 'huggingface-hub', 'numpy', 'pandas', 'pydantic',
            'safetensors', 'scikit-learn', 'scipy', 'sentence-transformers',
            'tokenizers', 'transformers'
        }

        with open(requirements_file, 'r') as f:
            for line in f:
                line = line.strip()

                # Skip empty lines and comments
                if not line or line.startswith('#'):
                    continue

                # Skip -r includes
                if line.startswith('-r '):
                    continue

                # Extract package name
                pkg_name = line.split('=')[0].split('>')[0].split('<')[0].strip()

                if '==' in line:
                    # Good: exact pin
                    continue
                elif '>=' in line and '<' in line:
                    # Constrained range - check if allowed
                    if pkg_name.lower() not in allowed_ranges:
                        pytest.fail(
                            f"Package '{pkg_name}' uses a constrained range but is not in
allowed list.\n"
                            f"Line: {line}\n"
                            f"Either use exact pin (==) or add to allowed_ranges if needed
 for dependency resolution."
                        )
                elif any(op in line for op in ['>=', '~=', '<=', '<', '>', '*']):
                    # Unconstrained or unusual range
                    pytest.fail(
                        f"Core requirements must use exact pins (==) or constrained ranges
 (>=X,<Y), "
                        f"found problematic constraint in: {line}\n"
                        f"Expected format: package==X.Y.Z or package>=X.Y.Z,<A.B.C"
                    )

    def test_audit_script_runs(self, project_root):
        """Verify audit script can run successfully."""
        script = project_root / "scripts" / "audit_dependencies.py"

        # Run the script
        result = subprocess.run(
            [sys.executable, str(script)],
            cwd=project_root,
            capture_output=True,
            text=True,
        )

        # Script should run without crashing (may exit with 1 for missing deps)
        assert result.returncode in [0, 1], (
            f"Audit script crashed with code {result.returncode}\n"
            f"stdout: {result.stdout}\n"
            f"stderr: {result.stderr}"
        )

        # Check that report was generated
        report_file = project_root / "dependency_audit_report.json"
        assert report_file.exists(), "Audit report not generated"

        # Verify report structure
        with open(report_file, 'r') as f:
            report = json.load(f)

        assert "summary" in report
        assert "package_classification" in report
        assert "missing_packages" in report

    def test_version_pin_checker(self, project_root):
        """Verify version pin checker works."""
        script = project_root / "scripts" / "check_version_pins.py"
        core_reqs = project_root / "requirements-core.txt"

        result = subprocess.run(
            [sys.executable, str(script), str(core_reqs)],
            cwd=project_root,
            capture_output=True,
            text=True,
        )

        # Should pass for core requirements (all exact pins)
        assert result.returncode == 0, (
            f"Version pin check failed for core requirements:\n"
            f"{result.stdout}\n{result.stderr}"
        )

    def test_requirements_files_well_formed(self, project_root):
        """Verify requirements files are well-formed."""
        requirements_files = [
            "requirements-core.txt",
            "requirements-optional.txt",
            "requirements-dev.txt",
            "requirements-docs.txt",
        ]

        for filename in requirements_files:
            filepath = project_root / filename

            with open(filepath, 'r') as f:
                for line_num, line in enumerate(f, 1):
                    line = line.strip()

                    # Skip empty lines and comments
                    if not line or line.startswith('#'):
                        continue

                    # Skip -r includes
                    if line.startswith('-r '):
                        continue

                    # Verify line format
                    assert not line.startswith('=='), (
                        f"{filename}:{line_num} - Malformed line (starts with ==):
{line}\n"
                        f"Expected format: package==version (e.g., numpy==2.2.1)"
                    )

                    # Verify contains package name
                    assert len(line.split('==')) >= 1, (
                        f"{filename}:{line_num} - Invalid format: {line}\n"
                        f"Expected format: package==version (e.g., numpy==2.2.1)"
                    )

    def test_makefile_has_dependency_targets(self, project_root):
        """Verify Makefile includes dependency management targets."""
        makefile = project_root / "Makefile"

        with open(makefile, 'r') as f:
            content = f.read()
        required_targets = [
            "deps:verify",
            "deps:lock",
            "deps:audit",
            "deps:clean",
        ]

        for target in required_targets:
            assert target in content, f"Makefile missing target: {target}"

    def test_ci_workflow_exists(self, project_root):
        """Verify CI workflow for dependency gates exists."""
        workflow = project_root / ".github" / "workflows" / "dependency-gates.yml"

        assert workflow.exists(), "Dependency gates CI workflow missing"

        with open(workflow, 'r') as f:
            content = f.read()

        # Check for required gates
        required_gates = [
            "Missing Import Detection",
            "Importability Verification",
            "Open Range Detection",
            "Freeze vs Lock Comparison",
            "Security Vulnerability Scan",
        ]

        for gate in required_gates:
            assert gate in content, f"CI workflow missing gate: {gate}"

    def test_dependencies_audit_md_structure(self, project_root):
        """Verify DEPENDENCIES_AUDIT.md has required sections."""
        doc = project_root / "DEPENDENCIES_AUDIT.md"

        with open(doc, 'r') as f:
            content = f.read()

        required_sections = [
            "## Overview",
            "## Dependency Classification",
            "## Package Inventory",
            "## Installation Profiles",
            "## Verification Procedures",
            "## Adding New Dependencies",
            "## Known Issues & Risks",
            "## CI/CD Gates",
            "## Makefile Targets",
        ]

        for section in required_sections:
            assert section in content, f"Documentation missing section: {section}"


class TestDependencyScripts:
    """Test individual dependency management scripts."""

    def test_generate_dependency_files_idempotent(self, tmp_path):
        """Verify generate_dependency_files.py is idempotent."""
        # This test would require running the generator twice and comparing
        # For now, we just verify the script exists and can be imported
        script_path = Path(__file__).parent.parent / "scripts" /
"generate_dependency_files.py"
        assert script_path.exists()

    def test_compare_freeze_lock_detects_differences(self, tmp_path):
        """Verify compare_freeze_lock.py detects version differences."""
        # Create test files
        freeze_file = tmp_path / "freeze.txt"
        lock_file = tmp_path / "lock.txt"

        freeze_file.write_text("numpy==2.2.1\npandas==2.2.3\n")
        lock_file.write_text("numpy==2.2.0\npandas==2.2.3\n")  # Different numpy version

        script = Path(__file__).parent.parent / "scripts" / "compare_freeze_lock.py"

        result = subprocess.run(
            [sys.executable, str(script), str(freeze_file), str(lock_file)],
            capture_output=True,
            text=True,
        )

        # Should detect difference
        assert result.returncode == 1, "Should detect version mismatch"
        assert "numpy" in result.stdout, "Should report numpy mismatch"

    def test_compare_freeze_lock_passes_when_equal(self, tmp_path):
        """Verify compare_freeze_lock.py passes when files match."""
        # Create identical test files
        freeze_file = tmp_path / "freeze.txt"
        lock_file = tmp_path / "lock.txt"

        content = "numpy==2.2.1\npandas==2.2.3\n"
        freeze_file.write_text(content)
        lock_file.write_text(content)

        script = Path(__file__).parent.parent / "scripts" / "compare_freeze_lock.py"

        result = subprocess.run(
            [sys.executable, str(script), str(freeze_file), str(lock_file)],
            capture_output=True,
            text=True,
        )

        # Should pass
        assert result.returncode == 0, "Should pass when files match"
        assert "SUCCESS" in result.stdout


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_determinism.py =====
"""
Tests for Determinism Infrastructure

Validates that seed management ensures reproducible execution across
all stochastic operations.
"""

import pytest
from saaaaaa.core.orchestrator.seed_registry import (
    SeedRegistry,
    get_global_seed_registry,
    reset_global_seed_registry,
    SEED_VERSION,
)


class TestSeedRegistry:
    """Test SeedRegistry deterministic seed generation."""

    def test_same_inputs_produce_same_seed(self):
        """Test that identical inputs always produce identical seeds."""
        registry = SeedRegistry()
        seed1 = registry.get_seed("plan_2024", "exec_001", "numpy")
        seed2 = registry.get_seed("plan_2024", "exec_001", "numpy")

        assert seed1 == seed2
        assert isinstance(seed1, int)
        assert 0 <= seed1 < 2**32

    def test_different_components_produce_different_seeds(self):
        """Test that different components get different seeds."""
        registry = SeedRegistry()

        np_seed = registry.get_seed("plan_2024", "exec_001", "numpy")
        py_seed = registry.get_seed("plan_2024", "exec_001", "python")
        qt_seed = registry.get_seed("plan_2024", "exec_001", "quantum")

