```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 24
================================================================================
Generated: 2025-12-07T06:17:30.363789
Files in this batch: 17
================================================================================


================================================================================
FILE: tests/calibration_system/__init__.py
================================================================================

"""
Calibration System Validation Suite

Comprehensive all-or-nothing quality gate for the calibration system.
Tests verify inventory consistency, layer correctness, intrinsic coverage,
AST extraction accuracy, orchestrator runtime, hardcoded values, and performance.

ALL tests must pass for the system to be considered production-ready.
"""



================================================================================
FILE: tests/calibration_system/generate_failure_report.py
================================================================================

#!/usr/bin/env python3
"""
Generate Calibration System Failure Report

Aggregates test results and generates a comprehensive failure report
when any validation test fails.
"""
import argparse
import json
from datetime import datetime
from pathlib import Path
from typing import Dict, List


def parse_args() -> argparse.Namespace:
    """Parse command line arguments"""
    parser = argparse.ArgumentParser(
        description="Generate calibration system failure report"
    )
    parser.add_argument("--test1", required=True, help="Test 1 outcome")
    parser.add_argument("--test2", required=True, help="Test 2 outcome")
    parser.add_argument("--test3", required=True, help="Test 3 outcome")
    parser.add_argument("--test4", required=True, help="Test 4 outcome")
    parser.add_argument("--test5", required=True, help="Test 5 outcome")
    parser.add_argument("--test6", required=True, help="Test 6 outcome")
    parser.add_argument("--test7", required=True, help="Test 7 outcome")
    parser.add_argument("--output", default="calibration_system_failure_report.md",
                        help="Output file path")
    return parser.parse_args()


def load_test_results() -> Dict[str, List[str]]:
    """Load detailed test results from XML files"""
    results_dir = Path("test-results")
    failures = {}

    if not results_dir.exists():
        return failures

    for xml_file in results_dir.glob("*.xml"):
        try:
            import xml.etree.ElementTree as ET
            tree = ET.parse(xml_file)
            root = tree.getroot()

            test_name = xml_file.stem
            test_failures = []

            for testcase in root.findall(".//testcase"):
                failure = testcase.find("failure")
                if failure is not None:
                    test_failures.append({
                        "name": testcase.get("name"),
                        "message": failure.get("message", ""),
                        "details": failure.text or ""
                    })

            if test_failures:
                failures[test_name] = test_failures

        except Exception as e:
            print(f"Warning: Could not parse {xml_file}: {e}")

    return failures


def generate_report(args: argparse.Namespace) -> str:
    """Generate the failure report"""
    tests = {
        "Test 1: Inventory Consistency": args.test1,
        "Test 2: Layer Correctness": args.test2,
        "Test 3: Intrinsic Coverage (â\211¥80%)": args.test3,
        "Test 4: AST Extraction Accuracy": args.test4,
        "Test 5: Orchestrator Runtime": args.test5,
        "Test 6: No Hardcoded Calibrations": args.test6,
        "Test 7: Performance Benchmarks": args.test7,
    }

    failures = load_test_results()
    failed_tests = [name for name, outcome in tests.items() if outcome != "success"]

    report = []
    report.append("# CALIBRATION SYSTEM FAILURE REPORT")
    report.append("")
    report.append(f"**Generated:** {datetime.now().isoformat()}")
    report.append("")

    if not failed_tests:
        report.append("## â\234\205 STATUS: PRODUCTION READY")
        report.append("")
        report.append("All calibration validation tests passed successfully.")
        report.append("")
        report.append("### Test Results")
        report.append("")
        for test_name, outcome in tests.items():
            report.append(f"- â\234\205 {test_name}: **{outcome}**")

    else:
        report.append("## â\235\214 STATUS: NOT READY FOR PRODUCTION")
        report.append("")
        report.append(f"**{len(failed_tests)} of {len(tests)} validation tests FAILED**")
        report.append("")
        report.append("### Critical Failures")
        report.append("")

        for test_name, outcome in tests.items():
            icon = "â\234\205" if outcome == "success" else "â\235\214"
            report.append(f"{icon} **{test_name}**: {outcome}")

        report.append("")
        report.append("---")
        report.append("")
        report.append("## Failure Analysis")
        report.append("")

        test_descriptions = {
            "Test 1": {
                "name": "Inventory Consistency",
                "purpose": "Verifies all methods are consistently defined across JSON files",
                "impact": "Inconsistent inventories can cause runtime errors and calibration mismatches",
                "remediation": [
                    "Ensure all methods in executors_methods.json exist in intrinsic_calibration.json",
                    "Remove orphaned calibration entries",
                    "Verify all 30 executors are properly defined"
                ]
            },
            "Test 2": {
                "name": "Layer Correctness",
                "purpose": "Validates architectural integrity with 8 required layers",
                "impact": "Missing layers break the execution pipeline",
                "remediation": [
                    "Each executor must have methods in all 8 layers: ingestion, extraction, transformation, validation, aggregation, scoring, reporting, meta",
                    "Verify LAYER_REQUIREMENTS mapping is complete",
                    "Check layer dependencies are acyclic"
                ]
            },
            "Test 3": {
                "name": "Intrinsic Coverage",
                "purpose": "Ensures â\211¥80% methods have computed calibrations",
                "impact": "Low coverage means system relies on defaults or fails",
                "remediation": [
                    "Run intrinsic calibration triage on uncalibrated methods",
                    "Ensure all 30 executors have at least one method with status='computed'",
                    "Document reasons for any 'excluded' or 'manual' methods"
                ]
            },
            "Test 4": {
                "name": "AST Extraction Accuracy",
                "purpose": "Validates extracted signatures match actual source code",
                "impact": "Signature mismatches cause method execution failures",
                "remediation": [
                    "Re-extract method signatures from source code",
                    "Update method inventory with correct signatures",
                    "Keep mismatch rate below 5%"
                ]
            },
            "Test 5": {
                "name": "Orchestrator Runtime",
                "purpose": "Tests correct layer evaluation and aggregation",
                "impact": "Runtime errors prevent policy analysis",
                "remediation": [
                    "Fix layer ordering issues",
                    "Ensure context propagation works correctly",
                    "Test calibration resolution without errors"
                ]
            },
            "Test 6": {
                "name": "No Hardcoded Calibrations",
                "purpose": "Scans for magic numbers in calibration-sensitive code",
                "impact": "Hardcoded values prevent systematic calibration",
                "remediation": [
                    "Move all thresholds, weights, scores to configuration",
                    "Use calibration_registry.get_calibration() instead of literals",
                    "Update intrinsic_calibration.json with externalized values"
                ]
            },
            "Test 7": {
                "name": "Performance Benchmarks",
                "purpose": "Validates load times and calibration speed",
                "impact": "Slow performance degrades user experience",
                "remediation": [
                    "Optimize JSON file size and structure",
                    "Ensure load intrinsic.json < 1s",
                    "Ensure calibrate 30 executors < 5s",
                    "Ensure calibrate 200 methods < 30s"
                ]
            }
        }

        for test_name in failed_tests:
            test_num = test_name.split(":")[0].replace("Test ", "Test")
            test_info = test_descriptions.get(test_num, {})

            report.append(f"### {test_name}")
            report.append("")

            if test_info:
                report.append(f"**Purpose:** {test_info.get('purpose', 'N/A')}")
                report.append("")
                report.append(f"**Impact:** {test_info.get('impact', 'N/A')}")
                report.append("")
                report.append("**Remediation Steps:**")
                report.append("")
                for step in test_info.get('remediation', []):
                    report.append(f"1. {step}")
                report.append("")

            test_key = f"test{test_num.replace('Test', '')}-" + test_info.get('name', '').lower().replace(' ', '-')
            if test_key in failures:
                report.append("**Specific Failures:**")
                report.append("")
                for failure in failures[test_key][:5]:
                    report.append(f"- `{failure['name']}`")
                    if failure['message']:
                        report.append(f"  - {failure['message'][:200]}")
                report.append("")

        report.append("---")
        report.append("")
        report.append("## Required Actions")
        report.append("")
        report.append("1. **DO NOT MERGE** this PR until all tests pass")
        report.append("2. Review failure details above")
        report.append("3. Apply remediation steps for each failed test")
        report.append("4. Re-run validation suite: `pytest tests/calibration_system/ -v`")
        report.append("5. Verify all tests pass locally before pushing")
        report.append("")
        report.append("## Support")
        report.append("")
        report.append("For assistance:")
        report.append("- Review test source code in `tests/calibration_system/`")
        report.append("- Check calibration documentation in `AGENTS.md`")
        report.append("- Examine test output artifacts for detailed error messages")

    return "\n".join(report)


def main():
    """Main entry point"""
    args = parse_args()
    report = generate_report(args)

    output_path = Path(args.output)
    output_path.write_text(report)

    print(f"â\234\223 Generated failure report: {output_path}")


if __name__ == "__main__":
    main()

```

```


================================================================================
FILE: tests/calibration_system/manual_verification.py
================================================================================

#!/usr/bin/env python3
"""
Manual Verification Script for Calibration System Tests

This script manually tests the calibration validation system by:
1. Running basic checks without pytest
2. Simulating failures to verify detection
3. Generating a report
"""
import json
import sys
from pathlib import Path


def check_executors_methods():
    """Check executors_methods.json structure"""
    print("=" * 60)
    print("CHECK 1: Executors Methods Structure")
    print("=" * 60)

    path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
    if not path.exists():
        print(f"â\235\214 FAIL: {path} not found")
        return False

    with open(path) as f:
        data = json.load(f)

    if not isinstance(data, list):
        print(f"â\235\214 FAIL: Expected list, got {type(data)}")
        return False

    print(f"â\234\205 Found {len(data)} executors")

    if len(data) != 30:
        print(f"â\235\214 FAIL: Expected 30 executors, found {len(data)}")
        return False

    print(f"â\234\205 Exactly 30 executors present")

    total_methods = sum(len(e.get("methods", [])) for e in data)
    print(f"â\234\205 Total methods across all executors: {total_methods}")

    return True


def check_intrinsic_calibration():
    """Check intrinsic_calibration.json structure"""
    print("\n" + "=" * 60)
    print("CHECK 2: Intrinsic Calibration Structure")
    print("=" * 60)

    path = Path("system/config/calibration/intrinsic_calibration.json")
    if not path.exists():
        print(f"â\235\214 FAIL: {path} not found")
        return False

    with open(path) as f:
        data = json.load(f)

    if not isinstance(data, dict):
        print(f"â\235\214 FAIL: Expected dict, got {type(data)}")
        return False

    methods = [k for k in data.keys() if k != "_metadata"]
    print(f"â\234\205 Found {len(methods)} method calibrations")

    if len(methods) == 0:
        print("â\232 ï¸\217  WARNING: No calibrations found (empty file)")
        return False

    computed = sum(1 for k, v in data.items()
                   if k != "_metadata" and v.get("status") == "computed")

    if len(methods) > 0:
        coverage = computed / len(methods) * 100
        print(f"Coverage: {computed}/{len(methods)} = {coverage:.1f}% with status='computed'")

        if coverage >= 80:
            print(f"â\234\205 Coverage â\211¥ 80%")
            return True
        else:
            print(f"â\235\214 FAIL: Coverage < 80%")
            return False

    return True


def check_inventory_consistency():
    """Check consistency between executor methods and calibrations"""
    print("\n" + "=" * 60)
    print("CHECK 3: Inventory Consistency")
    print("=" * 60)

    executors_path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
    intrinsic_path = Path("system/config/calibration/intrinsic_calibration.json")

    with open(executors_path) as f:
        executors = json.load(f)

    with open(intrinsic_path) as f:
        calibrations = json.load(f)

    all_methods = set()
    for executor in executors:
        for method in executor.get("methods", []):
            method_id = f"{method['class']}.{method['method']}"
            all_methods.add(method_id)

    calibrated_methods = set(calibrations.keys()) - {"_metadata"}

    missing = all_methods - calibrated_methods
    extra = calibrated_methods - all_methods

    print(f"Methods in executors: {len(all_methods)}")
    print(f"Methods in calibrations: {len(calibrated_methods)}")

    if missing:
        print(f"â\235\214 FAIL: {len(missing)} methods missing calibration")
        print(f"Sample missing: {list(missing)[:5]}")
        return False
    else:
        print(f"â\234\205 All executor methods have calibration entries")

    if extra:
        print(f"â\232 ï¸\217  WARNING: {len(extra)} extra calibrations")
        print(f"Sample extra: {list(extra)[:5]}")

    return True


def check_layer_coverage():
    """Check that all executors have all 8 layers"""
    print("\n" + "=" * 60)
    print("CHECK 4: Layer Coverage")
    print("=" * 60)

    required_layers = {
        "ingestion", "extraction", "transformation", "validation",
        "aggregation", "scoring", "reporting", "meta"
    }

    path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
    with open(path) as f:
        executors = json.load(f)

    failures = []

    for executor in executors:
        layers = set()
        for method in executor.get("methods", []):
            if "layer" in method:
                layers.add(method["layer"])

        missing = required_layers - layers
        if missing:
            failures.append((executor["executor_id"], missing))

    if failures:
        print(f"â\235\214 FAIL: {len(failures)} executors missing layers")
        for exec_id, missing in failures[:5]:
            print(f"  {exec_id}: missing {sorted(missing)}")
            return False
    else:
        print(f"â\234\205 All {len(executors)} executors have all 8 layers")
        return True


def simulate_failure_detection():
    """Simulate a failure by temporarily modifying data"""
    print("\n" + "=" * 60)
    print("SIMULATION: Failure Detection")
    print("=" * 60)
    print("This would test if the system detects intentional failures.")
    print("Skipping actual modification to preserve data integrity.")
    print("â\234\205 Simulation complete (dry run)")
    return True


def main():
    """Run all checks"""
    print("\n" + "=" * 60)
    print("CALIBRATION SYSTEM MANUAL VERIFICATION")
    print("=" * 60)

    results = {
        "executors_methods": check_executors_methods(),
        "intrinsic_calibration": check_intrinsic_calibration(),
        "inventory_consistency": check_inventory_consistency(),
        "layer_coverage": check_layer_coverage(),
        "failure_simulation": simulate_failure_detection(),
    }

    print("\n" + "=" * 60)
    print("SUMMARY")
    print("=" * 60)

    for check, passed in results.items():
        status = "â\234\205 PASS" if passed else "â\235\214 FAIL"
        print(f"{status}: {check}")

    all_passed = all(results.values())

    print("\n" + "=" * 60)
    if all_passed:
        print("â\234\205 ALL CHECKS PASSED - System appears ready")
    else:
        print("â\235\214 SOME CHECKS FAILED - System NOT ready")
    print("=" * 60)

    return 0 if all_passed else 1


if __name__ == "__main__":
    sys.exit(main())



```

```
================================================================================
FILE: tests/calibration_system/test_ast_extraction_accuracy.py
================================================================================

"""
Test 4: AST Extraction Accuracy - Comparing Extracted Signatures vs Actual Code

Validates that method signatures in the calibration system match actual source code:
- Parse Python files to extract actual method signatures
- Compare with signatures stored in calibration metadata
- Detect mismatches in parameter names, types, defaults

FAILURE CONDITION: Signature mismatch > 5% = SYSTEM NOT READY
"""
import ast
import json
import pytest
from pathlib import Path
from typing import Dict, List, Set, Any, Optional, Tuple
import inspect


class MethodSignatureExtractor(ast.NodeVisitor):
    """AST visitor to extract method signatures"""

    def __init__(self):
        self.signatures: Dict[str, Dict[str, Any]] = {}
        self.current_class: Optional[str] = None

    def visit_ClassDef(self, node: ast.ClassDef) -> None:
        """Visit class definition"""
        old_class = self.current_class
        self.current_class = node.name
        self.generic_visit(node)
        self.current_class = old_class

    def visit_FunctionDef(self, node: ast.FunctionDef) -> None:
        """Visit function/method definition"""
        if self.current_class:
            method_id = f"{self.current_class}.{node.name}"
        else:
            method_id = node.name

        args = []
        kwargs = []
        has_self = False

        for i, arg in enumerate(node.args.args):
            arg_name = arg.arg
            if i == 0 and arg_name in ("self", "cls"):
                has_self = True
                continue

            if i < len(node.args.args) - len(node.args.defaults):
                args.append(arg_name)
            else:
                kwargs.append(arg_name)

        returns = ast.unparse(node.returns) if node.returns else "Any"
        is_async = isinstance(node, ast.AsyncFunctionDef)

        self.signatures[method_id] = {
            "args": args,
            "kwargs": kwargs,
            "returns": returns,
            "is_async": is_async,
            "has_self": has_self,
            "lineno": node.lineno,
        }

    visit_AsyncFunctionDef = visit_FunctionDef


class TestASTExtractionAccuracy:

    MAX_MISMATCH_PERCENT = 5.0

    @pytest.fixture(scope="class")
    def executors_methods(self) -> Dict[str, Any]:
        """Load executors_methods.json"""
        path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
        with open(path) as f:
            return json.load(f)

    @pytest.fixture(scope="class")
    def source_signatures(self) -> Dict[str, Dict[str, Any]]:
        """Extract signatures from actual source files"""
        signatures = {}

        src_dirs = [
            Path("src/farfan_pipeline"),
            Path("farfan_core/farfan_core"),
        ]

        for src_dir in src_dirs:
            if not src_dir.exists():
                continue

            for py_file in src_dir.rglob("*.py"):
                if "test" in str(py_file) or "__pycache__" in str(py_file):
                    continue

                try:
                    tree = ast.parse(py_file.read_text(), filename=str(py_file))
                    extractor = MethodSignatureExtractor()
                    extractor.visit(tree)

                    for method_id, sig in extractor.signatures.items():
                        sig["file"] = str(py_file)
                        signatures[method_id] = sig

                except (SyntaxError, UnicodeDecodeError) as e:
                    print(f"Warning: Could not parse {py_file}: {e}")

        return signatures

    def test_source_signatures_extracted(self, source_signatures):
        """Verify we extracted signatures from source"""
        assert len(source_signatures) > 0, \
            "No method signatures extracted from source code"

        print(f"\nExtracted {len(source_signatures)} method signatures from source")

    def test_executor_methods_exist_in_source(
        self, executors_methods, source_signatures
    ):
        """Verify executor methods exist in source code"""
        missing_methods = []

        for executor in executors_methods:
            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"

                if method_id not in source_signatures:
                    missing_methods.append({
                        "executor": executor["executor_id"],
                        "method": method_id
                    })

        if missing_methods:
            msg = f"\nWARNING: {len(missing_methods)} methods not found in source:\n"
            for item in missing_methods[:10]:
                msg += f"  {item['executor']}: {item['method']}\n"
            print(msg)

    def test_signature_mismatch_rate(
        self, executors_methods, source_signatures
    ):
        """CRITICAL: Verify signature mismatch rate is below threshold"""
        total_checked = 0
        mismatches = []

        for executor in executors_methods:
            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"

                if method_id not in source_signatures:
                    continue

                total_checked += 1
                source_sig = source_signatures[method_id]

                if "signature" in method:
                    stored_sig = method["signature"]

                    if not self._signatures_match(stored_sig, source_sig):
                        mismatches.append({
                            "executor": executor["executor_id"],
                            "method": method_id,
                            "stored": stored_sig,
                            "actual": source_sig
                        })

        mismatch_rate = (len(mismatches) / total_checked * 100) if total_checked > 0 else 0

        if mismatches:
            msg = f"\nSignature mismatches ({len(mismatches)}/{total_checked} = {mismatch_rate:.1f}%):\n"
            for item in mismatches[:5]:
                msg += f"  {item['method']}:\n"
                msg += f"    Stored: {item['stored']}\n"
                msg += f"    Actual: {item['actual']}\n"
            print(msg)

        assert mismatch_rate <= self.MAX_MISMATCH_PERCENT, \
            f"Signature mismatch rate {mismatch_rate:.1f}% exceeds threshold {self.MAX_MISMATCH_PERCENT}%"

    def _signatures_match(
        self, stored: Dict[str, Any], actual: Dict[str, Any]
    ) -> bool:
        """Compare two signatures for equality"""
        if not isinstance(stored, dict) or not isinstance(actual, dict):
            return False

        stored_args = stored.get("args", [])
        actual_args = actual.get("args", [])

        if len(stored_args) != len(actual_args):
            return False

        for s_arg, a_arg in zip(stored_args, actual_args):
            if isinstance(s_arg, str) and isinstance(a_arg, str):
                if s_arg != a_arg:
                    return False

        return True

    def test_private_methods_marked(self, executors_methods):
        """Verify private methods (starting with _) are marked"""
        private_methods = []

        for executor in executors_methods:
            for method in executor["methods"]:
                if method["method"].startswith("_"):
                    private_methods.append({
                        "executor": executor["executor_id"],
                        "method": f"{method['class']}.{method['method']}"
                    })

        if private_methods:
            print(f"\nFound {len(private_methods)} private methods in executors")

    def test_async_methods_identified(self, source_signatures):
        """Identify async methods in source code"""
        async_methods = [
            method_id for method_id, sig in source_signatures.items()
            if sig.get("is_async")
        ]

        if async_methods:
            print(f"\nFound {len(async_methods)} async methods: {async_methods[:5]}")

    def test_method_parameter_count_reasonable(self, source_signatures):
        """Verify methods don't have excessive parameters"""
        MAX_PARAMS = 10

        excessive_params = []

        for method_id, sig in source_signatures.items():
            total_params = len(sig.get("args", [])) + len(sig.get("kwargs", []))

            if total_params > MAX_PARAMS:
                excessive_params.append({
                    "method": method_id,
                    "param_count": total_params,
                    "file": sig.get("file")
                })

        if excessive_params:
            msg = f"\nMethods with >{MAX_PARAMS} parameters:\n"
            for item in excessive_params[:5]:
                msg += f"  {item['method']}: {item['param_count']} params\n"
            print(msg)

    def test_return_type_annotations_present(self, source_signatures):
        """Check for return type annotations"""
        missing_returns = []

        for method_id, sig in source_signatures.items():
            if method_id.startswith("_"):
                continue

            if sig.get("returns") == "Any" or not sig.get("returns"):
                missing_returns.append(method_id)

        if missing_returns:
            coverage = (1 - len(missing_returns) / len(source_signatures)) * 100
            print(
                f"\nReturn type annotation coverage: {coverage:.1f}% "
                f"({len(source_signatures) - len(missing_returns)}/{len(source_signatures)})"
            )



================================================================================
FILE: tests/calibration_system/test_canonical_id_format.py
================================================================================

"""
test_canonical_id_format.py - Canonical ID Format Enforcement

```

```
This test suite enforces Constraint 1: canonical_method_catalogue_v2.json is the
ONLY source of truth for method IDs, and all IDs must follow canonical format.

