

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 24
3: =====
4: Generated: 2025-12-07T06:17:30.363789
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: tests/calibration_system/__init__.py
11: =====
12:
13: """
14: Calibration System Validation Suite
15:
16: Comprehensive all-or-nothing quality gate for the calibration system.
17: Tests verify inventory consistency, layer correctness, intrinsic coverage,
18: AST extraction accuracy, orchestrator runtime, hardcoded values, and performance.
19:
20: ALL tests must pass for the system to be considered production-ready.
21: """
22:
23:
24:
25: =====
26: FILE: tests/calibration_system/generate_failure_report.py
27: =====
28:
29: #!/usr/bin/env python3
30: """
31: Generate Calibration System Failure Report
32:
33: Aggregates test results and generates a comprehensive failure report
34: when any validation test fails.
35: """
36: import argparse
37: import json
38: from datetime import datetime
39: from pathlib import Path
40: from typing import Dict, List
41:
42:
43: def parse_args() -> argparse.Namespace:
44:     """Parse command line arguments"""
45:     parser = argparse.ArgumentParser(
46:         description="Generate calibration system failure report"
47:     )
48:     parser.add_argument("--test1", required=True, help="Test 1 outcome")
49:     parser.add_argument("--test2", required=True, help="Test 2 outcome")
50:     parser.add_argument("--test3", required=True, help="Test 3 outcome")
51:     parser.add_argument("--test4", required=True, help="Test 4 outcome")
52:     parser.add_argument("--test5", required=True, help="Test 5 outcome")
53:     parser.add_argument("--test6", required=True, help="Test 6 outcome")
54:     parser.add_argument("--test7", required=True, help="Test 7 outcome")
55:     parser.add_argument("--output", default="calibration_system_failure_report.md",
56:                         help="Output file path")
```

```
57:     return parser.parse_args()
58:
59:
60: def load_test_results() -> Dict[str, List[str]]:
61:     """Load detailed test results from XML files"""
62:     results_dir = Path("test-results")
63:     failures = {}
64:
65:     if not results_dir.exists():
66:         return failures
67:
68:     for xml_file in results_dir.glob("*.xml"):
69:         try:
70:             import xml.etree.ElementTree as ET
71:             tree = ET.parse(xml_file)
72:             root = tree.getroot()
73:
74:             test_name = xml_file.stem
75:             test_failures = []
76:
77:             for testcase in root.findall("./testcase"):
78:                 failure = testcase.find("failure")
79:                 if failure is not None:
80:                     test_failures.append({
81:                         "name": testcase.get("name"),
82:                         "message": failure.get("message", ""),
83:                         "details": failure.text or ""
84:                     })
85:
86:                 if test_failures:
87:                     failures[test_name] = test_failures
88:
89:             except Exception as e:
90:                 print(f"Warning: Could not parse {xml_file}: {e}")
91:
92:     return failures
93:
94:
95: def generate_report(args: argparse.Namespace) -> str:
96:     """Generate the failure report"""
97:     tests = {
98:         "Test 1: Inventory Consistency": args.test1,
99:         "Test 2: Layer Correctness": args.test2,
100:        "Test 3: Intrinsic Coverage (≥ 80)": args.test3,
101:        "Test 4: AST Extraction Accuracy": args.test4,
102:        "Test 5: Orchestrator Runtime": args.test5,
103:        "Test 6: No Hardcoded Calibrations": args.test6,
104:        "Test 7: Performance Benchmarks": args.test7,
105:    }
106:
107:    failures = load_test_results()
108:    failed_tests = [name for name, outcome in tests.items() if outcome != "success"]
109:
110:    report = []
111:    report.append("# CALIBRATION SYSTEM FAILURE REPORT")
112:    report.append("")
```