        # All seeds should be different
        assert np_seed != py_seed
        assert py_seed != qt_seed
        assert np_seed != qt_seed

    def test_different_policy_units_produce_different_seeds(self):
        """Test that different policy units get different seeds."""
        registry = SeedRegistry()

        seed1 = registry.get_seed("plan_2024", "exec_001", "numpy")
        seed2 = registry.get_seed("plan_2025", "exec_001", "numpy")

        assert seed1 != seed2

    def test_different_correlation_ids_produce_different_seeds(self):
        """Test that different correlation IDs get different seeds."""
        registry = SeedRegistry()

        seed1 = registry.get_seed("plan_2024", "exec_001", "numpy")
        seed2 = registry.get_seed("plan_2024", "exec_002", "numpy")

        assert seed1 != seed2

    def test_seed_caching_works(self):
        """Test that seeds are cached and reused."""
        registry = SeedRegistry()

        # First call
        seed1 = registry.get_seed("plan_2024", "exec_001", "numpy")
        audit_count_1 = len(registry.get_audit_log())

        # Second call (should use cache)
        seed2 = registry.get_seed("plan_2024", "exec_001", "numpy")
        audit_count_2 = len(registry.get_audit_log())

        assert seed1 == seed2
        assert audit_count_2 == audit_count_1  # No new audit entry

    def test_derive_seed_is_deterministic(self):
        """Test that derive_seed produces consistent output."""
        registry = SeedRegistry()

        seed1 = registry.derive_seed("test_material_123")
        seed2 = registry.derive_seed("test_material_123")

        assert seed1 == seed2
        assert isinstance(seed1, int)
        assert 0 <= seed1 < 2**32

    def test_derive_seed_different_inputs(self):
        """Test that different inputs produce different seeds."""
        registry = SeedRegistry()
        seed1 = registry.derive_seed("material_A")
        seed2 = registry.derive_seed("material_B")

        assert seed1 != seed2

    def test_audit_log_records_seeds(self):
        """Test that audit log records all seed generations."""
        registry = SeedRegistry()

        registry.get_seed("plan_2024", "exec_001", "numpy")
        registry.get_seed("plan_2024", "exec_001", "python")
        registry.get_seed("plan_2025", "exec_002", "quantum")

        audit_log = registry.get_audit_log()

        assert len(audit_log) == 3
        assert audit_log[0].policy_unit_id == "plan_2024"
        assert audit_log[0].component == "numpy"
        assert audit_log[1].component == "python"
        assert audit_log[2].policy_unit_id == "plan_2025"
        assert all(record.seed_version == SEED_VERSION for record in audit_log)

    def test_get_seeds_for_context(self):
        """Test getting all standard seeds for a context."""
        registry = SeedRegistry()

        seeds = registry.get_seeds_for_context("plan_2024", "exec_001")

        assert "numpy" in seeds
        assert "python" in seeds
        assert "quantum" in seeds
        assert "neuromorphic" in seeds
        assert "meta_learner" in seeds

        # All seeds should be different
        seed_values = list(seeds.values())
        assert len(seed_values) == len(set(seed_values))

    def test_clear_cache(self):
        """Test that clear_cache removes cached seeds."""
        registry = SeedRegistry()

        seed1 = registry.get_seed("plan_2024", "exec_001", "numpy")
        registry.clear_cache()

        # After clearing cache, should generate again (audit log grows)
        audit_count_before = len(registry.get_audit_log())
        seed2 = registry.get_seed("plan_2024", "exec_001", "numpy")
        audit_count_after = len(registry.get_audit_log())

        assert seed1 == seed2  # Same seed value
        assert audit_count_after > audit_count_before  # New audit entry

    def test_get_manifest_entry(self):
        """Test manifest entry generation."""
        registry = SeedRegistry()

        registry.get_seed("plan_2024", "exec_001", "numpy")
        registry.get_seed("plan_2024", "exec_001", "python")

        manifest = registry.get_manifest_entry("plan_2024", "exec_001")

        assert manifest["seed_version"] == SEED_VERSION
        assert manifest["seeds_generated"] == 2
        assert manifest["policy_unit_id"] == "plan_2024"
        assert manifest["correlation_id"] == "exec_001"
        assert "numpy" in manifest["seeds_by_component"]
        assert "python" in manifest["seeds_by_component"]


class TestGlobalSeedRegistry:
    """Test global seed registry singleton."""

    def test_get_global_registry(self):
        """Test getting global registry instance."""
        reset_global_seed_registry()

        registry1 = get_global_seed_registry()
        registry2 = get_global_seed_registry()

        assert registry1 is registry2  # Same instance

    def test_reset_global_registry(self):
        """Test resetting global registry."""
        reset_global_seed_registry()

        registry1 = get_global_seed_registry()
        registry1.get_seed("plan_2024", "exec_001", "numpy")

        reset_global_seed_registry()

        registry2 = get_global_seed_registry()
        assert registry2 is not registry1  # New instance
        assert len(registry2.get_audit_log()) == 0  # Empty audit log


class TestDeterminismAcrossRuns:
    """Test that same seeds produce identical results across runs."""

    def test_same_seed_different_registries(self):
        """Test that different registry instances produce same seeds."""
        registry1 = SeedRegistry()
        registry2 = SeedRegistry()

        seed1 = registry1.get_seed("plan_2024", "exec_001", "numpy")
        seed2 = registry2.get_seed("plan_2024", "exec_001", "numpy")

        assert seed1 == seed2

    def test_reproducible_numpy_random(self):
        """Test that NumPy RNG with same seed produces same output."""
        try:
            import numpy as np
        except ImportError:
            pytest.skip("NumPy not available")

        registry = SeedRegistry()
        seed = registry.get_seed("plan_2024", "exec_001", "numpy")

        # First run
        rng1 = np.random.default_rng(seed)
        values1 = rng1.random(10)

        # Second run with same seed
        rng2 = np.random.default_rng(seed)
        values2 = rng2.random(10)

        assert np.allclose(values1, values2)

    def test_reproducible_python_random(self):
        """Test that Python random with same seed produces same output."""
        import random

        registry = SeedRegistry()
        seed = registry.get_seed("plan_2024", "exec_001", "python")
        # First run
        random.seed(seed)
        values1 = [random.random() for _ in range(10)]

        # Second run with same seed
        random.seed(seed)
        values2 = [random.random() for _ in range(10)]

        assert values1 == values2


if __name__ == "__main__":
    pytest.main([__file__, "-v"])

===== FILE: tests/test_embedding_policy_contracts.py =====
"""Contract and property-based tests for PolicyAnalysisEmbedder._filter_by_pdq."""

from __future__ import annotations

import logging
import sys
import types
from typing import Any

import pytest
from hypothesis import HealthCheck, given, settings
from hypothesis import strategies as st

def _ensure_sentence_transformer_stub() -> None:
    """Provide a lightweight stub so embedding_policy imports without heavy deps."""

    if "sentence_transformers" in sys.modules:
        return

    stub = types.ModuleType("sentence_transformers")

    class _Stub:  # pragma: no cover - defensive stub
        def __init__(self, *args: Any, **kwargs: Any) -> None:  # noqa: D401 - simple stub
            raise RuntimeError("SentenceTransformer stub should not be instantiated in
tests")

    stub.SentenceTransformer = _Stub
    stub.CrossEncoder = _Stub
    sys.modules["sentence_transformers"] = stub

_ensure_sentence_transformer_stub()

from saaaaaa.processing.embedding_policy import PolicyAnalysisEmbedder  # noqa: E402  -
imported after stub

@pytest.fixture()
def embedder_stub() -> PolicyAnalysisEmbedder:
    """Create an embedder instance without running heavy initialisation."""

    instance = PolicyAnalysisEmbedder.__new__(PolicyAnalysisEmbedder)
    instance._logger = logging.getLogger("test.PolicyAnalysisEmbedder")
    return instance

@pytest.fixture()
def pdq_filter() -> dict[str, str]:
    return {"policy": "P1", "dimension": "D1"}

def test_filter_by_pdq_contract_shape(embedder_stub: PolicyAnalysisEmbedder, pdq_filter:
dict[str, str]) -> None:
    """Given documented inputs, the output remains a list of semantic chunks."""

    chunk = {
        "content": "evidence",
        "metadata": {"source": "unit-test"},
        "pdq_context": {
            "policy": "P1",
            "dimension": "D1",
            "question_unique_id": "P1-D1-Q1",
            "question": 1,
            "rubric_key": "D1-Q1",
        },
    }

    result = embedder_stub._filter_by_pdq([chunk], pdq_filter)

    assert isinstance(result, list)
    assert result == [chunk]

def test_filter_by_pdq_missing_context_logs_error(
    embedder_stub: PolicyAnalysisEmbedder, pdq_filter: dict[str, str], caplog:
pytest.LogCaptureFixture
) -> None:
    """Missing pdq_context results in a standardised contract error code."""