Tests:
- Canonical ID format validation (regex pattern)
- Cross-file consistency (all JSONs reference canonical IDs only)
- No duplicate IDs
- No invalid characters (spaces, special chars)
"""

import json
import re
from pathlib import Path
from typing import Dict, List, Set

import pytest


# Paths relative to repo root
REPO_ROOT = Path(__file__).parent.parent.parent
CATALOG_PATH = REPO_ROOT / "config" / "canonical_method_catalogue_v2.json"
PARAMETRIZED_PATH = REPO_ROOT / "config" / "canonic_inventory_methods_parametrized.json"
LAYERS_PATH = REPO_ROOT / "config" / "canonic_inventorry_methods_layers.json"
INTRINSIC_PATH = REPO_ROOT / "system" / "config" / "calibration" / "intrinsic_calibration.json"

# Canonical ID format: module.Class.method (with optional .linenumber)
# Must NOT contain: spaces, special chars except dot and underscore
CANONICAL_PATTERN = re.compile(r'^[a-zA-Z0-9_]+(\.[a-zA-Z0-9_]+)+$')

# Invalid patterns that should trigger failure
INVALID_PATTERNS = [
    re.compile(r'\s'),  # Spaces
    re.compile(r'::'),  # Double colon separator
    re.compile(r'/'),   # Forward slash
    re.compile(r'Copia de'),  # Spanish "Copy of" indicates backup file
]


def load_json(path: Path) -> Dict:
    """Load JSON file."""
    with open(path, 'r', encoding='utf-8') as f:
        return json.load(f)


@pytest.fixture(scope="module")
def canonical_ids() -> Set[str]:
    """Load canonical IDs from source of truth."""
    catalog = load_json(CATALOG_PATH)
    assert isinstance(catalog, list), "Catalog must be a list"

    ids = set()
    for entry in catalog:
        if 'unique_id' in entry:
            ids.add(entry['unique_id'])

    assert len(ids) > 0, "Catalog must contain at least one method"
    return ids


def extract_method_ids(file_path: Path, file_type: str) -> List[str]:
    """Extract method IDs from a JSON file."""
    data = load_json(file_path)

    if file_type == 'catalog':
        return [entry['unique_id'] for entry in data if 'unique_id' in entry]

    # Dict-based files (layers, intrinsic, parametrized)
    if isinstance(data, dict):
        # Skip metadata keys starting with $
        return [k for k in data.keys() if not k.startswith('$')]

    return []


class TestCanonicalIDFormat:
    """Test suite for canonical ID format validation."""

    def test_catalog_ids_match_canonical_pattern(self, canonical_ids):
        """[Constraint 1] All catalog IDs must match canonical pattern."""
        invalid_ids = []

        for id_ in canonical_ids:
            if not CANONICAL_PATTERN.match(id_):
                invalid_ids.append(id_)

        assert len(invalid_ids) == 0, (
            f"Found {len(invalid_ids)} IDs not matching canonical pattern:\n" +
            "\n".join(f"  - {id_}" for id_ in invalid_ids[:10])
        )

    def test_catalog_ids_no_invalid_patterns(self, canonical_ids):
        """[Constraint 1] Catalog IDs must not contain invalid patterns."""
        violations = {}

        for id_ in canonical_ids:
            for pattern in INVALID_PATTERNS:
                if pattern.search(id_):
                    if pattern.pattern not in violations:
                        violations[pattern.pattern] = []
                    violations[pattern.pattern].append(id_)

        assert len(violations) == 0, (
            "Found invalid patterns in catalog IDs:\n" +
            "\n".join(
                f"  Pattern '{p}': {len(ids)} violations\n" +
                "\n".join(f"    - {id_}" for id_ in ids[:5])
                for p, ids in violations.items()
            )
        )

    def test_catalog_no_duplicates(self, canonical_ids):
        """[Constraint 1] Catalog must not contain duplicate IDs."""
        catalog = load_json(CATALOG_PATH)
        all_ids = [entry['unique_id'] for entry in catalog if 'unique_id' in entry]

        duplicates = [id_ for id_ in all_ids if all_ids.count(id_) > 1]
        unique_duplicates = list(set(duplicates))

        assert len(unique_duplicates) == 0, (
            f"Found {len(unique_duplicates)} duplicate IDs in catalog:\n" +
            "\n".join(f"  - {id_} (appears {all_ids.count(id_)}x)"
                     for id_ in unique_duplicates[:10])
        )


class TestCrossFileConsistency:
    """Test suite for cross-file consistency validation."""

    def test_layers_json_uses_canonical_ids_only(self, canonical_ids):
        """[Constraint 1] All IDs in layers JSON must exist in canonical catalog."""
        layers_ids = extract_method_ids(LAYERS_PATH, 'layers')
        non_canonical = [id_ for id_ in layers_ids if id_ not in canonical_ids]

        assert len(non_canonical) == 0, (
            f"Found {len(non_canonical)} non-canonical IDs in layers JSON:\n" +
            "\n".join(f"  - {id_}" for id_ in non_canonical)
        )

    def test_intrinsic_json_uses_canonical_ids_only(self, canonical_ids):
        """[Constraint 1] All IDs in intrinsic calibration JSON must exist in canonical catalog."""
        intrinsic_ids = extract_method_ids(INTRINSIC_PATH, 'intrinsic')
        non_canonical = [id_ for id_ in intrinsic_ids if id_ not in canonical_ids]

        assert len(non_canonical) == 0, (
            f"Found {len(non_canonical)} non-canonical IDs in intrinsic JSON:\n" +
            "\n".join(f"  - {id_}" for id_ in non_canonical)
        )

    def test_parametrized_json_uses_canonical_ids_only(self, canonical_ids):
        """[Constraint 1] All IDs in parametrized JSON must exist in canonical catalog."""
        # Skip this test if file format is unexpected
        try:
            param_ids = extract_method_ids(PARAMETRIZED_PATH, 'parametrized')
        except Exception:
            pytest.skip("Parametrized JSON has unexpected format")
            return

        non_canonical = [id_ for id_ in param_ids if id_ not in canonical_ids]

        assert len(non_canonical) == 0, (
            f"Found {len(non_canonical)} non-canonical IDs in parametrized JSON:\n" +
            "\n".join(f"  - {id_}" for id_ in non_canonical[:10])
        )


class TestCatalogIntegrity:
    """Test suite for catalog structural integrity."""

    def test_catalog_is_list(self):
        """Catalog must be a JSON list."""
        catalog = load_json(CATALOG_PATH)
        assert isinstance(catalog, list), "Catalog must be a list of method entries"

    def test_catalog_entries_have_unique_id(self):
        """All catalog entries must have 'unique_id' field."""
        catalog = load_json(CATALOG_PATH)
        missing_id = [i for i, entry in enumerate(catalog) if 'unique_id' not in entry]

        assert len(missing_id) == 0, (
            f"Found {len(missing_id)} entries without 'unique_id' field:\n" +
            f"  Indices: {missing_id[:10]}"
        )

    def test_catalog_minimum_size(self, canonical_ids):
        """Catalog must contain reasonable number of methods."""
        # Should have at least 30 executors + other methods
        assert len(canonical_ids) >= 30, (
            f"Catalog only contains {len(canonical_ids)} methods. "
            f"Expected at least 30 (6 dimensions x 5 questions)."
        )


if __name__ == '__main__':
    # Allow running as script for quick validation
    pytest.main([__file__, '-v'])


================================================================================
FILE: tests/calibration_system/test_intrinsic_coverage.py
================================================================================

"""
Test 3: Intrinsic Coverage - â\211¥80% Methods + All 30 Executors Have status='computed'

Validates calibration completeness:
- At least 80% of all methods must have calibration status='computed'
- All 30 executors must have at least one method with status='computed'
- No executor should be entirely 'excluded' or 'manual'

FAILURE CONDITION: Coverage < 80% OR any executor without computed status = NOT READY
"""
import json
import pytest
from pathlib import Path
from typing import Dict, List, Set, Any


class TestIntrinsicCoverage:

    MIN_COVERAGE_PERCENT = 80.0

    @pytest.fixture(scope="class")
    def executors_methods(self) -> Dict[str, Any]:
        """Load executors_methods.json"""
        path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
        with open(path) as f:
            return json.load(f)

    @pytest.fixture(scope="class")
    def intrinsic_calibration(self) -> Dict[str, Any]:
        """Load intrinsic_calibration.json"""
        path = Path("system/config/calibration/intrinsic_calibration.json")
        with open(path) as f:
            return json.load(f)

    def test_intrinsic_calibration_not_empty(self, intrinsic_calibration):
        """Verify intrinsic_calibration.json is not empty"""
        methods = {k: v for k, v in intrinsic_calibration.items() if k != "_metadata"}
        assert len(methods) > 0, "intrinsic_calibration.json has no method entries"

    def test_80_percent_coverage_with_computed_status(
        self, executors_methods, intrinsic_calibration
    ):
        """CRITICAL: Verify at least 80% of methods have status='computed'"""
        all_methods = set()
        for executor in executors_methods:
            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"
                all_methods.add(method_id)

        total_methods = len(all_methods)
        computed_methods = set()

        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            if method_id in all_methods:
                status = calibration.get("status", "missing")
                if status == "computed":
                    computed_methods.add(method_id)

        coverage_percent = (len(computed_methods) / total_methods * 100) if total_methods > 0 else 0

        missing_computed = all_methods - computed_methods

        failure_msg = (
            f"CRITICAL: Calibration coverage is {coverage_percent:.1f}% "
            f"({len(computed_methods)}/{total_methods}), "
            f"minimum required is {self.MIN_COVERAGE_PERCENT}%\n"
            f"Methods without 'computed' status: {len(missing_computed)}\n"
        )

        if missing_computed and len(missing_computed) <= 20:
            failure_msg += "Missing methods:\n" + "\n".join(
                f"  - {m}" for m in sorted(missing_computed)
            )

        assert coverage_percent >= self.MIN_COVERAGE_PERCENT, failure_msg

    def test_all_30_executors_have_computed_methods(
        self, executors_methods, intrinsic_calibration
    ):
        """CRITICAL: Every executor must have at least one method with status='computed'"""
        failures = []

        for executor in executors_methods:
            executor_id = executor["executor_id"]
            has_computed = False

            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"
                calibration = intrinsic_calibration.get(method_id, {})

                if calibration.get("status") == "computed":
                    has_computed = True
                    break

            if not has_computed:
                failures.append(
                    f"Executor {executor_id} has no methods with status='computed'"
                )

        assert not failures, \
            f"CRITICAL: Executors without computed calibrations:\n" + "\n".join(failures)

    def test_status_field_valid_values(self, intrinsic_calibration):
        """Verify all status fields have valid values"""
        valid_statuses = {"computed", "excluded", "manual", "error"}

        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            status = calibration.get("status")
            assert status in valid_statuses, \
                f"Invalid status '{status}' for {method_id}. " \
                f"Must be one of: {valid_statuses}"

    def test_computed_methods_have_scores(self, intrinsic_calibration):
        """Verify methods with status='computed' have actual scores"""
        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            if calibration.get("status") == "computed":
                assert "b_theory" in calibration or "scores" in calibration, \
                    f"Method {method_id} has status='computed' but no scores"

    def test_no_fully_excluded_executors(self, executors_methods, intrinsic_calibration):
        """Warn if any executor has all methods excluded"""
        warnings = []

        for executor in executors_methods:
            executor_id = executor["executor_id"]
            all_excluded = True

            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"
                calibration = intrinsic_calibration.get(method_id, {})

                if calibration.get("status") != "excluded":
                    all_excluded = False
                    break

            if all_excluded:
                warnings.append(
                    f"WARNING: Executor {executor_id} has all methods excluded"
                )

        if warnings:
            print("\n" + "\n".join(warnings))

    def test_excluded_methods_have_reason(self, intrinsic_calibration):
        """Verify excluded methods have exclusion reason"""
        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            if calibration.get("status") == "excluded":
                assert "exclusion_reason" in calibration or "reason" in calibration, \
                    f"Excluded method {method_id} missing exclusion reason"

    def test_manual_methods_documented(self, intrinsic_calibration):
        """Verify methods with status='manual' are documented"""
        manual_methods = []

        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            if calibration.get("status") == "manual":
                manual_methods.append(method_id)
                assert "note" in calibration or "reason" in calibration, \
                    f"Manual method {method_id} missing documentation"

        if manual_methods:
            print(f"\nFound {len(manual_methods)} manual calibrations: {manual_methods[:5]}")

    def test_error_status_methods_tracked(self, intrinsic_calibration):
        """Track methods with status='error' for debugging"""
        error_methods = []

        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            if calibration.get("status") == "error":
                error_methods.append(method_id)

        if error_methods:
            print(
                f"\nWARNING: {len(error_methods)} methods have status='error': "
                f"{error_methods[:5]}"
            )

    def test_coverage_by_executor(self, executors_methods, intrinsic_calibration):
        """Report coverage statistics per executor"""
        stats = []

        for executor in executors_methods:
            executor_id = executor["executor_id"]
            total = len(executor["methods"])
            computed = 0

            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"
                calibration = intrinsic_calibration.get(method_id, {})

                if calibration.get("status") == "computed":
                    computed += 1

            coverage = (computed / total * 100) if total > 0 else 0
            stats.append((executor_id, coverage, computed, total))

        stats.sort(key=lambda x: x[1])

        low_coverage = [s for s in stats if s[1] < self.MIN_COVERAGE_PERCENT]

        if low_coverage:
            msg = "\nExecutors with coverage below threshold:\n"
            for executor_id, coverage, computed, total in low_coverage[:10]:
                msg += f"  {executor_id}: {coverage:.1f}% ({computed}/{total})\n"
            print(msg)

    def test_metadata_present_in_intrinsic(self, intrinsic_calibration):
        """Verify _metadata is present with version info"""
        assert "_metadata" in intrinsic_calibration, \
            "intrinsic_calibration.json missing _metadata"

        metadata = intrinsic_calibration["_metadata"]
        assert "version" in metadata, "_metadata missing version"
        assert "generated" in metadata or "last_updated" in metadata, \
            "_metadata missing timestamp"



================================================================================
FILE: tests/calibration_system/test_inventory_consistency.py
================================================================================

"""
Test 1: Inventory Consistency - Verifying Same Methods in All JSONs

Validates that all executor method inventories are consistent across:
- executors_methods.json: 30 executors with method lists
- intrinsic_calibration.json: All methods have calibration entries
- canonical method catalog (if exists)

FAILURE CONDITION: Any method missing from any inventory = SYSTEM NOT READY
"""
import json
import pytest
from pathlib import Path
from typing import Dict, List, Set, Any


class TestInventoryConsistency:
    @pytest.fixture(scope="class")
    def executors_methods(self) -> Dict[str, Any]:
        """Load executors_methods.json"""
        path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
        assert path.exists(), f"executors_methods.json not found at {path}"
        with open(path) as f:
            return json.load(f)

    @pytest.fixture(scope="class")
    def intrinsic_calibration(self) -> Dict[str, Any]:
        """Load intrinsic_calibration.json"""
        path = Path("system/config/calibration/intrinsic_calibration.json")
        assert path.exists(), f"intrinsic_calibration.json not found at {path}"
        with open(path) as f:
            return json.load(f)

    @pytest.fixture(scope="class")
    def calibration_rubric(self) -> Dict[str, Any]:
        """Load calibration rubric"""
        path = Path("config/intrinsic_calibration_rubric.json")
        assert path.exists(), f"calibration rubric not found at {path}"
        with open(path) as f:
            return json.load(f)

    def test_executors_methods_structure(self, executors_methods):
        """Verify executors_methods.json has correct structure"""
        assert isinstance(executors_methods, list), "executors_methods must be a list"
        assert len(executors_methods) > 0, "executors_methods cannot be empty"

        for executor in executors_methods:
            assert "executor_id" in executor, f"Executor missing executor_id: {executor}"
            assert "methods" in executor, f"Executor {executor.get('executor_id')} missing methods"
            assert isinstance(executor["methods"], list), \
                f"Executor {executor['executor_id']} methods must be a list"

    def test_30_executors_present(self, executors_methods):
        """Verify exactly 30 executors are defined"""
        executor_ids = [e["executor_id"] for e in executors_methods]
        assert len(executor_ids) == 30, \
            f"Expected 30 executors, found {len(executor_ids)}: {executor_ids}"

        expected_pattern = set()
        for d in range(1, 7):
            for q in range(1, 6):
                expected_pattern.add(f"D{d}-Q{q}")

        actual_set = set(executor_ids)
        missing = expected_pattern - actual_set
        extra = actual_set - expected_pattern

        assert not missing, f"Missing executors: {sorted(missing)}"
        assert not extra, f"Unexpected executors: {sorted(extra)}"

    def test_all_methods_unique_per_executor(self, executors_methods):
        """Verify no duplicate methods within each executor"""
        for executor in executors_methods:
            methods = executor["methods"]
            method_sigs = [f"{m['class']}.{m['method']}" for m in methods]
            duplicates = [sig for sig in method_sigs if method_sigs.count(sig) > 1]

            assert not duplicates, \
                f"Executor {executor['executor_id']} has duplicate methods: {set(duplicates)}"

    def test_method_format_consistency(self, executors_methods):
        """Verify all methods have required fields"""
        for executor in executors_methods:
            for method in executor["methods"]:
                assert "class" in method, \
                    f"Method in {executor['executor_id']} missing 'class': {method}"
                assert "method" in method, \
                    f"Method in {executor['executor_id']} missing 'method': {method}"
                assert isinstance(method["class"], str) and method["class"], \
                    f"Invalid class name in {executor['executor_id']}: {method}"
                assert isinstance(method["method"], str) and method["method"], \
                    f"Invalid method name in {executor['executor_id']}: {method}"

    def test_intrinsic_calibration_covers_all_methods(
        self, executors_methods, intrinsic_calibration
    ):
        """CRITICAL: Verify all executor methods have intrinsic calibration entries"""
        all_methods = set()
        for executor in executors_methods:
            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"
                all_methods.add(method_id)

        calibrated_methods = set(intrinsic_calibration.keys()) - {"_metadata"}

        missing_calibrations = all_methods - calibrated_methods
        extra_calibrations = calibrated_methods - all_methods

        failure_report = []
        if missing_calibrations:
            failure_report.append(
                f"CRITICAL: {len(missing_calibrations)} methods missing calibration:\n"
                + "\n".join(f"  - {m}" for m in sorted(missing_calibrations)[:20])
            )

        if extra_calibrations:
            failure_report.append(
                f"WARNING: {len(extra_calibrations)} calibrations for unknown methods:\n"
                + "\n".join(f"  - {m}" for m in sorted(extra_calibrations)[:20])
            )

        assert not missing_calibrations, "\n".join(failure_report)

    def test_calibration_status_field_present(self, intrinsic_calibration):
        """Verify all calibration entries have status field"""
        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            assert isinstance(calibration, dict), \
                f"Calibration for {method_id} must be a dict, got {type(calibration)}"

            assert "status" in calibration, \
                f"Calibration for {method_id} missing 'status' field"

            valid_statuses = {"computed", "excluded", "manual", "error"}