```
113:     report.append(f"**Generated:** {datetime.now().isoformat()}")
114:     report.append("")
115:
116:     if not failed_tests:
117:         report.append("## \u234\u205 STATUS: PRODUCTION READY")
118:         report.append("")
119:         report.append("All calibration validation tests passed successfully.")
120:         report.append("")
121:         report.append("### Test Results")
122:         report.append("")
123:         for test_name, outcome in tests.items():
124:             report.append(f"- \u234\u205 {test_name}: **{outcome}**")
125:
126:     else:
127:         report.append("## \u235\u214 STATUS: NOT READY FOR PRODUCTION")
128:         report.append("")
129:         report.append(f"**{len(failed_tests)} of {len(tests)} validation tests FAILED**")
130:         report.append("")
131:         report.append("### Critical Failures")
132:         report.append("")
133:
134:         for test_name, outcome in tests.items():
135:             icon = "\u234\u205" if outcome == "success" else "\u235\u214"
136:             report.append(f"({icon}) **{test_name}**: {outcome}")
137:
138:         report.append("")
139:         report.append("---")
140:         report.append("")
141:         report.append("## Failure Analysis")
142:         report.append("")
143:
144:         test_descriptions = {
145:             "Test 1": {
146:                 "name": "Inventory Consistency",
147:                 "purpose": "Verifies all methods are consistently defined across JSON files",
148:                 "impact": "Inconsistent inventories can cause runtime errors and calibration mismatches",
149:                 "remediation": [
150:                     "Ensure all methods in executors_methods.json exist in intrinsic_calibration.json",
151:                     "Remove orphaned calibration entries",
152:                     "Verify all 30 executors are properly defined"
153:                 ]
154:             },
155:             "Test 2": {
156:                 "name": "Layer Correctness",
157:                 "purpose": "Validates architectural integrity with 8 required layers",
158:                 "impact": "Missing layers break the execution pipeline",
159:                 "remediation": [
160:                     "Each executor must have methods in all 8 layers: ingestion, extraction, transformation, validation, aggregation, scoring, reporting, me
ta",
161:                     "Verify LAYER_REQUIREMENTS mapping is complete",
162:                     "Check layer dependencies are acyclic"
163:                 ]
164:             },
165:             "Test 3": {
166:                 "name": "Intrinsic Coverage",
167:                 "purpose": "Ensures \u211Y80% methods have computed calibrations",
```

```
168:         "impact": "Low coverage means system relies on defaults or fails",
169:         "remediation": [
170:             "Run intrinsic calibration triage on uncalibrated methods",
171:             "Ensure all 30 executors have at least one method with status='computed'",
172:             "Document reasons for any 'excluded' or 'manual' methods"
173:         ],
174:     },
175:     "Test 4": {
176:         "name": "AST Extraction Accuracy",
177:         "purpose": "Validates extracted signatures match actual source code",
178:         "impact": "Signature mismatches cause method execution failures",
179:         "remediation": [
180:             "Re-extract method signatures from source code",
181:             "Update method inventory with correct signatures",
182:             "Keep mismatch rate below 5%"
183:         ]
184:     },
185:     "Test 5": {
186:         "name": "Orchestrator Runtime",
187:         "purpose": "Tests correct layer evaluation and aggregation",
188:         "impact": "Runtime errors prevent policy analysis",
189:         "remediation": [
190:             "Fix layer ordering issues",
191:             "Ensure context propagation works correctly",
192:             "Test calibration resolution without errors"
193:         ]
194:     },
195:     "Test 6": {
196:         "name": "No Hardcoded Calibrations",
197:         "purpose": "Scans for magic numbers in calibration-sensitive code",
198:         "impact": "Hardcoded values prevent systematic calibration",
199:         "remediation": [
200:             "Move all thresholds, weights, scores to configuration",
201:             "Use calibration_registry.get_calibration() instead of literals",
202:             "Update intrinsic_calibration.json with externalized values"
203:         ]
204:     },
205:     "Test 7": {
206:         "name": "Performance Benchmarks",
207:         "purpose": "Validates load times and calibration speed",
208:         "impact": "Slow performance degrades user experience",
209:         "remediation": [
210:             "Optimize JSON file size and structure",
211:             "Ensure load intrinsic.json < 1s",
212:             "Ensure calibrate 30 executors < 5s",
213:             "Ensure calibrate 200 methods < 30s"
214:         ]
215:     }
216: }
217:
218: for test_name in failed_tests:
219:     test_num = test_name.split(":")[0].replace("Test ", "Test")
220:     test_info = test_descriptions.get(test_num, {})
221:
222:     report.append(f"### {test_name}")
223:     report.append("")
```

```
224:  
225:         if test_info:  
226:             report.append(f"**Purpose:** {test_info.get('purpose', 'N/A')})")  
227:             report.append("")  
228:             report.append(f"**Impact:** {test_info.get('impact', 'N/A')})")  
229:             report.append("")  
230:             report.append("**Remediation Steps:**")  
231:             report.append("")  
232:             for step in test_info.get('remediation', []):  
233:                 report.append(f"1. {step}")  
234:             report.append("")  
235:  
236:             test_key = f"test{test_num.replace('Test', '')}-" + test_info.get('name', '').lower().replace(' ', '-')  
237:             if test_key in failures:  
238:                 report.append("**Specific Failures:**")  
239:                 report.append("")  
240:                 for failure in failures[test_key][:5]:  
241:                     report.append(f"- '{failure['name']}'")  
242:                     if failure['message']:  
243:                         report.append(f" - {failure['message'][:200]}")  
244:                 report.append("")  
245:  
246:                 report.append("---")  
247:                 report.append("")  
248:                 report.append("## Required Actions")  
249:                 report.append("")  
250:                 report.append("1. **DO NOT MERGE** this PR until all tests pass")  
251:                 report.append("2. Review failure details above")  
252:                 report.append("3. Apply remediation steps for each failed test")  
253:                 report.append("4. Re-run validation suite: `pytest tests/calibration_system/ -v`")  
254:                 report.append("5. Verify all tests pass locally before pushing")  
255:                 report.append("")  
256:                 report.append("## Support")  
257:                 report.append("")  
258:                 report.append("For assistance:")  
259:                 report.append("- Review test source code in `tests/calibration_system/`")  
260:                 report.append("- Check calibration documentation in `AGENTS.md`")  
261:                 report.append("- Examine test output artifacts for detailed error messages")  
262:  
263:             return "\n".join(report)  
264:  
265:  
266: def main():  
267:     """Main entry point"""  
268:     args = parse_args()  
269:     report = generate_report(args)  
270:  
271:     output_path = Path(args.output)  
272:     output_path.write_text(report)  
273:  
274:     print(f"\u234\u234 Generated failure report: {output_path}")  
275:  
276:  
277: if __name__ == "__main__":  
278:     main()  
279:
```