    chunk = {"content": "irrelevant", "metadata": {}, "pdq_context": None}

    with caplog.at_level(logging.ERROR):
        result = embedder_stub._filter_by_pdq([chunk], pdq_filter)

    assert result == []
    assert (
        "ERR_CONTRACT_MISMATCH[fn=_filter_by_pdq, key='pdq_context', needed=True,
got=None, index=0]"
        in caplog.text
    )

@st.composite
def chunk_strategy(draw: st.DrawFn) -> Any:
    """Generate chunk payloads covering valid, missing, and malformed contexts."""

    variant = draw(st.sampled_from(["match", "mismatch", "missing", "nondict"]))

    if variant == "nondict":
        return draw(st.one_of(st.none(), st.integers(), st.text(max_size=8)))

    chunk: dict[str, Any] = {
        "content": draw(st.text(max_size=24)),
        "metadata": {"origin": draw(st.text(max_size=8))},
    }

    if variant == "match":
        chunk["pdq_context"] = {
            "policy": "P1",
            "dimension": "D1",
            "question_unique_id": "P1-D1-Q1",
            "question": 1,
            "rubric_key": "D1-Q1",
            "extra_field": draw(st.text(max_size=6)),
        }
        return chunk

    if variant == "mismatch":
        chunk["pdq_context"] = {
            "policy": draw(st.sampled_from(["P2", "P3"])),
            "dimension": draw(st.sampled_from(["D2", "D3"])),
            "question_unique_id": draw(st.text(min_size=1, max_size=6)),
            "question": draw(st.integers(min_value=1, max_value=99)),
            "rubric_key": draw(st.text(min_size=1, max_size=6)),
            "extra": draw(st.text(max_size=4)),
        }
        return chunk

    # missing / malformed pdq_context
    chunk["pdq_context"] = draw(
        st.one_of(
            st.none(),
            st.just({}),
            st.just({"policy": "P1"}),
            st.just({"dimension": "D1"}),
            st.just({"policy": None, "dimension": "D1"}),
            st.just({"policy": "P1", "dimension": None}),
            st.text(max_size=5),
            st.integers(min_value=-1, max_value=3),
        )
    )
    return chunk

@settings(
    max_examples=25,
    deadline=None,
    suppress_health_check=[
        HealthCheck.function_scoped_fixture,
        HealthCheck.too_slow,
    ],
)
@given(chunks=st.lists(chunk_strategy(), max_size=6))
def test_filter_by_pdq_property_based(
    embedder_stub: PolicyAnalysisEmbedder, pdq_filter: dict[str, str], chunks: list[Any]
) -> None:
    """Property-based check: the filter never breaks on malformed payloads."""

    result = embedder_stub._filter_by_pdq(chunks, pdq_filter)

    assert isinstance(result, list)
    assert all(chunk in chunks for chunk in result)
    for chunk in result:
        assert isinstance(chunk, dict)
        context = chunk["pdq_context"]
        assert context["policy"] == "P1"
        assert context["dimension"] == "D1"


===== FILE: tests/test_enhanced_argument_resolution.py =====
"""Tests for enhanced argument resolution with graph-aware intelligence.

This module tests the new enhanced argument resolution features including:
- Graph-aware context initialization
- Sophisticated segments resolution
- Graph object resolution (DiGraph for causal analysis)
- Graph node resolution (origen, destino for causal links)
- Statements resolution
- Graph construction helper
"""

from unittest.mock import Mock
import pytest


# Mark all tests in this module as outdated
pytestmark = pytest.mark.skip(reason="Argument resolution now in ArgRouter extended
tests")

from saaaaaa.core.orchestrator.executors import AdvancedDataFlowExecutor, D1Q1_Executor


class MockDoc:
    """Mock document for testing"""
    def __init__(self):
        self.raw_text = "This is a test document. It has multiple sentences."
        self.sentences = ["This is a test document.", "It has multiple sentences."]
        self.tables = []
        self.metadata = {}


class MockExecutor:
    """Mock method executor for testing"""
    def __init__(self):
        self.instances = {}

    def execute(self, class_name, method_name, **kwargs):
        return "mock_result"


class TestEnhancedArgumentContext:
    """Test enhanced argument context initialization"""

    def test_reset_argument_context_includes_graph_fields(self):
        """Test that reset includes graph-aware fields"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        ctx = executor._argument_context

        # Check enhanced graph-aware fields
        assert 'grafo' in ctx
        assert 'graph_nodes' in ctx
        assert 'graph_edges' in ctx
        assert 'statements' in ctx

        # Check enhanced segmentation fields
        assert 'segments' in ctx
        assert 'segment_metadata' in ctx

        # Check initial values
        assert ctx['grafo'] is None
        assert ctx['graph_nodes'] == []
        assert ctx['graph_edges'] == []
        assert ctx['statements'] == []
        assert ctx['segment_metadata'] == {}


class TestSegmentsResolution:
    """Test sophisticated segments resolution"""

    def test_segments_resolution_from_sentences(self):
        """Test that segments are resolved from sentences"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        # Resolve segments
        segments = executor._resolve_argument(
            name='segments',
            class_name='TestClass',
            method_name='test_method',
            doc=mock_doc,
            current_data=None,
            instance=Mock()
        )

        assert isinstance(segments, list)
        assert len(segments) > 0
        assert executor._argument_context['segment_metadata']['strategy'] ==
'sentence_based'

    def test_segments_resolution_from_text(self):
        """Test that segments are created from text when sentences unavailable"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()
        mock_doc.sentences = []  # No sentences available

        executor._reset_argument_context(mock_doc)

        # Resolve segments
        segments = executor._resolve_argument(
            name='segments',
            class_name='TestClass',
            method_name='test_method',
            doc=mock_doc,
            current_data=None,
            instance=Mock()
        )

        assert isinstance(segments, list)
        assert len(segments) > 0
        assert executor._argument_context['segment_metadata']['strategy'] ==
'semantic_split'


class TestGraphResolution:
    """Test graph object resolution"""

    def test_graph_resolution_returns_unset_when_unavailable(self):
        """Test that graph resolution returns _ARG_UNSET when unavailable"""
        from saaaaaa.core.orchestrator.executors import _ARG_UNSET

        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        # Create a mock instance without graph attributes
        mock_instance = Mock(spec=['some_method'])  # Spec without grafo/graph attributes

        # Resolve graph without any graph available
        grafo = executor._resolve_argument(
            name='grafo',
            class_name='TestClass',
            method_name='test_method',
            doc=mock_doc,
            current_data=None,
            instance=mock_instance
        )

        assert grafo is _ARG_UNSET

    def test_graph_fallback_creates_empty_digraph(self):
        """Test that graph fallback creates an empty NetworkX DiGraph"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        # Get fallback for graph
        try:
            grafo = executor._fallback_for(
                name='grafo',
                class_name='TestClass',
                method_name='test_method',
                instance=Mock()
            )

            # Check if NetworkX is available and graph was created
            if grafo is not None:
                import networkx as nx
                assert isinstance(grafo, nx.DiGraph)
                assert len(grafo.nodes()) == 0
        except ImportError:
            # NetworkX not available, should return None
            assert grafo is None


class TestNodeResolution:
    """Test graph node resolution (origen, destino)"""

    def test_origen_resolution_from_dict(self):
        """Test origen resolution from dictionary"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        current_data = {'origen': 'node_a', 'destino': 'node_b'}

        origen = executor._resolve_argument(
            name='origen',
            class_name='TestClass',
            method_name='test_method',
            doc=mock_doc,
            current_data=current_data,
            instance=Mock()
        )

        assert origen == 'node_a'

    def test_destino_resolution_from_tuple(self):
        """Test destino resolution from tuple"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        current_data = ('node_a', 'node_b')

        destino = executor._resolve_argument(
            name='destino',
            class_name='TestClass',
            method_name='test_method',
            doc=mock_doc,
            current_data=current_data,
            instance=Mock()
        )

        assert destino == 'node_b'

    def test_node_fallbacks(self):
        """Test node fallbacks return default node identifiers"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)
        origen = executor._fallback_for(
            name='origen',
            class_name='TestClass',
            method_name='test_method',
            instance=Mock()
        )

        destino = executor._fallback_for(
            name='destino',
            class_name='TestClass',
            method_name='test_method',
            instance=Mock()
        )

        assert origen == "node_0"
        assert destino == "node_1"


class TestStatementsResolution:
    """Test statements resolution"""