            assert calibration["status"] in valid_statuses, \
                f"Invalid status for {method_id}: {calibration['status']}"

    def test_no_empty_method_lists(self, executors_methods):
        """Verify no executor has empty method list"""
        for executor in executors_methods:
            assert len(executor["methods"]) > 0, \
                f"Executor {executor['executor_id']} has no methods"

    def test_method_naming_conventions(self, executors_methods):
        """Verify method names follow Python conventions"""
        import re

        class_pattern = re.compile(r'^[A-Z][a-zA-Z0-9]*$')
        method_pattern = re.compile(r'^[a-z_][a-z0-9_]*$')

        for executor in executors_methods:
            for method in executor["methods"]:
                class_name = method["class"]
                method_name = method["method"]

                assert class_pattern.match(class_name), \
                    f"Invalid class name in {executor['executor_id']}: {class_name}"
                assert method_pattern.match(method_name), \
                    f"Invalid method name in {executor['executor_id']}: {method_name}"



================================================================================
FILE: tests/calibration_system/test_layer_correctness.py
================================================================================

"""
Test 2: Layer Correctness - 30 Executors Have 8 Layers + Correct LAYER_REQUIREMENTS

Validates architectural integrity:
- All 30 executors must define exactly 8 layers
- LAYER_REQUIREMENTS mapping must be correct and complete
- Layer dependencies must be acyclic

FAILURE CONDITION: Any layer mismatch = SYSTEM NOT READY
"""
import json
import pytest
from pathlib import Path
from typing import Dict, List, Set, Any


class TestLayerCorrectness:

    REQUIRED_LAYERS = {
        "ingestion",
        "extraction",
        "transformation",
        "validation",
        "aggregation",
        "scoring",
        "reporting",
        "meta"
    }

    @pytest.fixture(scope="class")
    def executors_methods(self) -> Dict[str, Any]:
        """Load executors_methods.json"""
        path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
        with open(path) as f:
            return json.load(f)

    @pytest.fixture(scope="class")
    def layer_requirements_path(self) -> Path:
        """Find LAYER_REQUIREMENTS definition"""
        candidates = [
            Path("src/farfan_pipeline/core/orchestrator/executors.py"),
            Path("src/farfan_pipeline/core/orchestrator/base_executor_with_contract.py"),
            Path("src/farfan_pipeline/core/layers.py"),
        ]
        for path in candidates:
            if path.exists():
                return path
        pytest.skip("LAYER_REQUIREMENTS definition file not found")

    def test_8_layers_defined(self):
        """Verify exactly 8 layers are required"""
        assert len(self.REQUIRED_LAYERS) == 8, \
            f"Expected 8 layers, found {len(self.REQUIRED_LAYERS)}"

    def test_executor_methods_have_layer_field(self, executors_methods):
        """Verify all methods specify a layer"""
        for executor in executors_methods:
            for method in executor["methods"]:
                method_sig = f"{method['class']}.{method['method']}"

                assert "layer" in method, \
                    f"Method {method_sig} in {executor['executor_id']} missing 'layer' field"

                layer = method["layer"]
                assert layer in self.REQUIRED_LAYERS, \
                    f"Invalid layer ’{layer}’ for {method_sig} in {executor[’executor_id’]}. " \
                    f"Must be one of: {sorted(self.REQUIRED_LAYERS)}"

    def test_all_30_executors_have_all_8_layers(self, executors_methods):
        """CRITICAL: Verify each of 30 executors has methods in all 8 layers"""
        failures = []

        for executor in executors_methods:
            executor_id = executor["executor_id"]
            layers_present = set()

            for method in executor["methods"]:
                if "layer" in method:
                    layers_present.add(method["layer"])

            missing_layers = self.REQUIRED_LAYERS - layers_present

            if missing_layers:
                failures.append(
                    f"Executor {executor_id} missing layers: {sorted(missing_layers)}"
                )

        assert not failures, \
            f"CRITICAL: Executors with incomplete layer coverage:\n" + \
            "\n".join(failures)

    def test_layer_requirements_mapping_exists(self, layer_requirements_path):
        """Verify LAYER_REQUIREMENTS mapping is defined in code"""
        content = layer_requirements_path.read_text()

        assert "LAYER_REQUIREMENTS" in content, \
            f"LAYER_REQUIREMENTS not found in {layer_requirements_path}"

    def test_layer_requirements_covers_all_layers(self, layer_requirements_path):
        """Verify LAYER_REQUIREMENTS covers all 8 layers"""
        import ast

        content = layer_requirements_path.read_text()
        tree = ast.parse(content)

        layer_req_dict = None
        for node in ast.walk(tree):
            if isinstance(node, ast.Assign):
                for target in node.targets:
                    if isinstance(target, ast.Name) and target.id == "LAYER_REQUIREMENTS":
                        if isinstance(node.value, ast.Dict):
                            layer_req_dict = {}
                            for k, v in zip(node.value.keys, node.value.values):
                                if isinstance(k, ast.Constant):
                                    layer_req_dict[k.value] = True

        if layer_req_dict is None:
            pytest.skip("Could not parse LAYER_REQUIREMENTS from source")
```

```python

        defined_layers = set(layer_req_dict.keys())
        missing_layers = self.REQUIRED_LAYERS - defined_layers

        assert not missing_layers, \
            f"LAYER_REQUIREMENTS missing layers: {sorted(missing_layers)}"

    def test_layer_dependency_acyclicity(self, executors_methods):
        """Verify layer execution order is acyclic"""
        layer_order = [
            "ingestion",
            "extraction",
            "transformation",
            "validation",
            "aggregation",
            "scoring",
            "reporting",
            "meta"
        ]

        layer_to_position = {layer: i for i, layer in enumerate(layer_order)}

        for executor in executors_methods:
            for i, method in enumerate(executor["methods"]):
                if "layer" not in method:
                    continue

                current_layer = method["layer"]
                current_pos = layer_to_position.get(current_layer, -1)

                for prev_method in executor["methods"][:i]:
                    if "layer" not in prev_method:
                        continue
                    prev_layer = prev_method["layer"]
                    prev_pos = layer_to_position.get(prev_layer, -1)

                    if current_pos < prev_pos:
                        pytest.fail(
                            f"Layer ordering violation in {executor['executor_id']}: "
                            f"{prev_layer} (pos {prev_pos}) comes before "
                            f"{current_layer} (pos {current_pos})"
                        )

    def test_method_distribution_across_layers(self, executors_methods):
        """Analyze method distribution across layers"""
        layer_counts = {layer: 0 for layer in self.REQUIRED_LAYERS}

        for executor in executors_methods:
            for method in executor["methods"]:
                if "layer" in method:
                    layer_counts[method["layer"]] += 1

        for layer, count in layer_counts.items():
            assert count > 0, \
                f"Layer '{layer}' has no methods across all executors"

    def test_layer_field_consistency_with_method_names(self, executors_methods):
        """Verify layer assignment consistency with method naming patterns"""
        layer_hints = {
            "extract": "extraction",
            "ingest": "ingestion",
            "validate": "validation",
            "transform": "transformation",
            "aggregate": "aggregation",
            "score": "scoring",
            "report": "reporting",
        }

        warnings = []

        for executor in executors_methods:
            for method in executor["methods"]:
                if "layer" not in method:
                    continue

                method_name = method["method"].lower()
                assigned_layer = method["layer"]

                for hint, suggested_layer in layer_hints.items():
                    if hint in method_name and assigned_layer != suggested_layer:
                        warnings.append(
                            f"Potential layer mismatch: {method['class']}.{method['method']} "
                            f"in {executor['executor_id']} assigned to '{assigned_layer}' "
                            f"but name suggests '{suggested_layer}'"
                        )

        if warnings:
            print("\n".join(warnings[:10]))

    def test_no_orphan_layers(self, executors_methods):
        """Verify no executor has layer gaps in sequence"""
        layer_order = [
            "ingestion",
            "extraction",
            "transformation",
            "validation",
            "aggregation",
            "scoring",
            "reporting",
            "meta"
        ]

        for executor in executors_methods:
            layers_used = []
            for method in executor["methods"]:
                if "layer" in method:
                    layers_used.append(method["layer"])

            layer_positions = sorted([
                layer_order.index(layer) for layer in set(layers_used)
            ])

            if not layer_positions:
                continue

            for i in range(len(layer_positions) - 1):
                gap = layer_positions[i + 1] - layer_positions[i]
                if gap > 2:
                    pytest.fail(
                        f"Large layer gap in {executor['executor_id']}: "
                        f"{layer_order[layer_positions[i]]} -> "
                        f"{layer_order[layer_positions[i + 1]]}"
                    )



================================================================================
FILE: tests/calibration_system/test_no_hardcoded_calibrations.py
================================================================================

"""
Test 6: No Hardcoded Calibrations - Scanning for Magic Numbers in Calibration-Sensitive Areas

Validates that calibration parameters are not hardcoded:
- Scans source code for hardcoded thresholds, weights, scores
- Excludes test files and configuration files
- Fails if magic numbers found in calibration-sensitive code

FAILURE CONDITION: Any hardcoded calibration values = NOT READY

Integration: This test suite integrates with the pre-commit hook system.
The pre-commit hook (scripts/pre_commit_validators.py) uses the same validation
logic to block commits containing hardcoded calibration patterns.
"""
import re
import pytest
from pathlib import Path
from typing import Any, List, Dict, Tuple, Set


class TestNoHardcodedCalibrations:

    CALIBRATION_PATTERNS = [
        (r'threshold\s*=\s*[0-9.]+', 'threshold assignment'),
        (r'weight\s*=\s*[0-9.]+', 'weight assignment'),
        (r'score\s*=\s*[0-9.]+', 'score assignment'),
        (r'min_evidence\s*=\s*[0-9]+', 'min_evidence assignment'),
        (r'max_evidence\s*=\s*[0-9]+', 'max_evidence assignment'),
        (r'confidence\s*=\s*[0-9.]+', 'confidence assignment'),
        (r'penalty\s*=\s*[0-9.]+', 'penalty assignment'),
        (r'tolerance\s*=\s*[0-9.]+', 'tolerance assignment'),
        (r'sensitivity\s*=\s*[0-9.]+', 'sensitivity assignment'),
    ]

    EXCLUDED_PATHS = [
        "test",
        "__pycache__",
        ".git",
        "farfan-env",
        "venv",
        ".venv",
        "node_modules",
        "config",
        "system/config",
    ]

    ALLOWED_VALUES = {
        '0', '1', '0.0', '1.0', '0.5',
        '-1', '2', '10', '100',
    }

    @pytest.fixture(scope="class")
    def source_files(self) -> List[Path]:
        """Collect all Python source files to scan"""
        files = []

        src_dirs = [
            Path("src/farfan_pipeline"),
            Path("farfan_core/farfan_core"),
        ]

        for src_dir in src_dirs:
            if not src_dir.exists():
                continue

            for py_file in src_dir.rglob("*.py"):
                if self._should_exclude(py_file):
                    continue

                files.append(py_file)

        return files

    def _should_exclude(self, path: Path) -> bool:
        """Check if path should be excluded from scanning"""
        path_str = str(path)

        for excluded in self.EXCLUDED_PATHS:
            if excluded in path_str:
                return True

        return False

    def test_source_files_found(self, source_files):
        """Verify we have source files to scan"""
        assert len(source_files) > 0, \
            "No source files found to scan for hardcoded calibrations"

        print(f"\nScanning {len(source_files)} source files")

    def test_no_hardcoded_thresholds(self, source_files):
        """CRITICAL: Verify no hardcoded threshold values in source"""
        violations = self._scan_for_patterns(source_files, "threshold")

        if violations:
            msg = self._format_violations(violations, "threshold")
            pytest.fail(msg)

    def test_no_hardcoded_weights(self, source_files):
        """CRITICAL: Verify no hardcoded weight values in source"""
        violations = self._scan_for_patterns(source_files, "weight")

        if violations:
            msg = self._format_violations(violations, "weight")
            pytest.fail(msg)

    def test_no_hardcoded_scores(self, source_files):
        """CRITICAL: Verify no hardcoded score values in source"""
        violations = self._scan_for_patterns(source_files, "score")

        if violations:
            msg = self._format_violations(violations, "score")
            pytest.fail(msg)

    def test_no_hardcoded_calibration_params(self, source_files):
        """CRITICAL: Comprehensive scan for all calibration patterns"""
        all_violations = []

        for file_path in source_files:
            try:
                content = file_path.read_text()
                lines = content.split('\n')

                for lineno, line in enumerate(lines, 1):
                    if self._is_comment_or_docstring(line):
                        continue

                    for pattern, description in self.CALIBRATION_PATTERNS:
                        matches = re.finditer(pattern, line, re.IGNORECASE)

                        for match in matches:
                            value = match.group().split('=')[-1].strip()

                            if value not in self.ALLOWED_VALUES:
                                all_violations.append({
                                    'file': str(file_path),
                                    'line': lineno,
                                    'code': line.strip(),
                                    'pattern': description,
                                    'value': value
                                })

            except (UnicodeDecodeError, OSError) as e:
                print(f"Warning: Could not read {file_path}: {e}")

        if all_violations:
            msg = self._format_all_violations(all_violations)
            pytest.fail(msg)

    def _scan_for_patterns(
        self, source_files: List[Path], keyword: str
    ) -> List[Dict[str, Any]]:
        """Scan files for specific calibration pattern"""
        violations = []
        pattern = re.compile(rf'{keyword}\s*=\s*([0-9.]+)', re.IGNORECASE)

        for file_path in source_files:
            try:
                content = file_path.read_text()
                lines = content.split('\n')

                for lineno, line in enumerate(lines, 1):
                    if self._is_comment_or_docstring(line):
                        continue

                    matches = pattern.finditer(line)

                    for match in matches:
                        value = match.group(1)

                        if value not in self.ALLOWED_VALUES:
                            violations.append({
                                'file': str(file_path),
                                'line': lineno,
                                'code': line.strip(),
                                'value': value
                            })

            except (UnicodeDecodeError, OSError):
                pass

        return violations

    def _is_comment_or_docstring(self, line: str) -> bool:
        """Check if line is a comment or docstring"""
        stripped = line.strip()
        return (
            stripped.startswith('#') or
            stripped.startswith('"""') or
            stripped.startswith("'''")
        )

    def _format_violations(
        self, violations: List[Dict[str, Any]], keyword: str
    ) -> str:
        """Format violation message"""
        msg = (
            f"\nCRITICAL: Found {len(violations)} hardcoded {keyword} values:\n\n"
        )

        for v in violations[:10]:
            msg += f"  {v['file']}:{v['line']}\n"
            msg += f"    {v['code']}\n"
            msg += f"    Value: {v['value']}\n\n"

        if len(violations) > 10:
            msg += f"  ... and {len(violations) - 10} more\n"

        msg += "\nAll calibration parameters must be loaded from configuration files.\n"
        msg += "Move these values to intrinsic_calibration.json or calibration_registry.py\n"

        return msg

    def _format_all_violations(self, violations: List[Dict[str, Any]]) -> str:
        """Format all violations message"""
        msg = (
            f"\nCRITICAL: Found {len(violations)} hardcoded calibration values:\n\n"
        )

        by_file = {}
        for v in violations:
            file_path = v['file']
            if file_path not in by_file:
                by_file[file_path] = []
            by_file[file_path].append(v)

        for file_path, file_violations in sorted(by_file.items())[:5]:
            msg += f"  {file_path}:\n"
            for v in file_violations[:3]:
                msg += f"    Line {v['line']}: {v['code']}\n"
                msg += f"      {v['pattern']}: {v['value']}\n"
            if len(file_violations) > 3:
                msg += f"    ... and {len(file_violations) - 3} more\n"
            msg += "\n"

        if len(by_file) > 5:
            msg += f"  ... and {len(by_file) - 5} more files\n"

        msg += "\nAll calibration parameters must be externalized to configuration.\n"

        return msg

    def test_calibration_registry_used(self, source_files):
        """Verify calibration_registry is imported where needed"""
        imports_found = 0

        for file_path in source_files:
            if "calibration" in str(file_path).lower():
                continue

            try:
                content = file_path.read_text()

                if "calibration_registry" in content or "get_calibration" in content:
                    imports_found += 1

            except (UnicodeDecodeError, OSError):
                pass

        print(f"\nFound {imports_found} files using calibration_registry")

    def test_pre_commit_hook_exists(self):
        """Verify pre-commit hook is installed"""
        hook_path = Path(".git/hooks/pre-commit")
        assert hook_path.exists(), "Pre-commit hook not installed"

        content = hook_path.read_text()
        assert "pre_commit_validators.py" in content, \
            "Pre-commit hook does not call validators"

        print("\nâ\234\223 Pre-commit hook is installed and configured")

    def test_pre_commit_validators_exist(self):
        """Verify pre-commit validator scripts exist"""
        validators_path = Path("scripts/pre_commit_validators.py")
        assert validators_path.exists(), "Pre-commit validators script missing"

        update_hash_path = Path("scripts/update_hash_registry.py")
        assert update_hash_path.exists(), "Hash registry update script missing"

        print("\nâ\234\223 Pre-commit validator scripts exist")

    def test_no_inline_bayesian_priors(self, source_files):
        """Check for hardcoded Bayesian priors"""
        violations = []
        pattern = re.compile(r'prior\s*=\s*([0-9.]+)', re.IGNORECASE)

        for file_path in source_files:
            try:
                content = file_path.read_text()
                lines = content.split('\n')

                for lineno, line in enumerate(lines, 1):
                    if self._is_comment_or_docstring(line):
                        continue

                    if pattern.search(line):
                        violations.append({
                            'file': str(file_path),
                            'line': lineno,
                            'code': line.strip()
                        })

            except (UnicodeDecodeError, OSError):
                pass

        if violations:
            msg = f"\nFound {len(violations)} hardcoded prior values:\n"
            for v in violations[:5]:
                msg += f"  {v['file']}:{v['line']}: {v['code']}\n"
            print(msg)



================================================================================
FILE: tests/calibration_system/test_orchestrator_runtime.py
================================================================================

"""
Test 5: Orchestrator Runtime - Correct Layer Evaluation + Aggregation with Real Contexts

Validates runtime behavior:
- Layers execute in correct order
- Method results aggregate properly
- Context passing works correctly
- No runtime exceptions in calibration resolution

FAILURE CONDITION: Any runtime error OR incorrect aggregation = NOT READY

DEPRECATED: Test assumes outdated executors_methods.json structure and layer execution order.