    def test_statements_resolution_from_list(self):
        """Test statements resolution from list data"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        current_data = ["Statement 1", "Statement 2", "Statement 3"]

        statements = executor._resolve_argument(
            name='statements',
            class_name='TestClass',
            method_name='test_method',
            doc=mock_doc,
            current_data=current_data,
            instance=Mock()
        )

        assert statements == current_data
        assert executor._argument_context['statements'] == current_data


class TestUpdateArgumentContext:
    """Test enhanced context update with graph-aware tracking"""

    def test_update_tracks_graph_from_teoria_cambio(self):
        """Test that DiGraph from TeoriaCambio is tracked"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        # Create a mock graph result
        try:
            import networkx as nx
            mock_graph = nx.DiGraph()
            mock_graph.add_edge('A', 'B')
            mock_graph.add_edge('B', 'C')

            executor._update_argument_context(
                method_key='TeoriaCambio.construir_grafo_causal',
                result=mock_graph,
                class_name='TeoriaCambio',
                method_name='construir_grafo_causal'
            )

            ctx = executor._argument_context
            assert ctx['grafo'] is mock_graph
            assert len(ctx['graph_nodes']) == 3
            assert len(ctx['graph_edges']) == 2
        except ImportError:
            pytest.skip("NetworkX not available")

    def test_update_tracks_segments(self):
        """Test that segments from segmentation methods are tracked"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        segments = ["Segment 1", "Segment 2", "Segment 3"]

        executor._update_argument_context(
            method_key='PolicyTextProcessor.segment_into_sentences',
            result=segments,
            class_name='PolicyTextProcessor',
            method_name='segment_into_sentences'
        )

        ctx = executor._argument_context
        assert ctx['segments'] == segments
        assert ctx['segment_metadata']['strategy'] == 'method_result'
        assert ctx['segment_metadata']['count'] == 3


class TestConstructCausalGraph:
    """Test graph construction helper"""

    def test_construct_causal_graph_creates_graph(self):
        """Test that construct_causal_graph creates a NetworkX DiGraph"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)

        statements = [
            "A causes B",
            "B results in C",
            "C because of D"
        ]

        try:
            graph = executor._construct_causal_graph(statements, Mock())

            if graph is not None:
                import networkx as nx
                assert isinstance(graph, nx.DiGraph)
                # Should have some nodes from causal extraction
                assert len(graph.nodes()) > 0
        except ImportError:
            pytest.skip("NetworkX not available")

    def test_construct_causal_graph_handles_non_causal_statements(self):
        """Test that non-causal statements are handled gracefully"""
        mock_method_executor = MockExecutor()
        executor = D1Q1_Executor(mock_method_executor)
        mock_doc = MockDoc()

        executor._reset_argument_context(mock_doc)
        statements = [
            "This is a statement",
            "Another statement",
            "Yet another one"
        ]

        try:
            graph = executor._construct_causal_graph(statements, Mock())

            if graph is not None:
                import networkx as nx
                assert isinstance(graph, nx.DiGraph)
                # May have isolated nodes but no edges
                assert len(graph.nodes()) >= 0
        except ImportError:
            pytest.skip("NetworkX not available")


if __name__ == '__main__':
    pytest.main([__file__, '-v'])

===== FILE: tests/test_enhanced_recommendations.py =====
#!/usr/bin/env python3
"""
End-to-end test of the recommendation engine integration with orchestrator.
Tests all 7 enhanced features across MICRO, MESO, and MACRO levels.
"""

import json
import sys

# Add parent directory to path
from saaaaaa.analysis.recommendation_engine import load_recommendation_engine
from saaaaaa.utils.paths import tmp_dir

def test_enhanced_recommendation_engine():
    """Test all 7 enhanced features"""
    print("=" * 80)
    print("TESTING ENHANCED RECOMMENDATION ENGINE v2.0")
    print("=" * 80)

    # Initialize engine with enhanced rules
    print("\n1. Initializing engine with enhanced rules...")
    engine = load_recommendation_engine(
        rules_path="config/recommendation_rules_enhanced.json",
        schema_path="rules/recommendation_rules_enhanced.schema.json"
    )
    print(f"   ✓ Loaded {len(engine.rules['rules'])} rules")
    print(f"   ✓ MICRO rules: {len(engine.rules_by_level['MICRO'])}")
    print(f"   ✓ MESO rules: {len(engine.rules_by_level['MESO'])}")
    print(f"   ✓ MACRO rules: {len(engine.rules_by_level['MACRO'])}")

    # Test MICRO recommendations
    print("\n2. Testing MICRO recommendations...")
    micro_scores = {
        'PA01-DIM01': 1.2,   # Below threshold (1.65)
        'PA01-DIM02': 1.5,   # Below threshold
        'PA02-DIM03': 2.1,   # Above threshold
        'PA05-DIM05': 1.3,   # Below threshold
    }

    micro_recs = engine.generate_micro_recommendations(micro_scores)
    print(f"   ✓ Generated {micro_recs.rules_matched} MICRO recommendations")
    print(f"   ✓ Evaluated {micro_recs.total_rules_evaluated} rules")

    # Validate first MICRO recommendation has all 7 features
    if micro_recs.recommendations:
        rec = micro_recs.recommendations[0]
        print(f"\n   Testing enhanced features on {rec.rule_id}:")

        # Feature 1: Template parameterization
        assert rec.template_id is not None, "Missing template_id"
        assert rec.template_params is not None, "Missing template_params"
        print(f"   ✓ Feature 1 - Template ID: {rec.template_id}")

        # Feature 2: Execution logic
        assert rec.execution is not None, "Missing execution"
        assert 'trigger_condition' in rec.execution, "Missing trigger_condition"
        assert 'requires_approval' in rec.execution, "Missing requires_approval"
        print(f"   ✓ Feature 2 - Execution: {rec.execution['trigger_condition'][:50]}...")

        # Feature 3: Measurable indicators
        assert 'formula' in rec.indicator, "Missing indicator formula"
        assert 'data_source' in rec.indicator, "Missing data_source"
        assert 'data_source_query' in rec.indicator, "Missing data_source_query"
        assert 'acceptable_range' in rec.indicator, "Missing acceptable_range"
        print(f"   ✓ Feature 3 - Formula: {rec.indicator['formula']}")
        print(f"   ✓ Feature 3 - Data source: {rec.indicator['data_source']}")

        # Feature 4: Unambiguous time horizons
        assert 'duration_months' in rec.horizon, "Missing duration_months"
        assert 'milestones' in rec.horizon, "Missing milestones"
        assert 'start_type' in rec.horizon, "Missing start_type"
        print(f"   ✓ Feature 4 - Duration: {rec.horizon['duration_months']} months")
        print(f"   ✓ Feature 4 - Milestones: {len(rec.horizon['milestones'])} defined")

        # Feature 5: Testable verification
        assert len(rec.verification) > 0, "No verification artifacts"
        assert isinstance(rec.verification[0], dict), "Verification not structured"
        assert 'id' in rec.verification[0], "Missing verification ID"
        assert 'type' in rec.verification[0], "Missing verification type"
        print(f"   ✓ Feature 5 - Verification: {len(rec.verification)} artifacts")
        print(f"   ✓ Feature 5 - First artifact: {rec.verification[0]['id']}")

        # Feature 6: Cost tracking
        assert rec.budget is not None, "Missing budget"
        assert 'estimated_cost_cop' in rec.budget, "Missing estimated_cost_cop"
        assert 'cost_breakdown' in rec.budget, "Missing cost_breakdown"
        assert 'funding_sources' in rec.budget, "Missing funding_sources"
        print(f"   ✓ Feature 6 - Budget: ${rec.budget['estimated_cost_cop']:,} COP")
        print(f"   ✓ Feature 6 - Breakdown: 
Personal={rec.budget['cost_breakdown']['personal']:,}, "
              f"Consultancy={rec.budget['cost_breakdown']['consultancy']:,}")

        # Feature 7: Authority mapping
        assert 'legal_mandate' in rec.responsible, "Missing legal_mandate"
        assert 'approval_chain' in rec.responsible, "Missing approval_chain"
        assert 'escalation_path' in rec.responsible, "Missing escalation_path"
        print(f"   ✓ Feature 7 - Legal mandate: 
{rec.responsible['legal_mandate'][:50]}...")
        print(f"   ✓ Feature 7 - Approval chain: {len(rec.responsible['approval_chain'])} 
levels")

    # Test MESO recommendations
    print("\n3. Testing MESO recommendations...")
    cluster_data = {
        'CL01': {'score': 72.0, 'variance': 0.25, 'weak_pa': 'PA02'},
        'CL02': {'score': 45.0, 'variance': 0.12, 'weak_pa': 'PA05'},
    }

    meso_recs = engine.generate_meso_recommendations(cluster_data)
    print(f"   ✓ Generated {meso_recs.rules_matched} MESO recommendations")

    if meso_recs.recommendations:
        rec = meso_recs.recommendations[0]
        print(f"   ✓ MESO recommendation: {rec.rule_id}")
        assert rec.budget is not None, "MESO missing budget"
        assert rec.execution is not None, "MESO missing execution"
        print(f"   ✓ MESO budget: ${rec.budget['estimated_cost_cop']:,} COP")

    # Test MACRO recommendations
    print("\n4. Testing MACRO recommendations...")
    macro_data = {
        'macro_band': 'SATISFACTORIO',
        'clusters_below_target': ['CL02', 'CL03'],
        'variance_alert': 'MODERADA',
        'priority_micro_gaps': ['PA01-DIM05', 'PA05-DIM04', 'PA04-DIM04', 'PA08-DIM05']
    }

    macro_recs = engine.generate_macro_recommendations(macro_data)
    print(f"   ✓ Generated {macro_recs.rules_matched} MACRO recommendations")

    if macro_recs.recommendations:
        rec = macro_recs.recommendations[0]
        print(f"   ✓ MACRO recommendation: {rec.rule_id}")
        assert rec.budget is not None, "MACRO missing budget"
        assert rec.execution is not None, "MACRO missing execution"
        print(f"   ✓ MACRO budget: ${rec.budget['estimated_cost_cop']:,} COP")

    # Test generate_all_recommendations (orchestrator integration point)
    print("\n5. Testing generate_all_recommendations (orchestrator integration)...")
    all_recs = engine.generate_all_recommendations(
        micro_scores=micro_scores,
        cluster_data=cluster_data,
        macro_data=macro_data
    )

    print(f"   ✓ MICRO: {len(all_recs['MICRO'].recommendations)} recommendations")
    print(f"   ✓ MESO: {len(all_recs['MESO'].recommendations)} recommendations")
    print(f"   ✓ MACRO: {len(all_recs['MACRO'].recommendations)} recommendations")

    # Export test
    print("\n6. Testing export functionality...")
    export_root = tmp_dir() / "recommendation_exports"
    export_root.mkdir(parents=True, exist_ok=True)
    output_json = export_root / "test_recommendations.json"
    output_md = export_root / "test_recommendations.md"

    engine.export_recommendations(all_recs, str(output_json), format='json')
    print(f"   ✓ Exported to JSON: {output_json}")

    engine.export_recommendations(all_recs, str(output_md), format='markdown')
    print(f"   ✓ Exported to Markdown: {output_md}")

    # Verify JSON export has enhanced fields
    with open(output_json, encoding='utf-8') as f:
        exported = json.load(f)

    if exported['MICRO']['recommendations']:
        first_rec = exported['MICRO']['recommendations'][0]
        assert 'execution' in first_rec, "Exported JSON missing execution"
        assert 'budget' in first_rec, "Exported JSON missing budget"
        print("   ✓ Exported JSON contains all enhanced fields")

    print("\n" + "=" * 80)
    print("✓ ALL TESTS PASSED - Enhanced recommendation engine is fully functional")
    print("=" * 80)
    print("\nSummary:")
    print(f"  - Total recommendations: {len(all_recs['MICRO'].recommendations) +
len(all_recs['MESO'].recommendations) + len(all_recs['MACRO'].recommendations)}")
    print("  - All 7 advanced features validated")
    print("  - Orchestrator integration point tested")
    print("  - Export formats working (JSON, Markdown)")
    return True

if __name__ == '__main__':
    try:
        success = test_enhanced_recommendation_engine()
        sys.exit(0 if success else 1)
    except Exception as e:
        print(f"\n✗ TEST FAILED: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)

===== FILE: tests/test_executor_contracts.py =====
import json
import sys
import types
from glob import glob

import pytest
from jsonschema import Draft7Validator

# Stub missing modules to allow importing orchestrator package without circular errors
contract_loader = types.ModuleType("contract_loader")
class _Dummy:
    pass
contract_loader.JSONContractLoader = _Dummy
contract_loader.LoadError = _Dummy
contract_loader.LoadResult = _Dummy
sys.modules.setdefault("saaaaaa.core.orchestrator.contract_loader", contract_loader)

core_stub = types.ModuleType("saaaaaa.core.orchestrator.core")
class _StubMethodExecutor:
    def execute(self, *args, **kwargs):
        return {}
core_stub.MethodExecutor = _StubMethodExecutor
class _StubPreprocessedDocument:
    def __init__(self, document_id, raw_text, sentences, tables, metadata):
        self.document_id = document_id
        self.raw_text = raw_text
        self.sentences = sentences
        self.tables = tables
        self.metadata = metadata
core_stub.PreprocessedDocument = _StubPreprocessedDocument
core_stub.AbortRequested = type("AbortRequested", (), {})
core_stub.AbortSignal = type("AbortSignal", (), {})
core_stub.ResourceLimits = type("ResourceLimits", (), {})
core_stub.PhaseInstrumentation = type("PhaseInstrumentation", (), {})
core_stub.PhaseResult = type("PhaseResult", (), {})
core_stub.MicroQuestionRun = type("MicroQuestionRun", (), {})
core_stub.ScoredMicroQuestion = type("ScoredMicroQuestion", (), {})
core_stub.Evidence = type("Evidence", (), {})
core_stub.Orchestrator = type("Orchestrator", (), {})
sys.modules.setdefault("saaaaaa.core.orchestrator.core", core_stub)

from saaaaaa.core.orchestrator.class_registry import build_class_registry,
ClassRegistryError
from saaaaaa.core.orchestrator.core import PreprocessedDocument, MethodExecutor
from saaaaaa.core.orchestrator.executors_contract import D1Q1_Executor_Contract


def test_contracts_validate_against_schema():
    schema_path = "config/executor_contract.schema.json"
    with open(schema_path, encoding="utf-8") as f:
        schema = json.load(f)
    validator = Draft7Validator(schema)

    for path in glob("config/executor_contracts/*.json"):
        with open(path, encoding="utf-8") as f:
            contract = json.load(f)
        errors = list(validator.iter_errors(contract))
        assert not errors, f"Schema validation failed for {path}: {[e.message for e in
errors]}"


def test_contract_methods_exist_in_registry():
    try:
        registry = build_class_registry()
    except ClassRegistryError as exc:  # pragma: no cover - environment-specific
        pytest.skip(f"class registry unavailable: {exc}")

    for path in glob("config/executor_contracts/*.json"):
        contract = json.loads(open(path, encoding="utf-8").read())
        for mi in contract.get("method_inputs", []):
            cls_name = mi["class"]
            method_name = mi["method"]
            assert cls_name in registry, f"{cls_name} not in class registry for {path}"
            cls = registry[cls_name]
            assert hasattr(cls, method_name), f"{cls_name}.{method_name} missing for
{path}"


def test_d1q1_executor_contract_smoke(monkeypatch):
    # Fake MethodExecutor to avoid heavy dependencies
    fake_me = MethodExecutor.__new__(MethodExecutor)
    def _fake_execute(class_name: str, method_name: str, **kwargs):
        return {"class": class_name, "method": method_name, "payload": kwargs}
    fake_me.execute = _fake_execute  # type: ignore[attr-defined]

    executor = D1Q1_Executor_Contract(
        fake_me,
        signal_registry=None,
        config=None,
        questionnaire_provider=None,
    )

    doc = PreprocessedDocument(
        document_id="doc1",
        raw_text="some text",
        sentences=["some text"],
        tables=[],
        metadata={"chunk_count": 1},
    )

    question_context = {
        "base_slot": "D1-Q1",
        "question_id": "D1-Q1",
        "question_global": 1,
        "policy_area_id": "PA01",
        "dimension_id": "D1",
        "cluster_id": "C1",
        "patterns": [],
        "expected_elements": [],
    }

    result = executor.execute(doc, fake_me, question_context=question_context)
    assert result["base_slot"] == "D1-Q1"
    assert "evidence" in result
    assert "analysis" in result["evidence"]
    assert result["validation"]["valid"] is True

===== FILE: tests/test_executors_coherence_smoke.py =====
"""
Smoke tests for executor docstring-execution coherence.