See tests/DEPRECATED_TESTS.md for details.
"""
import json
import pytest
from pathlib import Path
from typing import Dict, List, Any
from unittest.mock import Mock, patch, MagicMock

pytestmark = pytest.mark.obsolete


class TestOrchestratorRuntime:

    @pytest.fixture(scope="class")
    def executors_methods(self) -> Dict[str, Any]:
        """Load executors_methods.json"""
        path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
        with open(path) as f:
            return json.load(f)

    @pytest.fixture(scope="class")
    def intrinsic_calibration(self) -> Dict[str, Any]:
        """Load intrinsic_calibration.json"""
        path = Path("system/config/calibration/intrinsic_calibration.json")
        with open(path) as f:
            return json.load(f)

    @pytest.fixture
    def mock_context(self) -> Dict[str, Any]:
        """Create a mock execution context"""
        return {
            "document_text": "Sample policy document for testing",
            "raw_text": "Sample policy document for testing",
            "document_id": "test_doc_001",
            "metadata": {
                "source": "test",
                "year": 2024
            }
        }

    def test_layer_execution_order_defined(self, executors_methods):
        """Verify layer execution order is consistent"""
        expected_order = [
            "ingestion",
            "extraction",
            "transformation",
            "validation",
            "aggregation",
            "scoring",
            "reporting",
            "meta"
        ]

        for executor in executors_methods:
            layers_sequence = []

            for method in executor["methods"]:
                if "layer" in method:
                    layer = method["layer"]
                    if not layers_sequence or layers_sequence[-1] != layer:
                        layers_sequence.append(layer)

            for i, layer in enumerate(layers_sequence):
                expected_pos = expected_order.index(layer)

                for j in range(i + 1, len(layers_sequence)):
                    next_layer = layers_sequence[j]
                    next_pos = expected_order.index(next_layer)

                    assert expected_pos <= next_pos, \
                        f"Layer order violation in {executor['executor_id']}: " \
                        f"{layer} (pos {expected_pos}) followed by {next_layer} (pos {next_pos})"

    def test_calibration_context_resolution(self, intrinsic_calibration):
        """Test that calibration context can be resolved without errors"""
        try:
            from src.farfan_pipeline.core.orchestrator.calibration_context import (
                CalibrationContext,
                resolve_contextual_calibration,
            )
        except ImportError:
            pytest.skip("calibration_context module not available")

        test_question_ids = ["D1Q1", "D3Q5", "D6Q3"]

        for qid in test_question_ids:
            try:
                context = CalibrationContext.from_question_id(qid)
                assert context.question_id == qid
                assert context.dimension > 0
                assert context.question_num > 0
            except Exception as e:
                pytest.fail(f"Failed to create context for {qid}: {e}")

    def test_method_execution_simulation(self, executors_methods, mock_context):
        """Simulate method execution to verify no obvious runtime errors"""
        sample_executor = executors_methods[0]

        for method in sample_executor["methods"][:3]:
            method_id = f"{method['class']}.{method['method']}"

            try:
                result = self._simulate_method_call(
                    method["class"],
                    method["method"],
                    mock_context
                )

                assert result is not None or result == {}, \
                    f"Method {method_id} returned None unexpectedly"

            except NotImplementedError:
                pass
            except Exception as e:
                pytest.fail(
                    f"Unexpected error simulating {method_id}: {type(e).__name__}: {e}"
                )

    def _simulate_method_call(
        self, class_name: str, method_name: str, context: Dict[str, Any]
    ) -> Any:
        """Simulate a method call (mock implementation)"""
        if method_name.startswith("_"):
            return {}

        return {"status": "simulated", "method": f"{class_name}.{method_name}"}

    def test_aggregation_logic_validation(self, executors_methods):
        """Verify aggregation layer methods are present"""
        aggregation_methods = []

        for executor in executors_methods:
            for method in executor["methods"]:
                if method.get("layer") == "aggregation":
                    aggregation_methods.append({
                        "executor": executor["executor_id"],
                        "method": f"{method['class']}.{method['method']}"
                    })

        assert len(aggregation_methods) > 0, \
            "No aggregation methods found across all executors"

        print(f"\nFound {len(aggregation_methods)} aggregation methods")

    def test_scoring_methods_present(self, executors_methods):
        """Verify scoring layer methods are present"""
        scoring_methods = []

        for executor in executors_methods:
            for method in executor["methods"]:
                if method.get("layer") == "scoring":
                    scoring_methods.append({
                        "executor": executor["executor_id"],
                        "method": f"{method['class']}.{method['method']}"
                    })

        assert len(scoring_methods) > 0, \
            "No scoring methods found across all executors"

        print(f"\nFound {len(scoring_methods)} scoring methods")

    def test_reporting_layer_completeness(self, executors_methods):
        """Verify all executors have reporting capability"""
        executors_without_reporting = []

        for executor in executors_methods:
            has_reporting = any(
                method.get("layer") == "reporting"
                for method in executor["methods"]
            )

            if not has_reporting:
                executors_without_reporting.append(executor["executor_id"])

        if executors_without_reporting:
            print(
                f"\nWARNING: {len(executors_without_reporting)} executors "
                f"without reporting layer: {executors_without_reporting[:5]}"
            )

    def test_context_propagation_structure(self, mock_context):
        """Verify context has expected structure for method execution"""
        required_fields = ["document_text", "raw_text"]

        for field in required_fields:
            assert field in mock_context, \
                f"Mock context missing required field: {field}"

    def test_no_circular_dependencies_in_layers(self, executors_methods):
        """Verify no circular dependencies between layers"""
        layer_dependencies = {}

        for executor in executors_methods:
            for i, method in enumerate(executor["methods"]):
                if "layer" not in method:
                    continue

                current_layer = method["layer"]

                if current_layer not in layer_dependencies:
                    layer_dependencies[current_layer] = set()

                for prev_method in executor["methods"][:i]:
                    if "layer" in prev_method:
                        layer_dependencies[current_layer].add(prev_method["layer"])

        for layer, deps in layer_dependencies.items():
            assert layer not in deps, \
                f"Circular dependency detected: layer {layer} depends on itself"

    def test_calibration_values_resolvable(self, intrinsic_calibration):
        """Verify calibration values can be accessed without errors"""
        for method_id, calibration in intrinsic_calibration.items():
            if method_id == "_metadata":
                continue

            if calibration.get("status") == "computed":
                assert isinstance(calibration, dict), \
                    f"Calibration for {method_id} must be a dict"



================================================================================
FILE: tests/calibration_system/test_performance_benchmarks.py
================================================================================

"""
Test 7: Performance Benchmarks - Load Times and Calibration Speed

Validates system performance:
- Load intrinsic.json: < 1 second
- Calibrate 30 executors: < 5 seconds
- Calibrate 200 methods: < 30 seconds

FAILURE CONDITION: Any performance threshold exceeded = NOT READY

DEPRECATED: Test uses hardcoded paths to system/config/calibration/intrinsic_calibration.json.
See tests/DEPRECATED_TESTS.md for details.
"""
import json
import time
import pytest
from pathlib import Path
from typing import Dict, Any, List

pytestmark = pytest.mark.obsolete


class TestPerformanceBenchmarks:

    MAX_LOAD_TIME_SECONDS = 1.0
    MAX_30_EXECUTORS_SECONDS = 5.0
    MAX_200_METHODS_SECONDS = 30.0

    @pytest.fixture(scope="class")
    def intrinsic_path(self) -> Path:
        """Path to intrinsic_calibration.json"""
        return Path("system/config/calibration/intrinsic_calibration.json")

    @pytest.fixture(scope="class")
    def executors_path(self) -> Path:
        """Path to executors_methods.json"""
        return Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")

    def test_load_intrinsic_json_performance(self, intrinsic_path):
        """CRITICAL: Load intrinsic_calibration.json in < 1 second"""
        assert intrinsic_path.exists(), f"File not found: {intrinsic_path}"

        start_time = time.time()

        with open(intrinsic_path) as f:
            data = json.load(f)

        load_time = time.time() - start_time

        assert load_time < self.MAX_LOAD_TIME_SECONDS, \
            f"Loading intrinsic_calibration.json took {load_time:.3f}s, " \
            f"exceeds limit of {self.MAX_LOAD_TIME_SECONDS}s"

        print(f"\nLoaded intrinsic_calibration.json in {load_time:.3f}s")
        print(f"File size: {intrinsic_path.stat().st_size / 1024:.1f} KB")
        print(f"Number of methods: {len([k for k in data.keys() if k != '_metadata'])}")

    def test_load_executors_methods_performance(self, executors_path):
        """Load executors_methods.json performance"""
        assert executors_path.exists(), f"File not found: {executors_path}"

        start_time = time.time()

        with open(executors_path) as f:
            data = json.load(f)

        load_time = time.time() - start_time

        assert load_time < self.MAX_LOAD_TIME_SECONDS, \
            f"Loading executors_methods.json took {load_time:.3f}s, " \
            f"exceeds limit of {self.MAX_LOAD_TIME_SECONDS}s"

        print(f"\nLoaded executors_methods.json in {load_time:.3f}s")
        print(f"Number of executors: {len(data)}")

    def test_calibrate_30_executors_performance(
        self, intrinsic_path, executors_path
    ):
        """CRITICAL: Calibrate all 30 executors in < 5 seconds"""
        with open(intrinsic_path) as f:
            calibrations = json.load(f)

        with open(executors_path) as f:
            executors = json.load(f)

        assert len(executors) == 30, f"Expected 30 executors, found {len(executors)}"

        start_time = time.time()

        calibrated_count = 0
        for executor in executors:
            for method in executor["methods"]:
                method_id = f"{method['class']}.{method['method']}"

                if method_id in calibrations:
                    calibration = calibrations[method_id]

                    _ = calibration.get("status")
                    _ = calibration.get("b_theory")
                    _ = calibration.get("b_impl")
                    _ = calibration.get("b_deploy")

                    calibrated_count += 1

        calibration_time = time.time() - start_time

        assert calibration_time < self.MAX_30_EXECUTORS_SECONDS, \
            f"Calibrating 30 executors took {calibration_time:.3f}s, " \
            f"exceeds limit of {self.MAX_30_EXECUTORS_SECONDS}s"

        print(f"\nCalibrated {calibrated_count} methods across 30 executors in {calibration_time:.3f}s")
        print(f"Average: {calibration_time / 30:.3f}s per executor")

    def test_calibrate_200_methods_performance(self, intrinsic_path):
        """CRITICAL: Calibrate 200 methods in < 30 seconds"""
        with open(intrinsic_path) as f:
            calibrations = json.load(f)

        methods = [k for k in calibrations.keys() if k != "_metadata"]

        if len(methods) < 200:
            pytest.skip(f"Only {len(methods)} methods available, need 200 for benchmark")

        test_methods = methods[:200]

        start_time = time.time()

        for method_id in test_methods:
            calibration = calibrations[method_id]

            _ = calibration.get("status")
            _ = calibration.get("b_theory", 0.0)
            _ = calibration.get("b_impl", 0.0)
            _ = calibration.get("b_deploy", 0.0)

            if calibration.get("status") == "computed":
                _ = calibration.get("evidence", {})

        calibration_time = time.time() - start_time

        assert calibration_time < self.MAX_200_METHODS_SECONDS, \
            f"Calibrating 200 methods took {calibration_time:.3f}s, " \
            f"exceeds limit of {self.MAX_200_METHODS_SECONDS}s"

        print(f"\nCalibrated 200 methods in {calibration_time:.3f}s")
        print(f"Average: {calibration_time / 200 * 1000:.3f}ms per method")

    def test_json_parsing_overhead(self, intrinsic_path):
        """Measure JSON parsing overhead"""
        content = intrinsic_path.read_text()

        iterations = 10
        start_time = time.time()

        for _ in range(iterations):
            _ = json.loads(content)

        total_time = time.time() - start_time
        avg_time = total_time / iterations

        print(f"\nJSON parsing: {avg_time * 1000:.3f}ms per parse (avg of {iterations})")

    def test_calibration_lookup_performance(self, intrinsic_path):
        """Measure calibration lookup performance"""
        with open(intrinsic_path) as f:
            calibrations = json.load(f)

        methods = [k for k in calibrations.keys() if k != "_metadata"]

        if not methods:
            pytest.skip("No methods in calibration file")

        iterations = 10000
        test_method = methods[0]

        start_time = time.time()

        for _ in range(iterations):
            _ = calibrations.get(test_method)

        total_time = time.time() - start_time
        avg_time = total_time / iterations

        print(f"\nDictionary lookup: {avg_time * 1000000:.3f}Î¼s per lookup (avg of {iterations})")

    def test_file_size_reasonable(self, intrinsic_path):
        """Verify intrinsic_calibration.json size is reasonable"""
        MAX_SIZE_MB = 10

        size_bytes = intrinsic_path.stat().st_size
        size_mb = size_bytes / (1024 * 1024)

        assert size_mb < MAX_SIZE_MB, \
            f"intrinsic_calibration.json is {size_mb:.2f}MB, exceeds {MAX_SIZE_MB}MB limit"

        print(f"\nFile size: {size_mb:.2f}MB")

    def test_concurrent_access_simulation(self, intrinsic_path):
        """Simulate concurrent access to calibration data"""
        import threading

        results = []

        def load_calibrations():
            with open(intrinsic_path) as f:
                data = json.load(f)
            results.append(len(data))

        threads = []
        num_threads = 5

        start_time = time.time()

        for _ in range(num_threads):
            thread = threading.Thread(target=load_calibrations)
            threads.append(thread)
            thread.start()

        for thread in threads:
            thread.join()

        total_time = time.time() - start_time

        assert len(results) == num_threads, \
            f"Expected {num_threads} results, got {len(results)}"

        print(f"\n{num_threads} concurrent loads completed in {total_time:.3f}s")

    def test_memory_footprint(self, intrinsic_path):
        """Estimate memory footprint of calibration data"""
        import sys

        with open(intrinsic_path) as f:
            data = json.load(f)

        size_estimate = sys.getsizeof(data)

        for key, value in data.items():
            size_estimate += sys.getsizeof(key)
            size_estimate += sys.getsizeof(value)

            if isinstance(value, dict):
                for k, v in value.items():
                    size_estimate += sys.getsizeof(k) + sys.getsizeof(v)

        size_mb = size_estimate / (1024 * 1024)

        print(f"\nEstimated memory footprint: {size_mb:.2f}MB")



================================================================================
FILE: tests/calibration_system/test_pre_commit_validators.py
================================================================================

"""
Test Pre-Commit Validators Integration

Tests the pre-commit hook system that enforces:
1. No hardcoded calibration patterns
2. JSON schema compliance
3. SHA256 hash verification
4. YAML prohibition
"""

import json
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "scripts"))

from pre_commit_validators import (
    ALLOWED_VALUES,
    CALIBRATION_PATTERNS,
    validate_json_schema,
    validate_no_hardcoded_calibrations,
    validate_no_yaml_files,
)


class TestPreCommitValidators:

    def test_hardcoded_calibration_detection(self, tmp_path):
        """Test detection of hardcoded calibration values"""
        test_file = tmp_path / "test_module.py"
        test_file.write_text(
            """
def calculate_score():
    weight = 0.75
    threshold = 0.8
    score = 0.9
    return weight * threshold * score
"""
        )

        is_valid, errors = validate_no_hardcoded_calibrations([test_file])

        assert not is_valid, "Should detect hardcoded calibrations"
        assert len(errors) > 0
        assert any("weight" in str(e) for e in errors)
        assert any("threshold" in str(e) for e in errors)
        assert any("score" in str(e) for e in errors)

    def test_allowed_values_pass(self, tmp_path):
        """Test that allowed literal values pass validation"""
        test_file = tmp_path / "test_module.py"
        test_file.write_text(
            """
def initialize():
    weight = 1.0
    threshold = 0.5
    score = 0
    min_count = 2
    max_count = 10
    return weight, threshold, score
"""
        )

        is_valid, errors = validate_no_hardcoded_calibrations([test_file])

        assert is_valid, f"Allowed values should pass: {errors}"

    def test_comment_exclusion(self, tmp_path):
        """Test that comments are excluded from validation"""
        test_file = tmp_path / "test_module.py"
        test_file.write_text(
            """
def example():
    # weight = 0.75  # This is a comment
    return load_weight_from_config()
"""
        )

        is_valid, errors = validate_no_hardcoded_calibrations([test_file])

        assert is_valid, "Comments should be excluded"

    def test_json_validation_valid(self, tmp_path):
        """Test valid JSON passes validation"""
        test_file = tmp_path / "config.json"
        test_file.write_text(
            json.dumps({"version": "1.0", "settings": {"key": "value"}}, indent=2)
        )