Validates that each executor's docstring accurately documents the methods
it executes in the correct order.
```

```
"""

import importlib
import re
import sys
import types
from typing import Dict, List, Set, Tuple

import pytest


# Stub orchestrator dependencies to avoid circular imports during test import
core_stub = types.ModuleType("saaaaaa.core.orchestrator.core")


class _FakeMethodExecutor:
    """Fake MethodExecutor that tracks method calls."""

    def __init__(self):
        self.calls: List[Tuple[str, str]] = []

    def execute(self, class_name: str, method_name: str, **payload):
        self.calls.append((class_name, method_name))
        # Return safe defaults for common return types
        return {"class": class_name, "method": method_name, "payload": payload}


core_stub.MethodExecutor = _FakeMethodExecutor
core_stub.AbortRequested = type("AbortRequested", (), {})
core_stub.AbortSignal = type("AbortSignal", (), {})
core_stub.ResourceLimits = type("ResourceLimits", (), {})
core_stub.PhaseInstrumentation = type("PhaseInstrumentation", (), {})
core_stub.PhaseResult = type("PhaseResult", (), {})
core_stub.MicroQuestionRun = type("MicroQuestionRun", (), {})
core_stub.ScoredMicroQuestion = type("ScoredMicroQuestion", (), {})
core_stub.Evidence = type("Evidence", (), {})
core_stub.PreprocessedDocument = type("PreprocessedDocument", (), {})
core_stub.Orchestrator = type("Orchestrator", (), {})
sys.modules["saaaaaa.core.orchestrator.core"] = core_stub

factory_stub = types.ModuleType("saaaaaa.core.orchestrator.factory")


def _fake_build_processor(me: _FakeMethodExecutor | None = None):
    bundle = types.SimpleNamespace()
    bundle.method_executor = me or _FakeMethodExecutor()
    return bundle


factory_stub.build_processor = _fake_build_processor
sys.modules["saaaaaa.core.orchestrator.factory"] = factory_stub

contract_loader = types.ModuleType("contract_loader")


class _Dummy:
    pass


contract_loader.JSONContractLoader = _Dummy
contract_loader.LoadError = _Dummy
contract_loader.LoadResult = _Dummy
sys.modules.setdefault("saaaaaa.core.orchestrator.contract_loader", contract_loader)

exec_mod = importlib.import_module("saaaaaa.core.orchestrator.executors")


def parse_docstring_methods(docstring: str) -> List[str]:
    """
    Parse method names from executor docstring.

    Expected format:
    Step N: Description - ClassName.method_name

    Returns list of "ClassName.method_name" strings in order.
    """
    if not docstring:
        return []

    methods = []
    # Match patterns like "Step N: ... - ClassName.method_name"
    step_pattern = re.compile(r"Step\s+\d+:.*?-\s+(\w+\.\w+)")
    # Also match old format "- ClassName.method_name"
    old_pattern = re.compile(r"^\s*-\s+(\w+\.\w+)\s*$", re.MULTILINE)

    # Try new format first
    step_matches = step_pattern.findall(docstring)
    if step_matches:
        methods = step_matches
    else:
        # Fall back to old format
        methods = old_pattern.findall(docstring)

    return methods


def get_executed_methods(executor_class, context: Dict) -> List[str]:
    """
    Execute an executor and return the list of methods it called.

    Returns list of "ClassName.method_name" strings in execution order.
    """
    fake_me = _FakeMethodExecutor()

    # Create executor instance
    executor = executor_class(
        executor_id="TEST-Q1",
        config={},
        method_executor=fake_me
    )

    try:
        # Execute with minimal context
        executor.execute(context)
    except Exception:
        # Some executors may fail due to missing data, that's OK
        # We just want to see what methods were attempted
        pass

    # Return methods in format "ClassName.method_name"
    return [f"{class_name}.{method_name}" for class_name, method_name in fake_me.calls]


# Test cases for executors with updated docstrings
EXECUTORS_TO_TEST = [
    "D3_Q3_TraceabilityValidator",
    "D3_Q4_TechnicalFeasibilityEvaluator",
    "D5_Q2_CompositeMeasurementValidator",
    "D6_Q5_ContextualAdaptabilityEvaluator",
]


@pytest.fixture
def minimal_context() -> Dict:
    """Provide minimal context for executor tests."""
    return {
        "document_text": "Sample policy document text for testing.",
        "tables": [],
        "metadata": {"title": "Test Plan", "timestamp": "2024-01-01"},
        "product_targets": [],
        "composite_indicators": [],
        "proxy_indicators": [],
        "policy_area": "PA01",
    }


class TestExecutorDocstringCoherence:
    """Test suite for executor docstring-execution coherence."""

    def test_docstring_methods_are_subset_of_executed(self, minimal_context):
        """
        Verify that methods documented in docstrings are actually executed.

        Note: Executors may execute additional helper methods not in docstring,
        but all documented methods MUST be executed.
        """
        for executor_name in EXECUTORS_TO_TEST:
            executor_class = getattr(exec_mod, executor_name, None)
            if executor_class is None:
                pytest.skip(f"Executor {executor_name} not found")
                continue

            # Parse docstring
            docstring = executor_class.__doc__ or ""
            documented_methods = parse_docstring_methods(docstring)

            if not documented_methods:
                # Old format docstring, skip
                continue

            # Get executed methods
            executed_methods = get_executed_methods(executor_class, minimal_context)

            # Check that documented methods are in executed methods
            documented_set = set(documented_methods)
            executed_set = set(executed_methods)

            missing = documented_set - executed_set
            assert not missing, (
                f"{executor_name}: Documented methods not executed: {missing}\n"
                f"Documented: {documented_methods}\n"
                f"Executed: {executed_methods}"
            )

    def test_metadata_has_required_counts(self, minimal_context):
        """
        Verify that metadata includes counts for all list-type raw_evidence fields.
        """
        for executor_name in EXECUTORS_TO_TEST:
            executor_class = getattr(exec_mod, executor_name, None)
            if executor_class is None:
                continue

            fake_me = _FakeMethodExecutor()
            executor = executor_class(
                executor_id="TEST-Q1",
                config={},
                method_executor=fake_me
            )

            try:
                result = executor.execute(minimal_context)
            except Exception:
                # Some executors may fail, that's OK for this test
                continue

            metadata = result.get("metadata", {})

            # Check for standard count fields
            assert "methods_executed" in metadata, f"{executor_name}: Missing
methods_executed in metadata"

    def test_nomenclature_is_snake_case(self, minimal_context):
        """
        Verify that all metadata keys use snake_case nomenclature.
        """
        camel_case_pattern = re.compile(r"[a-z]+[A-Z]")

        for executor_name in EXECUTORS_TO_TEST:
            executor_class = getattr(exec_mod, executor_name, None)
            if executor_class is None:
                continue

            fake_me = _FakeMethodExecutor()
            executor = executor_class(
                executor_id="TEST-Q1",
                config={},
                method_executor=fake_me
            )

            try:
                result = executor.execute(minimal_context)
            except Exception:
                continue

            metadata = result.get("metadata", {})
            raw_evidence = result.get("raw_evidence", {})

            # Check metadata keys
            for key in metadata.keys():
                assert not camel_case_pattern.search(key), (
                    f"{executor_name}: Metadata key '{key}' uses camelCase instead of
snake_case"
                )

            # Check top-level raw_evidence keys
            for key in raw_evidence.keys():
                assert not camel_case_pattern.search(key), (
                    f"{executor_name}: raw_evidence key '{key}' uses camelCase instead of
snake_case"
                )


class TestTypeProtection:
    """Test suite for type protection in executors."""

    def test_null_safe_get_operations(self, minimal_context):
        """
        Verify that executors handle None returns gracefully.
        """
        for executor_name in EXECUTORS_TO_TEST:
            executor_class = getattr(exec_mod, executor_name, None)
            if executor_class is None:
                continue

            # Create executor with fake method executor that returns None
            class NoneReturningExecutor(_FakeMethodExecutor):
                def execute(self, class_name: str, method_name: str, **payload):
                    return None

            fake_me = NoneReturningExecutor()
            executor = executor_class(
                executor_id="TEST-Q1",
                config={},
                method_executor=fake_me
            )

            # Should not raise AttributeError even with None returns
            try:
                result = executor.execute(minimal_context)
                # If we get here, the executor handled None gracefully
                assert "executor_id" in result
            except AttributeError as e:
                pytest.fail(f"{executor_name}: AttributeError with None return: {e}")
            except Exception:
                # Other exceptions are OK - we're testing type safety, not full execution
                pass