        is_valid, errors = validate_json_schema([test_file])

        assert is_valid, f"Valid JSON should pass: {errors}"

    def test_json_validation_invalid(self, tmp_path):
        """Test invalid JSON is detected"""
        test_file = tmp_path / "config.json"
        test_file.write_text("{invalid json")

        is_valid, errors = validate_json_schema([test_file])

        assert not is_valid, "Invalid JSON should be detected"
        assert len(errors) > 0

    def test_yaml_prohibition(self, tmp_path):
        """Test that YAML files are blocked"""
        yaml_file = tmp_path / "config.yaml"
        yaml_file.write_text("key: value")

        is_valid, errors = validate_no_yaml_files([yaml_file])

        assert not is_valid, "YAML files should be blocked"
        assert len(errors) > 0
        assert any("YAML" in str(e) or "yaml" in str(e) for e in errors)

    def test_yml_extension_blocked(self, tmp_path):
        """Test that .yml extension is also blocked"""
        yml_file = tmp_path / "config.yml"
        yml_file.write_text("key: value")

        is_valid, errors = validate_no_yaml_files([yml_file])

        assert not is_valid, ".yml files should be blocked"

    def test_calibration_patterns_comprehensive(self):
        """Test that all calibration patterns are defined"""
        expected_params = [
            "weight",
            "score",
            "threshold",
            "min_evidence",
            "max_evidence",
            "confidence",
            "penalty",
            "tolerance",
            "sensitivity",
            "prior",
        ]

        pattern_params = [param for _, param in CALIBRATION_PATTERNS]

        for param in expected_params:
            assert (
                param in pattern_params
            ), f"Calibration parameter '{param}' not in patterns"

    def test_allowed_values_defined(self):
        """Test that allowed literal values are properly defined"""
        assert "0" in ALLOWED_VALUES
        assert "1" in ALLOWED_VALUES
        assert "0.0" in ALLOWED_VALUES
        assert "1.0" in ALLOWED_VALUES
        assert "0.5" in ALLOWED_VALUES
        assert "-1" in ALLOWED_VALUES

    def test_config_path_exclusion(self):
        """Test that config/ paths are excluded from calibration checks"""
        config_file = Path("config/test_settings.py")

        if config_file.exists():
            content = config_file.read_text()
            original_content = content
        else:
            config_file.parent.mkdir(parents=True, exist_ok=True)
            original_content = None

        try:
            config_file.write_text("weight = 0.75\nthreshold = 0.8\n")

            is_valid, errors = validate_no_hardcoded_calibrations(
                [config_file.absolute()]
            )

            assert is_valid, "Config files should be excluded"
        finally:
            if original_content is not None:
                config_file.write_text(original_content)
            elif config_file.exists():
                config_file.unlink()

    def test_multiple_violations_reported(self, tmp_path):
        """Test that multiple violations are all reported"""
        test_file = tmp_path / "violations.py"
        test_file.write_text(
            """
def bad_code():
    weight = 0.75
    score = 0.85
    threshold = 0.9
    penalty = 0.15
    sensitivity = 0.65
    return weight + score + threshold + penalty + sensitivity
"""
        )

        is_valid, errors = validate_no_hardcoded_calibrations([test_file])

        assert not is_valid
        error_text = "\n".join(errors)
        assert "weight" in error_text
        assert "score" in error_text
        assert "threshold" in error_text
        assert "penalty" in error_text
        assert "sensitivity" in error_text

    def test_string_literals_excluded(self, tmp_path):
        """Test that string literals with patterns are excluded"""
        test_file = tmp_path / "strings.py"
        test_file.write_text(
            """
def example():
    message = "weight = 0.75"
    description = "Set threshold = 0.8 for best results"
    return message, description
"""
        )

        is_valid, errors = validate_no_hardcoded_calibrations([test_file])

        assert is_valid, "String literals should be excluded"

    def test_pre_commit_hook_integration(self):
        """Test that pre-commit hook file exists and is executable"""
        hook_path = Path(".git/hooks/pre-commit")

        assert hook_path.exists(), "Pre-commit hook not installed"

        content = hook_path.read_text()
        assert "pre_commit_validators.py" in content
        assert "bash" in content or "sh" in content

    def test_hash_registry_script_exists(self):
        """Test that hash registry update script exists"""
        script_path = Path("scripts/update_hash_registry.py")

        assert script_path.exists(), "Hash registry script missing"

        content = script_path.read_text()
        assert "sha256" in content.lower()
        assert "hash" in content.lower()

    def test_calibration_registry_pattern_consistency(self):
        """Test that patterns match calibration_registry.py structure"""
        registry_path = Path(
            "src/farfan_pipeline/core/calibration/calibration_registry.py"
        )

        if registry_path.exists():
            content = registry_path.read_text()

            for _pattern, param_name in CALIBRATION_PATTERNS:
                if param_name in ["weight", "score"]:
                    assert (
                        param_name in content
                    ), f"Calibration registry should reference {param_name}"



================================================================================
FILE: tests/core/orchestrator/test_chunk_matrix_builder.py
================================================================================

"""Unit tests for chunk matrix builder module."""

import pytest
from pathlib import Path

from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance
from farfan_pipeline.core.orchestrator.chunk_matrix_builder import (
    build_chunk_matrix,
    EXPECTED_CHUNK_COUNT,
    POLICY_AREAS,
    DIMENSIONS,
)


def create_chunk(
    chunk_id: int,
    policy_area_id: str,
    dimension_id: str,
    text: str = "Test chunk content",
) -> ChunkData:
    """Helper to create a valid ChunkData instance."""
    return ChunkData(
        id=chunk_id,
        text=text,
        chunk_type="diagnostic",
        sentences=[],
        tables=[],
        start_pos=0,
        end_pos=len(text),
        confidence=0.9,
        chunk_id=f"{policy_area_id}-{dimension_id}",
        policy_area_id=policy_area_id,
        dimension_id=dimension_id,
        provenance=Provenance(page_number=1),
    )


def create_full_document() -> PreprocessedDocument:
    """Create a valid PreprocessedDocument with all 60 chunks."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            chunks.append(create_chunk(idx, pa, dim, f"Content for {pa}-{dim}"))
            idx += 1

    return PreprocessedDocument(
        document_id="test-doc-001",
        raw_text="Complete policy document text",
        sentences=[],
        tables=[],
        metadata={"chunk_count": 60},
        chunks=chunks,
        processing_mode="chunked",
    )


def test_build_chunk_matrix_success():
    """build_chunk_matrix should accept exactly 60 valid chunks."""
    doc = create_full_document()
    matrix, keys = build_chunk_matrix(doc)

    assert len(matrix) == EXPECTED_CHUNK_COUNT
    assert len(keys) == EXPECTED_CHUNK_COUNT
    assert keys[0] == ("PA01", "DIM01")
    assert keys[-1] == ("PA10", "DIM06")

    chunk = matrix[("PA05", "DIM03")]
    assert chunk.policy_area_id == "PA05"
    assert chunk.dimension_id == "DIM03"
    assert chunk.text == "Content for PA05-DIM03"


def test_build_chunk_matrix_sorted_keys():
    """Keys should be sorted deterministically by PA then DIM."""
    doc = create_full_document()
    _, keys = build_chunk_matrix(doc)

    assert keys[0] == ("PA01", "DIM01")
    assert keys[5] == ("PA01", "DIM06")
    assert keys[6] == ("PA02", "DIM01")
    assert keys[-1] == ("PA10", "DIM06")

    for i in range(len(keys) - 1):
        assert keys[i] < keys[i + 1], f"Keys not sorted: {keys[i]} >= {keys[i+1]}"

```

```python

def test_build_chunk_matrix_rejects_59_chunks():
    """build_chunk_matrix should reject documents with 59 chunks."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            if pa == "PA10" and dim == "DIM06":
                break
            chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    doc = PreprocessedDocument(
        document_id="test-doc-002",
        raw_text="Incomplete document",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError, match=r"Missing chunk combinations: .*PA10', 'DIM06'.*"):
        build_chunk_matrix(doc)


def test_build_chunk_matrix_rejects_61_chunks():
    """build_chunk_matrix should reject documents with 61 chunks."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    chunks.append(create_chunk(idx, "PA01", "DIM01", "Duplicate chunk"))

    doc = PreprocessedDocument(
        document_id="test-doc-003",
        raw_text="Document with duplicate",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError, match="Duplicate.*PA01-DIM01"):
        build_chunk_matrix(doc)


def test_build_chunk_matrix_detects_duplicate_keys():
    """build_chunk_matrix should reject duplicate (PA, DIM) combinations."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            if pa == "PA05" and dim == "DIM03":
                continue
            chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    chunks.append(create_chunk(idx, "PA01", "DIM01", "Duplicate chunk"))

    doc = PreprocessedDocument(
        document_id="test-doc-004",
        raw_text="Document with duplicate key",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError, match="Duplicate.*PA01-DIM01"):
        build_chunk_matrix(doc)


def test_build_chunk_matrix_detects_missing_combinations():
    """build_chunk_matrix should detect missing PAÃ\227DIM combinations."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            if pa == "PA05" and dim == "DIM03":
                continue
            chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    chunks.append(create_chunk(idx, "PA05", "DIM04"))

    doc = PreprocessedDocument(
        document_id="test-doc-005",
        raw_text="Document with duplicate and missing",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError, match="Duplicate.*PA05-DIM04"):
        build_chunk_matrix(doc)


def test_build_chunk_matrix_reports_specific_missing_key():
    """build_chunk_matrix should report specific missing PA-DIM combination."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            if pa == "PA05" and dim == "DIM03":
                continue
            if pa == "PA07" and dim == "DIM02":
                continue
            chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    doc = PreprocessedDocument(
        document_id="test-doc-006",
        raw_text="Document missing PA05-DIM03 and PA07-DIM02",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError) as exc_info:
        build_chunk_matrix(doc)
    assert (
        "Missing chunk combinations: [('PA05', 'DIM03'), ('PA07', 'DIM02')]"
        in str(exc_info.value)
    )


def test_build_chunk_matrix_rejects_null_policy_area():
    """build_chunk_matrix should reject chunks with null policy_area_id."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            if pa == "PA01" and dim == "DIM01":
                chunk = ChunkData(
                    id=idx,
                    text="Test",
                    chunk_type="diagnostic",
                    sentences=[],
                    tables=[],
                    start_pos=0,
                    end_pos=4,
                    confidence=0.9,
                    chunk_id="PA01-DIM01",
                    policy_area_id=None,
                    dimension_id=dim,
                )
                chunks.append(chunk)
            else:
                chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    doc = PreprocessedDocument(
        document_id="test-doc-007",
        raw_text="Document with null policy_area_id",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError, match="null policy_area_id"):
        build_chunk_matrix(doc)

```

```python

def test_build_chunk_matrix_rejects_null_dimension():
    """build_chunk_matrix should reject chunks with null dimension_id."""
    chunks = []
    idx = 0
    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            if pa == "PA01" and dim == "DIM01":
                chunk = ChunkData(
                    id=idx,
                    text="Test",
                    chunk_type="diagnostic",
                    sentences=[],
                    tables=[],
                    start_pos=0,
                    end_pos=4,
                    confidence=0.9,
                    chunk_id="PA01-DIM01",
                    policy_area_id=pa,
                    dimension_id=None,
                )
                chunks.append(chunk)
            else:
                chunks.append(create_chunk(idx, pa, dim))
            idx += 1

    doc = PreprocessedDocument(
        document_id="test-doc-008",
        raw_text="Document with null dimension_id",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
    )

    with pytest.raises(ValueError, match="null dimension_id"):
        build_chunk_matrix(doc)


def test_build_chunk_matrix_rejects_invalid_policy_area():
    """build_chunk_matrix should reject invalid policy area format."""
    with pytest.raises(ValueError, match="Invalid chunk_id ’PA00-DIM01’"):
        create_chunk(0, "PA00", "DIM01")


def test_build_chunk_matrix_rejects_invalid_dimension():
    """build_chunk_matrix should reject invalid dimension format."""
    with pytest.raises(ValueError, match="Invalid chunk_id ’PA01-DIM07’"):
        create_chunk(0, "PA01", "DIM07")


def test_build_chunk_matrix_rejects_malformed_chunk_id():
    """build_chunk_matrix should reject malformed chunk IDs."""
    with pytest.raises(ValueError, match="Invalid chunk_id ’P01-DIM01’"):
        create_chunk(0, "P01", "DIM01")

```

```

def test_build_chunk_matrix_preserves_content():
    """build_chunk_matrix should preserve chunk content and metadata."""
    doc = create_full_document()
    matrix, _ = build_chunk_matrix(doc)

    chunk = matrix[("PA07", "DIM04")]
    assert chunk.text == "Content for PA07-DIM04"
    assert chunk.policy_area_id == "PA07"
    assert chunk.dimension_id == "DIM04"
    assert chunk.chunk_type == "diagnostic"
    assert chunk.confidence == 0.9
    assert chunk.provenance is not None
    assert chunk.provenance.page_number == 1


def test_build_chunk_matrix_deterministic_ordering():
    """Multiple calls should produce identical key ordering."""
    doc = create_full_document()

    _, keys1 = build_chunk_matrix(doc)
    _, keys2 = build_chunk_matrix(doc)

    assert keys1 == keys2


def test_build_chunk_matrix_all_combinations_present():
    """Matrix should contain all 60 PAÃ\227DIM combinations."""
    doc = create_full_document()
    matrix, _ = build_chunk_matrix(doc)

    for pa in POLICY_AREAS:
        for dim in DIMENSIONS:
            assert (pa, dim) in matrix
            chunk = matrix[(pa, dim)]
            assert chunk.policy_area_id == pa
            assert chunk.dimension_id == dim



================================================================================
FILE: tests/core/orchestrator/test_contract_verification.py
================================================================================

"""Tests for pre-execution contract verification in BaseExecutorWithContract.

This test module validates the contract verification functionality that ensures
all 30 base executor contracts are valid before execution begins.
"""
from __future__ import annotations

import pytest
from pathlib import Path
from unittest.mock import MagicMock, patch

from farfan_pipeline.core.orchestrator.base_executor_with_contract import (
    BaseExecutorWithContract,
)


class TestContractVerification:
    """Test suite for contract verification functionality."""

    def test_verify_all_base_contracts_calls_verify_single(self):
        """Test that verify_all_base_contracts attempts to verify all 30 contracts."""
        with patch.object(
            BaseExecutorWithContract,
            "_verify_single_contract",
            return_value={
                "passed": True,
                "errors": [],
                "warnings": [],
                "contract_version": "v3",
                "contract_path": "/fake/path",
            },
        ) as mock_verify:
            result = BaseExecutorWithContract.verify_all_base_contracts(
                class_registry={}
            )

            assert mock_verify.call_count == 30
            assert result["total_contracts"] == 30
            assert result["passed"] is True
            assert len(result["verified_contracts"]) == 30

    def test_verify_all_base_contracts_with_errors(self):
        """Test that errors are accumulated correctly."""

        def mock_verify_single(base_slot, class_registry=None):
            if base_slot in ["D1-Q1", "D2-Q2"]:
                return {
                    "passed": False,
                    "errors": [f"Error in {base_slot}"],
                    "warnings": [],
                    "contract_version": "v3",
                    "contract_path": "/fake/path",
                }
            return {
                "passed": True,
                "errors": [],
                "warnings": [],
                "contract_version": "v3",
                "contract_path": "/fake/path",
            }

        with patch.object(
            BaseExecutorWithContract,
            "_verify_single_contract",
            side_effect=mock_verify_single,
        ):
            result = BaseExecutorWithContract.verify_all_base_contracts(
                class_registry={}
            )

            assert result["total_contracts"] == 30
            assert result["passed"] is False
            assert len(result["errors"]) == 2
            assert len(result["verified_contracts"]) == 28
            assert any("D1-Q1" in err for err in result["errors"])
            assert any("D2-Q2" in err for err in result["errors"])

    def test_verify_all_base_contracts_caches_result(self):
        """Test that subsequent calls use cached result."""
        BaseExecutorWithContract._factory_contracts_verified = False
        BaseExecutorWithContract._factory_verification_errors = []

        with patch.object(
            BaseExecutorWithContract,
            "_verify_single_contract",
            return_value={
                "passed": True,
                "errors": [],
                "warnings": [],
                "contract_version": "v3",
                "contract_path": "/fake/path",
            },
        ) as mock_verify:
            result1 = BaseExecutorWithContract.verify_all_base_contracts(
                class_registry={}
            )
            result2 = BaseExecutorWithContract.verify_all_base_contracts(
                class_registry={}
            )

            assert mock_verify.call_count == 30
            assert result1 == result2

    def test_verify_single_contract_missing_file(self):
        """Test verification fails when contract file doesn’t exist."""