===== FILE: tests/test_executors_injection.py =====
import importlib
import sys
import types

import pytest

# Stub orchestrator dependencies to avoid circular imports during test import
core_stub = types.ModuleType("saaaaaa.core.orchestrator.core")

class _FakeMethodExecutor:
    def __init__(self):
        self.calls = []

    def execute(self, class_name: str, method_name: str, **payload):
        self.calls.append((class_name, method_name, payload))
        return {"class": class_name, "method": method_name, "payload": payload}

core_stub.MethodExecutor = _FakeMethodExecutor
core_stub.AbortRequested = type("AbortRequested", (), {})
core_stub.AbortSignal = type("AbortSignal", (), {})
core_stub.ResourceLimits = type("ResourceLimits", (), {})
core_stub.PhaseInstrumentation = type("PhaseInstrumentation", (), {})
core_stub.PhaseResult = type("PhaseResult", (), {})
core_stub.MicroQuestionRun = type("MicroQuestionRun", (), {})
core_stub.ScoredMicroQuestion = type("ScoredMicroQuestion", (), {})
core_stub.Evidence = type("Evidence", (), {})
core_stub.PreprocessedDocument = type("PreprocessedDocument", (), {})
core_stub.Orchestrator = type("Orchestrator", (), {})
sys.modules["saaaaaa.core.orchestrator.core"] = core_stub

factory_stub = types.ModuleType("saaaaaa.core.orchestrator.factory")
def _fake_build_processor(me: _FakeMethodExecutor | None = None):
    bundle = types.SimpleNamespace()
    bundle.method_executor = me or _FakeMethodExecutor()
    return bundle
factory_stub.build_processor = _fake_build_processor
sys.modules["saaaaaa.core.orchestrator.factory"] = factory_stub

contract_loader = types.ModuleType("contract_loader")
class _Dummy:  # minimal placeholder types
    pass
contract_loader.JSONContractLoader = _Dummy
contract_loader.LoadError = _Dummy
contract_loader.LoadResult = _Dummy
sys.modules.setdefault("saaaaaa.core.orchestrator.contract_loader", contract_loader)

exec_mod = importlib.import_module("saaaaaa.core.orchestrator.executors")


def test_run_phase2_executors_uses_method_executor(monkeypatch):
    """Ensure executors are instantiated with MethodExecutor and routed calls go through
```

```
it."""

    class FakeMethodExecutor:
        def __init__(self):
            self.calls = []

        def execute(self, class_name: str, method_name: str, **payload):
            self.calls.append((class_name, method_name, payload))
            return {"class": class_name, "method": method_name, "payload": payload}

    # Patch MethodExecutor type used by BaseExecutor and factory wiring
    monkeypatch.setattr(exec_mod, "MethodExecutor", FakeMethodExecutor)

    fake_executor = FakeMethodExecutor()

    class FakeBundle:
        def __init__(self, me):
            self.method_executor = me

    # Provide a bundle that supplies the fake MethodExecutor
    monkeypatch.setattr(exec_mod, "build_processor", lambda: FakeBundle(fake_executor))
    monkeypatch.setattr(exec_mod, "load_executor_config", lambda _: {})

    # Minimal executor that exercises _execute_method once
    class DummyExecutor(exec_mod.BaseExecutor):
        def execute(self, context: dict):
            result = self._execute_method("DummyClass", "dummy_method", context,
foo="bar")
            return {
                "executor_id": self.executor_id,
                "raw_evidence": result,
                "metadata": {"methods_executed": [log["method"] for log in
self.execution_log]},
                "execution_metrics": {},
            }

    monkeypatch.setattr(exec_mod, "EXECUTOR_REGISTRY", {"D-TEST": DummyExecutor})

    context_package = {"doc": "text"}
    results = exec_mod.run_phase2_executors(context_package, ["PA01"])

    # Validate that MethodExecutor.execute was called with merged context + kwargs
    assert fake_executor.calls == [
        ("DummyClass", "dummy_method", {"doc": "text", "policy_area": "PA01", "foo":
"bar"})
    ]
    assert "PA01" in results
    assert "D-TEST" in results["PA01"]

===== FILE: tests/test_flux_contracts.py =====
"""
Contract tests for FLUX pipeline.

Tests phase compatibility, preconditions, postconditions, and determinism.
"""

# stdlib
from __future__ import annotations

import json
from typing import Any

# third-party
import polars as pl
import pyarrow as pa
import pytest
from hypothesis import given, strategies as st
from pydantic import ValidationError
```

```
# project
from saaaaaa.flux.configs import (
    AggregateConfig,
    ChunkConfig,
    IngestConfig,
    NormalizeConfig,
    ReportConfig,
    ScoreConfig,
    SignalsConfig,
)
from saaaaaa.flux.models import (
    AggregateDeliverable,
    AggregateExpectation,
    ChunkDeliverable,
    ChunkExpectation,
    DocManifest,
    IngestDeliverable,
    NormalizeDeliverable,
    NormalizeExpectation,
    ReportDeliverable,
    ReportExpectation,
    ScoreDeliverable,
    ScoreExpectation,
    SignalsDeliverable,
    SignalsExpectation,
)
from saaaaaa.flux.phases import (
    CompatibilityError,
    PostconditionError,
    PreconditionError,
    assert_compat,
    run_aggregate,
    run_chunk,
    run_ingest,
    run_normalize,
    run_report,
    run_score,
    run_signals,
)


class TestCompatibilityContracts:
    """Test phase compatibility contracts."""

    def test_ingest_to_normalize_compatibility(self) -> None:
        """IngestDeliverable is compatible with NormalizeExpectation."""
        manifest = DocManifest(document_id="test-doc", source_uri="test://uri")
        ingest_del = IngestDeliverable(
            manifest=manifest,
            raw_text="test content",
            tables=[],
            provenance_ok=True,
        )

        # Should not raise
        assert_compat(ingest_del, NormalizeExpectation)

    def test_normalize_to_chunk_compatibility(self) -> None:
        """NormalizeDeliverable is compatible with ChunkExpectation."""
        norm_del = NormalizeDeliverable(
            sentences=["sentence 1", "sentence 2"],
            sentence_meta=[{"idx": 0}, {"idx": 1}],
        )

        # Should not raise
        assert_compat(norm_del, ChunkExpectation)
    def test_chunk_to_signals_compatibility(self) -> None:
        """ChunkDeliverable is compatible with SignalsExpectation."""
        chunk_del = ChunkDeliverable(
            chunks=[{"id": "c0", "text": "chunk"}],
            chunk_index={"micro": [], "meso": ["c0"], "macro": []},
        )

        # Should not raise
        assert_compat(chunk_del, SignalsExpectation)

    def test_signals_to_aggregate_compatibility(self) -> None:
        """SignalsDeliverable is compatible with AggregateExpectation."""
        sig_del = SignalsDeliverable(
            enriched_chunks=[{"id": "c0", "patterns_used": 5}],
            used_signals={"present": True},
        )

        # Should not raise
        assert_compat(sig_del, AggregateExpectation)

    def test_aggregate_to_score_compatibility(self) -> None:
        """AggregateDeliverable is compatible with ScoreExpectation."""
        tbl = pa.table({"item_id": ["c0"], "patterns_used": [5]})
        agg_del = AggregateDeliverable(
            features=tbl,
            aggregation_meta={"rows": 1},
        )

        # Should not raise
        assert_compat(agg_del, ScoreExpectation)

    def test_score_to_report_compatibility(self) -> None:
        """ScoreDeliverable is compatible with ReportExpectation."""
        df = pl.DataFrame(
            {"item_id": ["c0"], "metric": ["precision"], "value": [0.95]}
        )
        score_del = ScoreDeliverable(scores=df, calibration={})

        # Should not raise
        assert_compat(score_del, ReportExpectation)

    def test_incompatible_raises_error(self) -> None:
        """Incompatible deliverable raises CompatibilityError."""
        # Create a deliverable that's missing required fields for the next expectation
        # For example, IngestDeliverable without manifest field won't match
NormalizeExpectation
        # Since Pydantic validates at construction, we need to create an invalid structure

        # Using a different type that won't match
        chunk_del = ChunkDeliverable(
            chunks=[{"id": "c0"}],
            chunk_index={"micro": [], "meso": [], "macro": []}
        )

        # ChunkDeliverable has different fields than NormalizeExpectation
        with pytest.raises(CompatibilityError):
            assert_compat(chunk_del, NormalizeExpectation)


class TestPreconditions:
    """Test phase preconditions."""

    def test_ingest_requires_nonempty_uri(self) -> None:
        """run_ingest requires non-empty input_uri."""
        cfg = IngestConfig()

        with pytest.raises(PreconditionError, match="non-empty input_uri"):
            run_ingest(cfg, input_uri="")
        with pytest.raises(PreconditionError, match="non-empty input_uri"):
            run_ingest(cfg, input_uri="   ")

    def test_signals_requires_registry_get(self) -> None:
        """run_signals requires registry_get callable."""
        cfg = SignalsConfig()
        chunk_del = ChunkDeliverable(
            chunks=[{"id": "c0"}], chunk_index={"micro": [], "meso": [], "macro": []}
        )

        with pytest.raises(PreconditionError, match="registry_get not None"):
            run_signals(cfg, chunk_del, registry_get=None)  # type: ignore[arg-type]

    def test_aggregate_requires_nonempty_group_by(self) -> None:
        """run_aggregate requires group_by not empty."""
        cfg = AggregateConfig(group_by=[])
        sig_del = SignalsDeliverable(
            enriched_chunks=[{"id": "c0"}], used_signals={}
        )

        with pytest.raises(PreconditionError, match="group_by not empty"):
            run_aggregate(cfg, sig_del)

    def test_score_requires_nonempty_metrics(self) -> None:
        """run_score requires metrics not empty."""
        cfg = ScoreConfig(metrics=[])
        tbl = pa.table({"item_id": ["c0"]})
        agg_del = AggregateDeliverable(features=tbl, aggregation_meta={})

        with pytest.raises(PreconditionError, match="metrics not empty"):
            run_score(cfg, agg_del)


class TestPostconditions:
    """Test phase postconditions."""

    def test_normalize_postcondition_nonempty_sentences(self) -> None:
        """run_normalize ensures non-empty sentences."""
        cfg = NormalizeConfig()
        manifest = DocManifest(document_id="test")
        ing_del = IngestDeliverable(
            manifest=manifest, raw_text="", tables=[], provenance_ok=True
        )

        with pytest.raises(PostconditionError, match="non-empty sentences"):
            run_normalize(cfg, ing_del)

    def test_chunk_postcondition_valid_index_keys(self) -> None:
        """run_chunk ensures chunk_index has valid keys."""
        # This is tested implicitly in run_chunk
        # Postcondition checks for micro/meso/macro keys
        pass


class TestDeterminism:
    """Test deterministic execution."""

    def test_ingest_deterministic_fingerprint(self) -> None:
        """run_ingest produces same fingerprint for same input."""
        import os
        import tempfile
        from pathlib import Path

        os.environ["HF_ONLINE"] = "1"

        cfg = IngestConfig()
        with tempfile.NamedTemporaryFile(mode="w", delete=False, suffix=".txt") as f:
            f.write("test content")
            uri = f.name

        try:
            outcome1 = run_ingest(cfg, input_uri=uri)
            outcome2 = run_ingest(cfg, input_uri=uri)
            assert outcome1.fingerprint == outcome2.fingerprint
        finally:
            Path(uri).unlink()
            del os.environ["HF_ONLINE"]

    def test_normalize_deterministic_fingerprint(self) -> None:
        """run_normalize produces same fingerprint for same input."""
        cfg = NormalizeConfig()
        manifest = DocManifest(document_id="test")
        ing_del = IngestDeliverable(
            manifest=manifest,
            raw_text="Line 1\nLine 2\nLine 3",
            tables=[],
            provenance_ok=True,
        )

        outcome1 = run_normalize(cfg, ing_del)
        outcome2 = run_normalize(cfg, ing_del)

        assert outcome1.fingerprint == outcome2.fingerprint

    def test_full_pipeline_deterministic(self) -> None:
        """Full pipeline produces same fingerprints across runs."""
        import os
        import tempfile
        from pathlib import Path

        os.environ["HF_ONLINE"] = "1"

        with tempfile.NamedTemporaryFile(mode="w", delete=False, suffix=".txt") as f:
            f.write("test content")
            input_uri = f.name

        try:
            # Run 1
            ing_cfg = IngestConfig()
            ing_out1 = run_ingest(ing_cfg, input_uri=input_uri)
            ing_del1 = IngestDeliverable.model_validate(ing_out1.payload)

            norm_cfg = NormalizeConfig()
            norm_out1 = run_normalize(norm_cfg, ing_del1)

            # Run 2
            ing_out2 = run_ingest(ing_cfg, input_uri=input_uri)
            ing_del2 = IngestDeliverable.model_validate(ing_out2.payload)

            norm_out2 = run_normalize(norm_cfg, ing_del2)

            # Fingerprints must match
            assert ing_out1.fingerprint == ing_out2.fingerprint
            assert norm_out1.fingerprint == norm_out2.fingerprint
        finally:
            Path(input_uri).unlink()
            del os.environ["HF_ONLINE"]


class TestConfigValidation:
    """Test configuration validation."""

    def test_ingest_config_frozen(self) -> None:
        """IngestConfig is frozen."""
        cfg = IngestConfig()

        with pytest.raises(ValidationError):
            cfg.enable_ocr = False  # type: ignore[misc]

    def test_normalize_config_from_env(self) -> None:
        """NormalizeConfig can be created from environment."""
        import os

        os.environ["FLUX_NORMALIZE_UNICODE_FORM"] = "NFKC"
        os.environ["FLUX_NORMALIZE_KEEP_DIACRITICS"] = "false"

        cfg = NormalizeConfig.from_env()
        assert cfg.unicode_form == "NFKC"
        assert cfg.keep_diacritics is False

        # Cleanup
        del os.environ["FLUX_NORMALIZE_UNICODE_FORM"]
        del os.environ["FLUX_NORMALIZE_KEEP_DIACRITICS"]

    def test_all_configs_have_from_env(self) -> None:
        """All config classes have from_env method."""
        configs = [
            IngestConfig,
            NormalizeConfig,
            ChunkConfig,
            SignalsConfig,
            AggregateConfig,
            ScoreConfig,
            ReportConfig,
        ]

        for cfg_cls in configs:
            assert hasattr(cfg_cls, "from_env")
            cfg = cfg_cls.from_env()
            assert isinstance(cfg, cfg_cls)


@pytest.mark.property
class TestPropertyBasedContracts:
    """Property-based tests with Hypothesis."""

    @given(st.text(min_size=1).filter(lambda s: s.strip()))
    def test_ingest_always_produces_fingerprint(self, content: str) -> None:
        """run_ingest always produces a 64-char fingerprint."""
        import tempfile
        from pathlib import Path

        cfg = IngestConfig()

        with tempfile.NamedTemporaryFile(mode="w", delete=False, suffix=".txt") as f:
            f.write(content)
            uri = f.name

        try:
            outcome = run_ingest(cfg, input_uri=uri)
            assert len(outcome.fingerprint) == 64
            assert outcome.fingerprint.isalnum()
        finally:
            Path(uri).unlink()

    @given(st.lists(st.text(min_size=1), min_size=1, max_size=100))
    def test_normalize_preserves_sentence_count(self, sentences: list[str]) -> None:
        """run_normalize preserves input sentence structure."""
        cfg = NormalizeConfig()
        manifest = DocManifest(document_id="test")
        raw_text = "\n".join(sentences)
        ing_del = IngestDeliverable(
            manifest=manifest,
            raw_text=raw_text,
            tables=[],
            provenance_ok=True,
        )

        outcome = run_normalize(cfg, ing_del)
        norm_del = NormalizeDeliverable.model_validate(outcome.payload)

        # Should have same number of non-empty sentences
        expected_count = len([s for s in raw_text.split('\n') if s.strip()])
        assert len(norm_del.sentences) == expected_count
        assert len(norm_del.sentence_meta) == expected_count

    @given(
        st.lists(
            st.dictionaries(
                st.text(min_size=1, max_size=10), st.integers() | st.text()
            ),
            min_size=1,
            max_size=50,
        )
    )
    def test_signals_preserves_chunk_count(
        self, chunks: list[dict[str, Any]]
    ) -> None:
        """run_signals preserves chunk count."""
        cfg = SignalsConfig()
        chunk_del = ChunkDeliverable(
            chunks=chunks,
            chunk_index={"micro": [], "meso": [], "macro": []},
        )

        def dummy_registry(policy_area: str) -> dict[str, Any] | None:
            return {"patterns": ["p1"]}

        outcome = run_signals(cfg, chunk_del, registry_get=dummy_registry)
        sig_del = SignalsDeliverable.model_validate(outcome.payload)

        assert len(sig_del.enriched_chunks) == len(chunks)

    @given(st.lists(st.text(min_size=1, max_size=20), min_size=1, max_size=10))
    def test_chunk_index_contains_chunk_ids(self, texts: list[str]) -> None:
        """run_chunk produces chunk_index containing all chunk IDs."""
        cfg = ChunkConfig()
        norm_del = NormalizeDeliverable(
            sentences=texts,
            sentence_meta=[{} for _ in texts],
        )

        outcome = run_chunk(cfg, norm_del)
        chunk_del = ChunkDeliverable.model_validate(outcome.payload)

        # All chunk IDs should be in the index
        all_index_ids = (
```