        BaseExecutorWithContract._factory_contracts_verified = False

        result = BaseExecutorWithContract._verify_single_contract(
            "D99-Q99", class_registry={}
        )

        assert result["passed"] is False
        assert len(result["errors"]) > 0
        assert any("not found" in err.lower() for err in result["errors"])
        assert result["contract_path"] is None

    def test_verify_v2_contract_fields(self):
        """Test v2 contract field validation."""
        valid_contract = {
            "method_inputs": [
                {"class": "TestClass", "method": "test_method"}
            ],
            "assembly_rules": [],
            "validation_rules": [],
        }

        errors = BaseExecutorWithContract._verify_v2_contract_fields(
            valid_contract, "D1-Q1", class_registry={"TestClass": object}
        )
        assert len(errors) == 0

    def test_verify_v2_contract_missing_fields(self):
        """Test v2 contract validation catches missing fields."""
        invalid_contract = {
            "method_inputs": []
        }

        errors = BaseExecutorWithContract._verify_v2_contract_fields(
            invalid_contract, "D1-Q1", class_registry={}
        )

        assert len(errors) >= 2
        assert any("assembly_rules" in err for err in errors)
        assert any("validation_rules" in err for err in errors)

    def test_verify_v2_contract_method_class_not_in_registry(self):
        """Test v2 contract validation catches missing method classes."""
        contract = {
            "method_inputs": [
                {"class": "MissingClass", "method": "test_method"}
            ],
            "assembly_rules": [],
            "validation_rules": [],
        }

        errors = BaseExecutorWithContract._verify_v2_contract_fields(
            contract, "D1-Q1", class_registry={}
        )

        assert len(errors) >= 1
        assert any("MissingClass" in err and "not found" in err for err in errors)

    def test_verify_v3_contract_fields(self):
        """Test v3 contract field validation."""
        valid_contract = {
            "identity": {
                "base_slot": "D1-Q1"
            },
            "method_binding": {
                "orchestration_mode": "single_method",
                "class_name": "TestClass",
                "method_name": "test_method",
            },
            "evidence_assembly": {
                "assembly_rules": []
            },
            "validation_rules": {},
            "question_context": {
                "expected_elements": []
            },
            "error_handling": {},
        }

        errors = BaseExecutorWithContract._verify_v3_contract_fields(
            valid_contract, "D1-Q1", class_registry={"TestClass": object}
        )
        assert len(errors) == 0

    def test_verify_v3_contract_missing_identity(self):
        """Test v3 contract validation catches missing identity."""
        invalid_contract = {
            "method_binding": {},
            "evidence_assembly": {"assembly_rules": []},
            "validation_rules": {},
            "question_context": {"expected_elements": []},
            "error_handling": {},
        }

        errors = BaseExecutorWithContract._verify_v3_contract_fields(
            invalid_contract, "D1-Q1", class_registry={}
        )

        assert any("identity" in err.lower() for err in errors)

    def test_verify_v3_contract_base_slot_mismatch(self):
        """Test v3 contract validation catches base_slot mismatch."""
        contract = {
            "identity": {
                "base_slot": "D2-Q1"
            },
            "method_binding": {
                "class_name": "TestClass",
                "method_name": "test_method",
            },
            "evidence_assembly": {"assembly_rules": []},
            "validation_rules": {},
            "question_context": {"expected_elements": []},
            "error_handling": {},
        }

        errors = BaseExecutorWithContract._verify_v3_contract_fields(
            contract, "D1-Q1", class_registry={"TestClass": object}
        )

        assert any("mismatch" in err.lower() and "D1-Q1" in err for err in errors)

    def test_verify_v3_contract_multi_method_pipeline(self):
        """Test v3 contract validation for multi_method_pipeline mode."""
        contract = {
            "identity": {"base_slot": "D1-Q1"},
            "method_binding": {
                "orchestration_mode": "multi_method_pipeline",
                "methods": [
                    {"class_name": "Class1", "method_name": "method1"},
                    {"class_name": "Class2", "method_name": "method2"},
                ]
            },
            "evidence_assembly": {"assembly_rules": []},
            "validation_rules": {},
            "question_context": {"expected_elements": []},
            "error_handling": {},
        }

        errors = BaseExecutorWithContract._verify_v3_contract_fields(
            contract,
            "D1-Q1",
            class_registry={"Class1": object, "Class2": object}
        )

        assert len(errors) == 0

    def test_verify_v3_contract_multi_method_missing_class(self):
        """Test v3 multi-method validation catches missing classes."""
        contract = {
            "identity": {"base_slot": "D1-Q1"},
            "method_binding": {
                "orchestration_mode": "multi_method_pipeline",
                "methods": [
                    {"class_name": "Class1", "method_name": "method1"},
                    {"class_name": "MissingClass", "method_name": "method2"},
                ]
            },
            "evidence_assembly": {"assembly_rules": []},
            "validation_rules": {},
            "question_context": {"expected_elements": []},
            "error_handling": {},
        }

        errors = BaseExecutorWithContract._verify_v3_contract_fields(
            contract,
            "D1-Q1",
            class_registry={"Class1": object}
        )

        assert any("MissingClass" in err and "not found" in err for err in errors)

    def test_verify_v3_contract_missing_expected_elements(self):
        """Test v3 contract validation catches missing expected_elements."""
        contract = {
            "identity": {"base_slot": "D1-Q1"},
            "method_binding": {
                "class_name": "TestClass",
                "method_name": "test_method",
            },
            "evidence_assembly": {"assembly_rules": []},
            "validation_rules": {},
            "question_context": {},
            "error_handling": {},
        }

        errors = BaseExecutorWithContract._verify_v3_contract_fields(
            contract, "D1-Q1", class_registry={"TestClass": object}
        )

        assert any("expected_elements" in err for err in errors)

    def test_base_slot_to_q_number_calculation(self):
        """Test that base slot to Q number calculation is correct."""
        test_cases = [
            ("D1-Q1", "Q001"),
            ("D1-Q5", "Q005"),
            ("D2-Q1", "Q006"),
            ("D3-Q3", "Q013"),
            ("D6-Q5", "Q030"),
        ]

        for base_slot, expected_q_id in test_cases:
            dimension = int(base_slot[1])
            question = int(base_slot[4])
            q_number = (dimension - 1) * 5 + question
            q_id = f"Q{q_number:03d}"
            assert q_id == expected_q_id, f"Failed for {base_slot}: got {q_id}, expected {expected_q_id}"



================================================================================
FILE: tests/core/test_chunk_data_extensions.py
================================================================================

"""
Tests for ChunkData extended fields and validation.

Validates expected_elements schema expectations and document_position
byte offset tracking with comprehensive __post_init__ validation.
"""

import logging

import pytest

from farfan_pipeline.core.types import ChunkData


@pytest.fixture
def minimal_valid_chunk():
    """Minimal valid chunk with empty expected_elements and no document_position."""
    return ChunkData(
        id=100,
        text="Minimal chunk text",
        chunk_type="diagnostic",
        sentences=[],
        tables=[],
        start_pos=0,
        end_pos=50,
        confidence=0.9,
        policy_area_id="PA01",
        dimension_id="DIM01",
    )
```

```


@pytest.fixture
def chunk_with_position():
    """Chunk with document_position range."""
    return ChunkData(
        id=101,
        text="Chunk with position",
        chunk_type="activity",
        sentences=[0],
        tables=[],
        start_pos=0,
        end_pos=20,
        confidence=0.85,
        policy_area_id="PA02",
        dimension_id="DIM02",
        document_position=(1000, 2000),
    )


@pytest.fixture
def chunk_with_expected_elements():
    """Chunk with populated expected_elements schema."""
    return ChunkData(
        id=102,
        text="Chunk with schema expectations",
        chunk_type="indicator",
        sentences=[0, 1],
        tables=[],
        start_pos=0,
        end_pos=30,
        confidence=0.92,
        policy_area_id="PA03",
        dimension_id="DIM03",
        expected_elements=[
            {"type": "table", "required": True, "minimum": 1},
            {"type": "numeric_data", "required": False},
            {"type": "citation", "minimum": 0},
        ],
    )


@pytest.fixture
def fully_populated_chunk():
    """Chunk with both expected_elements and document_position."""
    return ChunkData(
        id=103,
        text="Fully populated chunk",
        chunk_type="resource",
        sentences=[0, 1, 2],
        tables=[0],
        start_pos=0,
        end_pos=25,
        confidence=0.95,
        policy_area_id="PA04",
        dimension_id="DIM04",
        expected_elements=[
            {"type": "budget_allocation", "required": True, "minimum": 1},
            {"type": "timeframe"},
        ],
        document_position=(5000, 5500),
    )


class TestChunkDataValidFields:
    """Test suite for valid ChunkData configurations."""

    def test_minimal_valid_chunk(self, minimal_valid_chunk):
        """Minimal chunk with empty fields should be valid."""
        assert minimal_valid_chunk.expected_elements == []
        assert minimal_valid_chunk.document_position is None
        assert minimal_valid_chunk.text == "Minimal chunk text"

    def test_chunk_with_position_range(self, chunk_with_position):
        """Chunk with valid position range should be accepted."""
        assert chunk_with_position.document_position == (1000, 2000)
        assert chunk_with_position.text == "Chunk with position"

    def test_chunk_with_expected_elements_schema(self, chunk_with_expected_elements):
        """Chunk with populated expected_elements should be valid."""
        assert len(chunk_with_expected_elements.expected_elements) == 3
        assert chunk_with_expected_elements.expected_elements[0]["type"] == "table"
        assert chunk_with_expected_elements.expected_elements[0]["required"] is True
        assert chunk_with_expected_elements.expected_elements[0]["minimum"] == 1
        assert (
            chunk_with_expected_elements.expected_elements[1]["type"] == "numeric_data"
        )
        assert chunk_with_expected_elements.expected_elements[1]["required"] is False
        assert chunk_with_expected_elements.expected_elements[2]["type"] == "citation"
        assert chunk_with_expected_elements.expected_elements[2]["minimum"] == 0

    def test_fully_populated_chunk_both_fields(self, fully_populated_chunk):
        """Chunk with both new fields populated should be valid."""
        assert len(fully_populated_chunk.expected_elements) == 2
        assert fully_populated_chunk.document_position == (5000, 5500)
        assert fully_populated_chunk.expected_elements[0]["type"] == "budget_allocation"
        assert fully_populated_chunk.expected_elements[1]["type"] == "timeframe"

    def test_zero_length_position_triggers_warning(self, caplog):
        """Zero-length position range should trigger warning log entry."""
        with caplog.at_level(logging.WARNING):
            chunk = ChunkData(
                id=999,
                text="Test zero-length position",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=26,
                confidence=0.8,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=(1500, 1500),
            )
        assert "zero-length document_position" in caplog.text
        assert chunk.document_position == (1500, 1500)


class TestChunkDataTextValidation:
    """Test suite for text field validation."""

    def test_empty_text_raises_error(self):
        """Empty text should raise ValueError."""
        with pytest.raises(ValueError, match="text cannot be empty"):
            ChunkData(
                id=1,
                text="",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
            )

    def test_whitespace_only_text_raises_error(self):
        """Whitespace-only text should raise ValueError."""
        with pytest.raises(ValueError, match="text cannot be empty"):
            ChunkData(
                id=1,
                text="   \n\t  ",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
            )


class TestChunkDataDocumentPositionValidation:
    """Test suite for document_position field validation."""

    def test_inverted_position_range_raises_error(self):
        """Position range with end < start should raise ValueError."""
        with pytest.raises(ValueError, match="end offset.*must be >= start offset"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=(2000, 1000),
            )

    def test_negative_start_offset_raises_error(self):
        """Negative start offset should raise ValueError."""
        with pytest.raises(ValueError, match="start offset must be non-negative"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=(-100, 1000),
            )

    def test_non_tuple_document_position_raises_error(self):
        """Non-tuple document_position should raise ValueError."""
        with pytest.raises(ValueError, match="document_position must be a tuple"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=[1000, 2000],
            )

    def test_wrong_arity_document_position_raises_error(self):
        """document_position with != 2 elements should raise ValueError."""
        with pytest.raises(ValueError, match="must have exactly 2 elements"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=(1000, 2000, 3000),
            )

    def test_non_integer_position_start_raises_error(self):
        """Non-integer start position element should raise ValueError."""
        with pytest.raises(ValueError, match="start.*must be an integer"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=("1000", 2000),
            )

    def test_non_integer_position_end_raises_error(self):
        """Non-integer end position element should raise ValueError."""
        with pytest.raises(ValueError, match="end.*must be an integer"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                document_position=(1000, 2000.5),
            )


class TestChunkDataExpectedElementsValidation:
    """Test suite for expected_elements field validation."""

    def test_non_list_expected_elements_raises_error(self):
        """Non-list expected_elements should raise ValueError."""
        with pytest.raises(ValueError, match="expected_elements must be a list"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements="not a list",
            )

    def test_non_dict_expected_element_raises_error(self):
        """Non-dict element in expected_elements should raise ValueError."""
        with pytest.raises(ValueError, match="must be a dict"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=["not a dict"],
            )

    def test_missing_type_key_raises_error(self):
        """expected_elements dict without 'type' key should raise ValueError."""
        with pytest.raises(ValueError, match="missing required 'type' key"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=[{"required": True}],
            )

    def test_non_string_type_value_raises_error(self):
        """Non-string 'type' value should raise ValueError."""
        with pytest.raises(ValueError, match=r"\['type'\] must be a string"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=[{"type": 123}],
            )

    def test_non_boolean_required_value_raises_error(self):
        """Non-boolean 'required' value should raise ValueError."""
        with pytest.raises(ValueError, match=r"\['required'\] must be a boolean"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=[{"type": "table", "required": "yes"}],
            )

    def test_non_integer_minimum_value_raises_error(self):
        """Non-integer 'minimum' value should raise ValueError."""
        with pytest.raises(ValueError, match=r"\['minimum'\] must be an integer"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=[{"type": "table", "minimum": 1.5}],
            )

    def test_negative_minimum_value_raises_error(self):
        """Negative 'minimum' value should raise ValueError."""
        with pytest.raises(ValueError, match=r"\['minimum'\] must be non-negative"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=[{"type": "table", "minimum": -1}],
            )


class TestChunkDataMultipleElementValidation:
    """Test validation across multiple expected_elements."""

    def test_multiple_valid_elements(self):
        """Multiple valid expected_elements should all be accepted."""
        chunk = ChunkData(
            id=200,
            text="Complex schema",
            chunk_type="indicator",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=15,
            confidence=0.9,
            policy_area_id="PA05",
            dimension_id="DIM05",
            expected_elements=[
                {"type": "table", "required": True, "minimum": 2},
                {"type": "figure", "required": False, "minimum": 0},
                {"type": "citation", "minimum": 1},
                {"type": "formula"},
            ],
        )
        assert len(chunk.expected_elements) == 4

    def test_error_in_second_element_reports_correct_index(self):
        """Error in second element should report index 1."""
        with pytest.raises(ValueError, match=r"expected_elements\[1\]"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
                expected_elements=[
                    {"type": "valid_type"},
                    {"type": 123},  # Invalid
                ],
            )


class TestChunkDataFieldInteraction:
    """Test interaction between new fields and existing validation."""

    def test_new_fields_with_invalid_chunk_id_still_fails(self):
        """Invalid chunk_id should still fail even with valid new fields."""
        with pytest.raises(ValueError, match="chunk_id"):
            ChunkData(
                id=1,
                text="Valid text",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA99",
                dimension_id="DIM99",
                expected_elements=[{"type": "table"}],
                document_position=(1000, 2000),
            )

    def test_new_fields_preserved_after_chunk_id_derivation(self):
        """New fields should be preserved when chunk_id is derived."""
        chunk = ChunkData(
            id=1,
            text="Valid text",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=10,
            confidence=0.9,
            policy_area_id="PA01",
            dimension_id="DIM01",
            expected_elements=[{"type": "table"}],
            document_position=(1000, 2000),
        )
        assert chunk.chunk_id == "PA01-DIM01"
        assert len(chunk.expected_elements) == 1
        assert chunk.document_position == (1000, 2000)



================================================================================
FILE: tests/core/test_chunk_matrix_builder.py
================================================================================

"""
Comprehensive test suite for ChunkMatrix validation and construction.

Tests cover:
- Deterministic ordering (multi-run stability)
- 60-chunk invariant enforcement
- Missing/duplicate chunk detection
- chunk_id validation (PA01-PA10, DIM01-DIM06 format)
- ValueError message verification
- Audit log structure validation
- Property-based tests using Hypothesis
"""

from datetime import datetime
from typing import Any

import pytest
from hypothesis import HealthCheck, given, settings
from hypothesis import strategies as st

from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
from farfan_pipeline.synchronization.irrigation_synchronizer import ChunkMatrix


def create_chunk(
    chunk_id: int,
    policy_area_id: str,
    dimension_id: str,
    text: str = "test content",
    chunk_type: str = "diagnostic",
) -> ChunkData:
    """Factory for creating test chunks with configurable attributes."""
    return ChunkData(
        id=chunk_id,
        text=text,
        chunk_type=chunk_type,  # type: ignore[arg-type]
        sentences=[],
        tables=[],
        start_pos=0,
        end_pos=len(text),
        confidence=0.95,
        policy_area_id=policy_area_id,
        dimension_id=dimension_id,
    )


def create_complete_document(chunk_order: str = "sequential") -> PreprocessedDocument:
    """Create a valid document with all 60 required chunks.

    Args:
        chunk_order: How to order chunks - 'sequential', 'reversed', or 'shuffled'
    """
    chunks = []
    chunk_id = 0
    for pa_num in range(1, 11):
        for dim_num in range(1, 7):
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"
            chunks.append(create_chunk(chunk_id, pa_id, dim_id))
            chunk_id += 1

    if chunk_order == "reversed":
        chunks = list(reversed(chunks))
    elif chunk_order == "shuffled":
        import random

        random.seed(42)
        random.shuffle(chunks)

    return PreprocessedDocument(
        document_id="test-doc",
        raw_text="test",
        sentences=[],
        tables=[],
        metadata={},
        chunks=chunks,
        ingested_at=datetime.now(),
    )


class TestChunkMatrixDeterministicOrdering:
    """Test deterministic ordering and multi-run stability."""

    def test_matrix_construction_is_deterministic_sequential_order(self) -> None:
        """ChunkMatrix construction should be deterministic across multiple runs with sequential input."""
        doc = create_complete_document("sequential")

        matrices = [ChunkMatrix(doc) for _ in range(5)]

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"

                chunks = [m.get_chunk(pa_id, dim_id) for m in matrices]
                first_chunk = chunks[0]

                for chunk in chunks[1:]:
                    assert chunk.id == first_chunk.id
                    assert chunk.text == first_chunk.text
                    assert chunk.policy_area_id == first_chunk.policy_area_id
                    assert chunk.dimension_id == first_chunk.dimension_id

    def test_matrix_construction_is_deterministic_reversed_order(self) -> None:
        """ChunkMatrix should produce same results regardless of input chunk ordering."""
        doc_sequential = create_complete_document("sequential")
        doc_reversed = create_complete_document("reversed")

        matrix_seq = ChunkMatrix(doc_sequential)
        matrix_rev = ChunkMatrix(doc_reversed)

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"

                chunk_seq = matrix_seq.get_chunk(pa_id, dim_id)
                chunk_rev = matrix_rev.get_chunk(pa_id, dim_id)

                assert chunk_seq.policy_area_id == chunk_rev.policy_area_id
                assert chunk_seq.dimension_id == chunk_rev.dimension_id

    def test_matrix_construction_is_deterministic_shuffled_order(self) -> None:
        """ChunkMatrix should handle shuffled input order deterministically."""
        doc_sequential = create_complete_document("sequential")
        doc_shuffled = create_complete_document("shuffled")

        matrix_seq = ChunkMatrix(doc_sequential)
        matrix_shuf = ChunkMatrix(doc_shuffled)

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"

                chunk_seq = matrix_seq.get_chunk(pa_id, dim_id)
                chunk_shuf = matrix_shuf.get_chunk(pa_id, dim_id)

                assert chunk_seq.policy_area_id == chunk_shuf.policy_area_id
                assert chunk_seq.dimension_id == chunk_shuf.dimension_id

    def test_multi_run_stability_with_identical_input(self) -> None:
        """Multiple ChunkMatrix constructions from same document should be identical."""
        doc = create_complete_document()

        runs = 10
        matrices = [ChunkMatrix(doc) for _ in range(runs)]

        test_keys = [("PA01", "DIM01"), ("PA05", "DIM03"), ("PA10", "DIM06")]

        for pa_id, dim_id in test_keys:
            chunks = [m.get_chunk(pa_id, dim_id) for m in matrices]
            first = chunks[0]

            for chunk in chunks[1:]:
                assert chunk.id == first.id
                assert chunk.policy_area_id == first.policy_area_id
                assert chunk.dimension_id == first.dimension_id


class TestSixtyChunkInvariantEnforcement:
    """Test strict enforcement of 60-chunk requirement."""

    def test_accepts_exactly_60_chunks(self) -> None:
        """ChunkMatrix should accept exactly 60 valid chunks."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        chunk = matrix.get_chunk("PA01", "DIM01")
        assert chunk.policy_area_id == "PA01"
        assert chunk.dimension_id == "DIM01"

    def test_rejects_59_chunks(self) -> None:
        """ChunkMatrix should reject documents with 59 chunks by detecting missing combination."""
        doc = create_complete_document()
        doc.chunks = doc.chunks[:59]

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Missing chunk combinations" in error_msg

    def test_rejects_61_chunks(self) -> None:
        """ChunkMatrix should reject documents with 61 chunks."""
        doc = create_complete_document()
        first_chunk = doc.chunks[0]
        assert first_chunk.policy_area_id is not None
        assert first_chunk.dimension_id is not None
        duplicate_chunk = create_chunk(
            60, first_chunk.policy_area_id, first_chunk.dimension_id
        )
        doc.chunks.append(duplicate_chunk)

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        assert "Duplicate (PA, DIM) combination detected" in str(exc_info.value)

    def test_rejects_0_chunks(self) -> None:
        """ChunkMatrix should reject documents with no chunks."""
        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=[],
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Missing chunk combinations" in error_msg

    def test_rejects_arbitrary_incorrect_count(self) -> None:
        """ChunkMatrix should reject any count other than 60."""
        for count in [1, 10, 30, 45, 58, 59, 62, 70]:
            if count > 60:
                continue
            doc = create_complete_document()
            doc.chunks = doc.chunks[:count]

            with pytest.raises(ValueError) as exc_info:
                ChunkMatrix(doc)

            error_msg = str(exc_info.value)
            assert (
                "Missing chunk combinations" in error_msg
                or "Chunk Matrix Invariant Violation" in error_msg
            )


class TestMissingChunkDetection:
    """Test detection and reporting of missing chunk combinations."""

    def test_detects_single_missing_chunk(self) -> None:
        """ChunkMatrix should detect and report a single missing PA-DIM combination."""
        chunks = []
        chunk_id = 0
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                if pa_num == 5 and dim_num == 3:
                    continue
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Missing chunk combinations" in error_msg
        assert "PA05', 'DIM03" in error_msg

    def test_detects_multiple_missing_chunks(self) -> None:
        """ChunkMatrix should detect and report multiple missing combinations."""
        chunks = []
        chunk_id = 0
        missing_keys = [("PA02", "DIM01"), ("PA07", "DIM04"), ("PA10", "DIM06")]

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                if (pa_id, dim_id) in missing_keys:
                    continue
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Missing chunk combinations" in error_msg
        for pa_id, dim_id in missing_keys:
            assert f"{pa_id}-{dim_id}" in error_msg

    def test_missing_chunks_reported_in_sorted_order(self) -> None:
        """Missing chunk combinations should be reported in sorted order."""
        chunks = []
        chunk_id = 0
        missing_keys = [("PA10", "DIM06"), ("PA01", "DIM01"), ("PA05", "DIM03")]

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                if (pa_id, dim_id) in missing_keys:
                    continue
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Missing chunk combinations" in error_msg

        missing_str = error_msg.split("Missing chunk combinations: ")[1]
        missing_items = missing_str.strip("[]")
        missing_list = [m.strip() for m in missing_items.split(", ")]
        assert missing_list == sorted(missing_list)


class TestDuplicateChunkDetection:
    """Test detection and reporting of duplicate chunk combinations."""

    def test_detects_duplicate_pa_dim_combination(self) -> None:
        """ChunkMatrix should detect duplicate PA-DIM combinations."""
        chunks = []
        chunks.append(create_chunk(0, "PA01", "DIM01"))
        chunks.append(create_chunk(1, "PA01", "DIM01"))

        chunk_id = 2
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                if pa_num == 1 and dim_num == 1:
                    continue
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Duplicate" in error_msg
        assert "PA01-DIM01" in error_msg

    def test_detects_duplicate_with_60_chunks(self) -> None:
        """ChunkMatrix should detect duplicates even when chunk count is 60."""
        doc = create_complete_document()
        doc.chunks[0] = create_chunk(0, "PA01", "DIM02")

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert (
            "Duplicate" in error_msg
            or "Missing chunk combinations" in error_msg
        )

    def test_detects_multiple_duplicates(self) -> None:
        """ChunkMatrix should detect first duplicate in sequence."""
        chunks = []
        chunks.append(create_chunk(0, "PA01", "DIM01"))
        chunks.append(create_chunk(1, "PA01", "DIM01"))
        chunks.append(create_chunk(2, "PA02", "DIM02"))
        chunks.append(create_chunk(3, "PA02", "DIM02"))

        chunk_id = 4
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                if (pa_num == 1 and dim_num == 1) or (pa_num == 2 and dim_num == 2):
                    continue
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Duplicate" in error_msg


class TestChunkIdValidation:
    """Test validation of chunk_id format (PA01-PA10, DIM01-DIM06)."""

    def test_accepts_valid_pa_range(self) -> None:
        """ChunkMatrix should accept all valid PA values (PA01-PA10)."""
        for pa_num in range(1, 11):
            pa_id = f"PA{pa_num:02d}"
            chunks = []
            chunk_id = 0

            for dim_num in range(1, 7):
                dim_id = f"DIM{dim_num:02d}"
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

            for other_pa_num in range(1, 11):
                if other_pa_num == pa_num:
                    continue
                other_pa_id = f"PA{other_pa_num:02d}"
                for dim_num in range(1, 7):
                    dim_id = f"DIM{dim_num:02d}"
                    chunks.append(create_chunk(chunk_id, other_pa_id, dim_id))
                    chunk_id += 1

            doc = PreprocessedDocument(
                document_id="test-doc",
                raw_text="test",
                sentences=[],
                tables=[],
                metadata={},
                chunks=chunks,
                ingested_at=datetime.now(),
            )

            matrix = ChunkMatrix(doc)
            assert matrix.get_chunk(pa_id, "DIM01").policy_area_id == pa_id

    def test_accepts_valid_dim_range(self) -> None:
        """ChunkMatrix should accept all valid DIM values (DIM01-DIM06)."""
        for dim_num in range(1, 7):
            dim_id = f"DIM{dim_num:02d}"
            doc = create_complete_document()
            matrix = ChunkMatrix(doc)

            for pa_num in range(1, 11):
                pa_id = f"PA{pa_num:02d}"
                assert matrix.get_chunk(pa_id, dim_id).dimension_id == dim_id

    def test_rejects_pa00(self) -> None:
        """ChunkData should reject PA00 at creation time."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, "PA00", "DIM01")

        error_msg = str(exc_info.value)
        assert "Invalid chunk_id" in error_msg
        assert "PA00-DIM01" in error_msg

    def test_rejects_pa11(self) -> None:
        """ChunkData should reject PA11 at creation time."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, "PA11", "DIM01")

        error_msg = str(exc_info.value)
        assert "Invalid chunk_id" in error_msg
        assert "PA11-DIM01" in error_msg

    def test_rejects_dim00(self) -> None:
        """ChunkData should reject DIM00 at creation time."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, "PA01", "DIM00")

        error_msg = str(exc_info.value)
        assert "Invalid chunk_id" in error_msg
        assert "PA01-DIM00" in error_msg

    def test_rejects_dim07(self) -> None:
        """ChunkData should reject DIM07 at creation time."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, "PA01", "DIM07")

        error_msg = str(exc_info.value)
        assert "Invalid chunk_id" in error_msg
        assert "PA01-DIM07" in error_msg

    def test_rejects_malformed_pa_format(self) -> None:
        """ChunkData should reject malformed PA identifiers at creation time."""
        invalid_formats = ["P01", "PA1", "PA001", "pa01", "XA01", "PA-01"]

        for invalid_pa in invalid_formats:
            with pytest.raises(ValueError) as exc_info:
                create_chunk(0, invalid_pa, "DIM01")

            error_msg = str(exc_info.value)
            assert "Invalid chunk_id" in error_msg

    def test_rejects_malformed_dim_format(self) -> None:
        """ChunkData should reject malformed DIM identifiers at creation time."""
        invalid_formats = ["D01", "DIM1", "DIM001", "dim01", "XIM01", "DIM-01"]

        for invalid_dim in invalid_formats:
            with pytest.raises(ValueError) as exc_info:
                create_chunk(0, "PA01", invalid_dim)

            error_msg = str(exc_info.value)
            assert "Invalid chunk_id" in error_msg


class TestNullMetadataValidation:
    """Test validation of null policy_area_id and dimension_id."""

    def test_rejects_null_policy_area_id(self) -> None:
        """ChunkData should reject chunks with null policy_area_id at creation time."""
        with pytest.raises(ValueError) as exc_info:
            ChunkData(
                id=0,
                text="test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=4,
                confidence=0.95,
                policy_area_id=None,
                dimension_id="DIM01",
            )

        error_msg = str(exc_info.value)
        assert "chunk_id is required" in error_msg

    def test_rejects_null_dimension_id(self) -> None:
        """ChunkData should reject chunks with null dimension_id at creation time."""
        with pytest.raises(ValueError) as exc_info:
            ChunkData(
                id=0,
                text="test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=4,
                confidence=0.95,
                policy_area_id="PA01",
                dimension_id=None,
            )

        error_msg = str(exc_info.value)
        assert "chunk_id is required" in error_msg

    def test_rejects_multiple_null_metadata(self) -> None:
        """ChunkData should detect missing chunk_id when both PA and DIM are null."""
        with pytest.raises(ValueError) as exc_info:
            ChunkData(
                id=0,
                text="test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=4,
                confidence=0.95,
                policy_area_id=None,
                dimension_id=None,
            )

        error_msg = str(exc_info.value)
        assert "chunk_id is required" in error_msg


class TestValueErrorMessages:
    """Test clarity and completeness of ValueError messages."""

    def test_missing_chunk_error_includes_expected_format(self) -> None:
        """Missing chunk error should explain expected format."""
        doc = create_complete_document()
        doc.chunks = doc.chunks[:59]

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "PA" in error_msg and "DIM" in error_msg

    def test_invalid_format_error_includes_expected_pattern(self) -> None:
        """Invalid format error should show expected PA{01-10}-DIM{01-06} pattern."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, "PA99", "DIM01")

        error_msg = str(exc_info.value)
        assert (
            "expected format PA{01-10}-DIM{01-06}" in error_msg.lower()
            or "invalid chunk_id" in error_msg.lower()
        )

    def test_duplicate_error_includes_specific_key(self) -> None:
        """Duplicate error should include the specific duplicate key."""
        chunks = [create_chunk(0, "PA05", "DIM03"), create_chunk(1, "PA05", "DIM03")]

        chunk_id = 2
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                if pa_num == 5 and dim_num == 3:
                    continue
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "PA05" in error_msg and "DIM03" in error_msg

    def test_null_metadata_error_message(self) -> None:
        """Null metadata error should have clear message."""
        with pytest.raises(ValueError) as exc_info:
            ChunkData(
                id=42,
                text="test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=4,
                confidence=0.95,
                policy_area_id=None,
                dimension_id="DIM01",
            )

        error_msg = str(exc_info.value)
        assert "chunk_id is required" in error_msg


class TestChunkMatrixAccess:
    """Test ChunkMatrix access patterns and guarantees."""

    def test_get_chunk_returns_correct_chunk(self) -> None:
        """get_chunk should return the chunk with matching PA and DIM."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        chunk = matrix.get_chunk("PA07", "DIM04")
        assert chunk.policy_area_id == "PA07"
        assert chunk.dimension_id == "DIM04"

    def test_get_chunk_all_60_combinations(self) -> None:
        """get_chunk should work for all 60 valid PA-DIM combinations."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunk = matrix.get_chunk(pa_id, dim_id)
                assert chunk.policy_area_id == pa_id
                assert chunk.dimension_id == dim_id

    def test_get_chunk_raises_keyerror_for_invalid_combination(self) -> None:
        """get_chunk should raise KeyError for invalid combinations."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        with pytest.raises(KeyError) as exc_info:
            matrix.get_chunk("PA11", "DIM01")

        error_msg = str(exc_info.value)
        assert "Chunk not found" in error_msg
        assert "PA11-DIM01" in error_msg

    def test_get_chunk_preserves_chunk_metadata(self) -> None:
        """get_chunk should preserve all chunk metadata."""
        doc = create_complete_document()
        doc.chunks[0] = ChunkData(
            id=999,
            text="specific content",
            chunk_type="activity",
            sentences=[1, 2, 3],
            tables=[0],
            start_pos=100,
            end_pos=200,
            confidence=0.87,
            policy_area_id="PA01",
            dimension_id="DIM01",
        )

        matrix = ChunkMatrix(doc)
        chunk = matrix.get_chunk("PA01", "DIM01")

        assert chunk.id == 999
        assert chunk.text == "specific content"
        assert chunk.chunk_type == "activity"
        assert chunk.sentences == [1, 2, 3]
        assert chunk.tables == [0]
        assert chunk.start_pos == 100
        assert chunk.end_pos == 200
        assert chunk.confidence == 0.87


@st.composite
def valid_chunk_id_strategy(draw: Any) -> int:  # type: ignore[misc]
    """Strategy for generating valid chunk IDs."""
    return draw(st.integers(min_value=0, max_value=10000))  # type: ignore[no-any-return]


@st.composite
def valid_pa_id_strategy(draw: Any) -> str:  # type: ignore[misc]
    """Strategy for generating valid PA identifiers (PA01-PA10)."""
    pa_num: int = draw(st.integers(min_value=1, max_value=10))
    return f"PA{pa_num:02d}"


@st.composite
def valid_dim_id_strategy(draw: Any) -> str:  # type: ignore[misc]
    """Strategy for generating valid DIM identifiers (DIM01-DIM06)."""
    dim_num: int = draw(st.integers(min_value=1, max_value=6))
    return f"DIM{dim_num:02d}"


@st.composite
def invalid_pa_id_strategy(draw: Any) -> str:  # type: ignore[misc]
    """Strategy for generating invalid PA identifiers."""
    return draw(  # type: ignore[no-any-return]
        st.one_of(
            st.just("PA00"),
            st.just("PA11"),
            st.just("PA99"),
            st.just("P01"),
            st.just("PA1"),
            st.just("pa01"),
        )
    )


@st.composite
def invalid_dim_id_strategy(draw: Any) -> str:  # type: ignore[misc]
    """Strategy for generating invalid DIM identifiers."""
    return draw(  # type: ignore[no-any-return]
        st.one_of(
            st.just("DIM00"),
            st.just("DIM07"),
            st.just("DIM99"),
            st.just("D01"),
            st.just("DIM1"),
            st.just("dim01"),
        )
    )


class TestChunkMatrixPropertyBased:
    """Property-based tests using Hypothesis for ChunkMatrix validation."""

    @given(chunk_order=st.sampled_from(["sequential", "reversed", "shuffled"]))
    @settings(max_examples=20, deadline=2000)
    def test_property_ordering_invariance(self, chunk_order: str) -> None:  # type: ignore[misc]
        """Property: ChunkMatrix construction is invariant to input chunk ordering."""
        doc = create_complete_document(chunk_order)
        matrix = ChunkMatrix(doc)

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunk = matrix.get_chunk(pa_id, dim_id)
                assert chunk.policy_area_id == pa_id
                assert chunk.dimension_id == dim_id

    @given(
        pa_id=valid_pa_id_strategy(),
        dim_id=valid_dim_id_strategy(),
    )
    @settings(max_examples=50, deadline=1000)
    def test_property_valid_ids_accepted(self, pa_id: str, dim_id: str) -> None:  # type: ignore[misc]
        """Property: All valid PA-DIM combinations are accepted."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        chunk = matrix.get_chunk(pa_id, dim_id)
        assert chunk.policy_area_id == pa_id
        assert chunk.dimension_id == dim_id

    @given(invalid_pa=invalid_pa_id_strategy())
    @settings(
        max_examples=20,
        deadline=1000,
        suppress_health_check=[HealthCheck.filter_too_much],
    )
    def test_property_invalid_pa_rejected(self, invalid_pa: str) -> None:  # type: ignore[misc]
        """Property: Invalid PA identifiers are rejected at ChunkData creation."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, invalid_pa, "DIM01")

        assert "Invalid chunk_id" in str(exc_info.value)

    @given(invalid_dim=invalid_dim_id_strategy())
    @settings(
        max_examples=20,
        deadline=1000,
        suppress_health_check=[HealthCheck.filter_too_much],
    )
    def test_property_invalid_dim_rejected(self, invalid_dim: str) -> None:  # type: ignore[misc]
        """Property: Invalid DIM identifiers are rejected at ChunkData creation."""
        with pytest.raises(ValueError) as exc_info:
            create_chunk(0, "PA01", invalid_dim)

        assert "Invalid chunk_id" in str(exc_info.value)

    @given(
        pa_num=st.integers(min_value=1, max_value=10),
        dim_num=st.integers(min_value=1, max_value=6),
    )
    @settings(max_examples=60, deadline=1000)
    def test_property_all_combinations_accessible(  # type: ignore[misc]
        self, pa_num: int, dim_num: int
    ) -> None:
        """Property: All 60 PA-DIM combinations are accessible via get_chunk."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        pa_id = f"PA{pa_num:02d}"
        dim_id = f"DIM{dim_num:02d}"

        chunk = matrix.get_chunk(pa_id, dim_id)
        assert chunk is not None
        assert chunk.policy_area_id == pa_id
        assert chunk.dimension_id == dim_id

    @given(
        missing_pa=st.integers(min_value=1, max_value=10),
        missing_dim=st.integers(min_value=1, max_value=6),
    )
    @settings(max_examples=30, deadline=1000)
    def test_property_missing_chunks_detected(  # type: ignore[misc]
        self, missing_pa: int, missing_dim: int
    ) -> None:
        """Property: Missing any PA-DIM combination is detected."""
        chunks = []
        chunk_id = 0
        missing_pa_id = f"PA{missing_pa:02d}"
        missing_dim_id = f"DIM{missing_dim:02d}"

        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                if pa_id == missing_pa_id and dim_id == missing_dim_id:
                    continue
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Missing chunk combinations" in error_msg
        assert f"{missing_pa_id}-{missing_dim_id}" in error_msg

    @given(
        dup_pa=st.integers(min_value=1, max_value=10),
        dup_dim=st.integers(min_value=1, max_value=6),
    )
    @settings(max_examples=30, deadline=1000)
    def test_property_duplicates_detected(self, dup_pa: int, dup_dim: int) -> None:  # type: ignore[misc]
        """Property: Duplicate PA-DIM combinations are always detected."""
        dup_pa_id = f"PA{dup_pa:02d}"
        dup_dim_id = f"DIM{dup_dim:02d}"

        chunks = [
            create_chunk(0, dup_pa_id, dup_dim_id),
            create_chunk(1, dup_pa_id, dup_dim_id),
        ]

        chunk_id = 2
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                if pa_id == dup_pa_id and dim_id == dup_dim_id:
                    continue
                chunks.append(create_chunk(chunk_id, pa_id, dim_id))
                chunk_id += 1

        doc = PreprocessedDocument(
            document_id="test-doc",
            raw_text="test",
            sentences=[],
            tables=[],
            metadata={},
            chunks=chunks,
            ingested_at=datetime.now(),
        )

        with pytest.raises(ValueError) as exc_info:
            ChunkMatrix(doc)

        error_msg = str(exc_info.value)
        assert "Duplicate" in error_msg
        assert dup_pa_id in error_msg and dup_dim_id in error_msg

    @given(text=st.text(min_size=1, max_size=1000))
    @settings(max_examples=30, deadline=1000)
    def test_property_chunk_content_preserved(self, text: str) -> None:  # type: ignore[misc]
        """Property: Chunk text content is preserved through matrix construction."""
        doc = create_complete_document()
        doc.chunks[0] = create_chunk(0, "PA01", "DIM01", text=text)

        matrix = ChunkMatrix(doc)
        chunk = matrix.get_chunk("PA01", "DIM01")

        assert chunk.text == text


class TestChunkMatrixAuditLog:
    """Test audit log structure and validation metadata."""

    def test_matrix_internal_structure_is_dict(self) -> None:
        """ChunkMatrix internal structure should be accessible as dict."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        assert hasattr(matrix, "chunks")
        assert isinstance(matrix.chunks, dict)

    def test_matrix_stores_all_60_chunks(self) -> None:
        """ChunkMatrix should store exactly 60 chunks internally."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        assert len(matrix.chunks) == ChunkMatrix.EXPECTED_CHUNK_COUNT

    def test_matrix_keys_are_tuples(self) -> None:
        """ChunkMatrix keys should be (policy_area_id, dimension_id) tuples."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        for key in matrix.chunks.keys():
            assert isinstance(key, tuple)
            assert len(key) == 2
            assert isinstance(key[0], str)
            assert isinstance(key[1], str)

    def test_matrix_values_are_chunk_data(self) -> None:
        """ChunkMatrix values should be ChunkData instances."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        for value in matrix.chunks.values():
            assert isinstance(value, ChunkData)

    def test_matrix_contains_all_expected_keys(self) -> None:
        """ChunkMatrix should contain all 60 expected PA-DIM combinations as keys."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        expected_keys = {(pa, dim) for pa in ChunkMatrix.POLICY_AREAS for dim in ChunkMatrix.DIMENSIONS}

        assert set(matrix.chunks.keys()) == expected_keys

    def test_matrix_key_to_chunk_mapping_is_correct(self) -> None:
        """ChunkMatrix keys should correctly map to their corresponding chunks."""
        doc = create_complete_document()
        matrix = ChunkMatrix(doc)

        for (pa_id, dim_id), chunk in matrix.chunks.items():
            assert chunk.policy_area_id == pa_id
            assert chunk.dimension_id == dim_id


class TestChunkMatrixConstants:
    """Test ChunkMatrix class constants and configuration."""

    def test_policy_areas_constant(self) -> None:
        """ChunkMatrix.POLICY_AREAS should contain PA01-PA10."""
        expected = ChunkMatrix.POLICY_AREAS
        assert expected == ChunkMatrix.POLICY_AREAS

    def test_dimensions_constant(self) -> None:
        """ChunkMatrix.DIMENSIONS should contain DIM01-DIM06."""
        expected = ChunkMatrix.DIMENSIONS
        assert expected == ChunkMatrix.DIMENSIONS

    def test_expected_chunk_count_constant(self) -> None:
        """ChunkMatrix.EXPECTED_CHUNK_COUNT should match PAÃ\227DIM cardinality."""
        assert ChunkMatrix.EXPECTED_CHUNK_COUNT == len(ChunkMatrix.POLICY_AREAS) * len(
            ChunkMatrix.DIMENSIONS
        )

    def test_chunk_id_pattern_validates_correctly(self) -> None:
        """ChunkMatrix.CHUNK_ID_PATTERN should validate correct formats."""
        pattern = ChunkMatrix.CHUNK_ID_PATTERN

        valid_ids = [
            f"{pa}-DIM{dim[-2:]}" if dim.startswith("DIM") else f"{pa}-{dim}"
            for pa in ChunkMatrix.POLICY_AREAS
            for dim in ChunkMatrix.DIMENSIONS
        ]

        for chunk_id in valid_ids:
            assert pattern.match(chunk_id), f"Valid ID rejected: {chunk_id}"

    def test_chunk_id_pattern_rejects_invalid(self) -> None:
        """ChunkMatrix.CHUNK_ID_PATTERN should reject invalid formats."""
        pattern = ChunkMatrix.CHUNK_ID_PATTERN

        invalid_ids = [
            "PA0-DIM01",
            "PA001-DIM01",
            "P01-DIM01",
            "PA01-DIM1",
            "PA01-DIM001",
            "pa01-dim01",
            "PA01_DIM01",
            "PA01-DIM0A",
            "XX01-DIM01",
        ]

        for chunk_id in invalid_ids:
            assert not pattern.match(chunk_id), f"Invalid ID accepted: {chunk_id}"


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



================================================================================
FILE: tests/core/test_chunk_router.py
================================================================================

"""
Tests for chunk_router.py - Routing logic formalization and contracts.

Validates the ExecutionMap contract, serialization/deserialization,
and deterministic behavior of routing logic.
"""

import hashlib
import json

import pytest
from pydantic import ValidationError

pytestmark = pytest.mark.obsolete

from farfan_pipeline.core.orchestrator.chunk_router import (
    ChunkRouter,
    ExecutionMap,
    deserialize_execution_map,
    serialize_execution_map,
)
from farfan_pipeline.core.types import ChunkData


@pytest.fixture
def sample_chunks():
    """Create sample chunks with policy area and dimension assignments."""
    return [
        ChunkData(
            id=1,
            text="Diagnostic text about women’s rights",
            chunk_type="diagnostic",
            sentences=[0, 1, 2],
            tables=[],
            start_pos=0,
            end_pos=100,
            confidence=0.95,
            policy_area_id="PA01",
            dimension_id="DIM01",
        ),
        ChunkData(
            id=2,
            text="Activity description for violence prevention",
            chunk_type="activity",
            sentences=[3, 4, 5],
            tables=[],
            start_pos=101,
            end_pos=200,
            confidence=0.92,
            policy_area_id="PA02",
            dimension_id="DIM02",
        ),
        ChunkData(
            id=3,
            text="Indicator for environmental metrics",
            chunk_type="indicator",
            sentences=[6, 7],
            tables=[],
            start_pos=201,
            end_pos=300,
            confidence=0.88,
            policy_area_id="PA03",
            dimension_id="DIM03",
        ),
    ]


@pytest.fixture
def router():
    """Create a ChunkRouter instance."""
    return ChunkRouter()


class TestExecutionMapContract:
    """Test suite for ExecutionMap contract validation."""

    def test_valid_execution_map(self):
        """Valid ExecutionMap should be created successfully."""
        routing_rules = {
            "PA01:DIM01": "D1Q1",
            "PA02:DIM02": "D2Q1",
        }
        canonical_repr = json.dumps(
            routing_rules, sort_keys=True, separators=(",", ":")
        )
        map_hash = hashlib.sha256(canonical_repr.encode("utf-8")).hexdigest()

        execution_map = ExecutionMap(
            version="1.0.0",
            map_hash=map_hash,
            routing_rules=routing_rules,
        )

        assert execution_map.version == "1.0.0"
        assert execution_map.map_hash == map_hash
        assert execution_map.routing_rules == routing_rules

    def test_invalid_version_format(self):
        """Version must match semantic versioning format."""
        with pytest.raises(ValidationError, match="version"):
            ExecutionMap(
                version="v1",  # Invalid format
                map_hash="a" * 64,
                routing_rules={"PA01:DIM01": "D1Q1"},
            )

    def test_invalid_hash_length(self):
        """map_hash must be exactly 64 characters."""
        with pytest.raises(ValidationError, match="map_hash"):
            ExecutionMap(
                version="1.0.0",
                map_hash="abc123",  # Too short
                routing_rules={"PA01:DIM01": "D1Q1"},
            )

    def test_invalid_hash_format(self):
        """map_hash must be valid hexadecimal."""
        with pytest.raises(ValidationError, match="hexadecimal"):
            ExecutionMap(
                version="1.0.0",
                map_hash="z" * 64,  # Invalid hex characters
                routing_rules={"PA01:DIM01": "D1Q1"},
            )

    def test_empty_routing_rules(self):
        """routing_rules cannot be empty."""
        with pytest.raises(ValidationError, match="routing_rules cannot be empty"):
            ExecutionMap(
                version="1.0.0",
                map_hash="a" * 64,
                routing_rules={},
            )

    def test_invalid_routing_key_format(self):
        """Routing keys must be in 'policy_area_id:dimension_id' format."""
        with pytest.raises(ValidationError, match="policy_area_id:dimension_id"):
            ExecutionMap(
                version="1.0.0",
                map_hash="a" * 64,
                routing_rules={"INVALID_KEY": "D1Q1"},
            )

    def test_invalid_routing_key_missing_parts(self):
        """Routing keys must have both policy_area_id and dimension_id."""
        with pytest.raises(ValidationError, match="non-empty"):
            ExecutionMap(
                version="1.0.0",
                map_hash="a" * 64,
                routing_rules={"PA01:": "D1Q1"},  # Missing dimension_id
            )

    def test_empty_executor_class(self):
        """Executor class must be non-empty string."""
        with pytest.raises(ValidationError, match="non-empty string"):
            ExecutionMap(
                version="1.0.0",
                map_hash="a" * 64,
                routing_rules={"PA01:DIM01": ""},
            )

    def test_get_executor(self):
        """get_executor should retrieve correct executor for given area/dimension."""
        execution_map = ExecutionMap(
            version="1.0.0",
            map_hash="a" * 64,
            routing_rules={
                "PA01:DIM01": "D1Q1",
                "PA02:DIM02": "D2Q1",
            },
        )

        assert execution_map.get_executor("PA01", "DIM01") == "D1Q1"
        assert execution_map.get_executor("PA02", "DIM02") == "D2Q1"
        assert execution_map.get_executor("PA99", "DIM99") is None


class TestGenerateExecutionMap:
    """Test suite for generate_execution_map functionality."""

    def test_generate_execution_map_basic(self, router, sample_chunks):
        """Generate execution map from valid chunks."""
        execution_map = router.generate_execution_map(sample_chunks)

        assert execution_map.version == "1.0.0"
        expected_rules_count = 3
        assert len(execution_map.routing_rules) == expected_rules_count
        assert "PA01:DIM01" in execution_map.routing_rules
        assert "PA02:DIM02" in execution_map.routing_rules
        assert "PA03:DIM03" in execution_map.routing_rules

        assert execution_map.routing_rules["PA01:DIM01"] == "D1Q1"
        assert execution_map.routing_rules["PA02:DIM02"] == "D2Q1"
        assert execution_map.routing_rules["PA03:DIM03"] == "D3Q1"

    def test_generate_execution_map_deterministic(self, router, sample_chunks):
        """Execution map generation must be deterministic."""
        map1 = router.generate_execution_map(sample_chunks)
        map2 = router.generate_execution_map(sample_chunks)

        assert map1.version == map2.version
        assert map1.map_hash == map2.map_hash
        assert map1.routing_rules == map2.routing_rules

    def test_generate_execution_map_order_independent(self, router, sample_chunks):
        """Execution map should be same regardless of chunk order."""
        import random

        chunks_copy = sample_chunks.copy()
        random.shuffle(chunks_copy)

        map1 = router.generate_execution_map(sample_chunks)
        map2 = router.generate_execution_map(chunks_copy)

        assert map1.map_hash == map2.map_hash
        assert map1.routing_rules == map2.routing_rules

    def test_generate_execution_map_missing_policy_area(self, router):
        """Should raise error if chunk missing policy_area_id."""
        chunks = [
            ChunkData(
                id=1,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id=None,  # Missing
                dimension_id="DIM01",
            )
        ]

        with pytest.raises(ValueError, match="policy_area_id"):
            router.generate_execution_map(chunks)

    def test_generate_execution_map_missing_dimension(self, router):
        """Should raise error if chunk missing dimension_id."""
        chunks = [
            ChunkData(
                id=1,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id=None,  # Missing
            )
        ]

        with pytest.raises(ValueError, match="dimension_id"):
            router.generate_execution_map(chunks)

    def test_generate_execution_map_unknown_chunk_type(self, router):
        """Unknown chunk types should be marked as UNROUTED."""
        chunks = [
            ChunkData(
                id=1,
                text="Test",
                chunk_type="unknown_type",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
            )
        ]

        execution_map = router.generate_execution_map(chunks)
        assert execution_map.routing_rules["PA01:DIM01"] == "UNROUTED_UNKNOWN_TYPE"

    def test_hash_verification(self, router, sample_chunks):
        """Generated hash should match computed hash of routing rules."""
        execution_map = router.generate_execution_map(sample_chunks)

        canonical_repr = json.dumps(
            dict(sorted(execution_map.routing_rules.items())),
            sort_keys=True,
            separators=(",", ":"),
        )
        expected_hash = hashlib.sha256(canonical_repr.encode("utf-8")).hexdigest()

        assert execution_map.map_hash == expected_hash

    def test_custom_version(self, router, sample_chunks):
        """Should accept custom version string."""
        execution_map = router.generate_execution_map(sample_chunks, version="2.1.5")
        assert execution_map.version == "2.1.5"


class TestSerialization:
    """Test suite for serialization and deserialization."""

    def test_serialize_execution_map(self, router, sample_chunks):
        """Serialize ExecutionMap to JSON string."""
        execution_map = router.generate_execution_map(sample_chunks)
        serialized = serialize_execution_map(execution_map)

        assert isinstance(serialized, str)
        data = json.loads(serialized)

        assert data["version"] == execution_map.version
        assert data["map_hash"] == execution_map.map_hash
        assert data["routing_rules"] == execution_map.routing_rules

    def test_deserialize_execution_map(self, router, sample_chunks):
        """Deserialize JSON string back to ExecutionMap."""
        execution_map = router.generate_execution_map(sample_chunks)
        serialized = serialize_execution_map(execution_map)
        deserialized = deserialize_execution_map(serialized)

        assert deserialized.version == execution_map.version
        assert deserialized.map_hash == execution_map.map_hash
        assert deserialized.routing_rules == execution_map.routing_rules

    def test_roundtrip_serialization(self, router, sample_chunks):
        """Roundtrip serialization should preserve data exactly."""
        original = router.generate_execution_map(sample_chunks)
        serialized = serialize_execution_map(original)
        deserialized = deserialize_execution_map(serialized)
        reserialized = serialize_execution_map(deserialized)

        assert serialized == reserialized
        assert original.version == deserialized.version
        assert original.map_hash == deserialized.map_hash
        assert original.routing_rules == deserialized.routing_rules

    def test_deserialize_invalid_json(self):
        """Should raise error for invalid JSON."""
        with pytest.raises(ValueError, match="Invalid JSON"):
            deserialize_execution_map("not valid json")

    def test_deserialize_missing_fields(self):
        """Should raise validation error for missing required fields."""
        incomplete_json = json.dumps({"version": "1.0.0"})

        with pytest.raises(ValidationError):
            deserialize_execution_map(incomplete_json)

    def test_serialization_deterministic(self, router, sample_chunks):
        """Serialization must be deterministic."""
        execution_map = router.generate_execution_map(sample_chunks)

        serialized1 = serialize_execution_map(execution_map)
        serialized2 = serialize_execution_map(execution_map)

        assert serialized1 == serialized2


class TestChunkRouterCompatibility:
    """Test suite for backward compatibility with existing ChunkRouter methods."""

    def test_route_chunk_backward_compat(self, router, sample_chunks):
        """route_chunk should continue to work as before."""
        chunk = sample_chunks[0]
        route = router.route_chunk(chunk)

        assert route.chunk_id == chunk.id
        assert route.chunk_type == chunk.chunk_type
        assert route.executor_class == "D1Q1"
        assert route.skip_reason is None

    def test_should_use_full_graph(self, router):
        """should_use_full_graph should identify graph methods correctly."""
        assert router.should_use_full_graph("construir_grafo_causal", "TeoriaCambio")
        assert router.should_use_full_graph(
            "extract_causal_hierarchy", "CausalExtractor"
        )
        assert not router.should_use_full_graph("some_regular_method")

    def test_get_relevant_executors(self, router):
        """get_relevant_executors should return executor list for chunk type."""
        assert router.get_relevant_executors("diagnostic") == ["D1Q1", "D1Q2", "D1Q5"]
        assert router.get_relevant_executors("activity") == [
            "D2Q1",
            "D2Q2",
            "D2Q3",
            "D2Q4",
            "D2Q5",
        ]
        assert router.get_relevant_executors("unknown") == []


class TestDeterminism:
    """Property-based tests for determinism guarantees."""

    def test_same_input_same_output(self, router):
        """Multiple calls with identical input must produce identical output."""
        chunks = [
            ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[i],
                tables=[],
                start_pos=i * 100,
                end_pos=(i + 1) * 100,
                confidence=0.9,
                policy_area_id=f"PA{i:02d}",
                dimension_id=f"DIM{(i % 6) + 1:02d}",
            )
            for i in range(1, 11)
        ]

        results = [router.generate_execution_map(chunks) for _ in range(10)]

        first_hash = results[0].map_hash
        first_rules = results[0].routing_rules

        for result in results[1:]:
            assert result.map_hash == first_hash
            assert result.routing_rules == first_rules

    def test_hash_collision_resistance(self, router):
        """Different routing rules should produce different hashes."""
        chunks1 = [
            ChunkData(
                id=1,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="DIM01",
            )
        ]

        chunks2 = [
            ChunkData(
                id=1,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA02",
                dimension_id="DIM01",
            )
        ]

        map1 = router.generate_execution_map(chunks1)
        map2 = router.generate_execution_map(chunks2)

        assert map1.map_hash != map2.map_hash
        assert map1.routing_rules != map2.routing_rules
```