```
280:  
281:  
282: =====  
283: FILE: tests/calibration_system/manual_verification.py  
284: =====  
285:  
286: #!/usr/bin/env python3  
287: """  
288: Manual Verification Script for Calibration System Tests  
289:  
290: This script manually tests the calibration validation system by:  
291: 1. Running basic checks without pytest  
292: 2. Simulating failures to verify detection  
293: 3. Generating a report  
294: """  
295: import json  
296: import sys  
297: from pathlib import Path  
298:  
299:  
300: def check_executors_methods():  
301:     """Check executors_methods.json structure"""  
302:     print("=" * 60)  
303:     print("CHECK 1: Executors Methods Structure")  
304:     print("=" * 60)  
305:  
306:     path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")  
307:     if not path.exists():  
308:         print(f"\u27e8\235\214 FAIL: {path} not found")  
309:         return False  
310:  
311:     with open(path) as f:  
312:         data = json.load(f)  
313:  
314:     if not isinstance(data, list):  
315:         print(f"\u27e8\235\214 FAIL: Expected list, got {type(data)}")  
316:         return False  
317:  
318:     print(f"\u27e8\234\205 Found {len(data)} executors")  
319:  
320:     if len(data) != 30:  
321:         print(f"\u27e8\235\214 FAIL: Expected 30 executors, found {len(data)}")  
322:         return False  
323:  
324:     print(f"\u27e8\234\205 Exactly 30 executors present")  
325:  
326:     total_methods = sum(len(e.get("methods", [])) for e in data)  
327:     print(f"\u27e8\234\205 Total methods across all executors: {total_methods}")  
328:  
329:     return True  
330:  
331:  
332: def check_intrinsic_calibration():  
333:     """Check intrinsic_calibration.json structure"""  
334:     print("\n" + "=" * 60)  
335:     print("CHECK 2: Intrinsic Calibration Structure")
```

```
336:     print("=" * 60)
337:
338:     path = Path("system/config/calibration/intrinsic_calibration.json")
339:     if not path.exists():
340:         print(f"\u235\u214 FAIL: {path} not found")
341:         return False
342:
343:     with open(path) as f:
344:         data = json.load(f)
345:
346:     if not isinstance(data, dict):
347:         print(f"\u235\u214 FAIL: Expected dict, got {type(data)}")
348:         return False
349:
350:     methods = [k for k in data.keys() if k != "_metadata"]
351:     print(f"\u234\u205 Found {len(methods)} method calibrations")
352:
353:     if len(methods) == 0:
354:         print("\u232 i.\u217 WARNING: No calibrations found (empty file)")
355:         return False
356:
357:     computed = sum(1 for k, v in data.items()
358:                     if k != "_metadata" and v.get("status") == "computed")
359:
360:     if len(methods) > 0:
361:         coverage = computed / len(methods) * 100
362:         print(f"Coverage: {computed}/{len(methods)} = {coverage:.1f}% with status='computed'")
363:
364:         if coverage >= 80:
365:             print(f"\u234\u205 Coverage \u211\u2014 80%")
366:             return True
367:         else:
368:             print(f"\u235\u214 FAIL: Coverage < 80%")
369:             return False
370:
371:     return True
372:
373:
374: def check_inventory_consistency():
375:     """Check consistency between executor methods and calibrations"""
376:     print("\n" + "=" * 60)
377:     print("CHECK 3: Inventory Consistency")
378:     print("=" * 60)
379:
380:     executors_path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
381:     intrinsic_path = Path("system/config/calibration/intrinsic_calibration.json")
382:
383:     with open(executors_path) as f:
384:         executors = json.load(f)
385:
386:     with open(intrinsic_path) as f:
387:         calibrations = json.load(f)
388:
389:     all_methods = set()
390:     for executor in executors:
391:         for method in executor.get("methods", []):
```

```
392:         method_id = f"{method['class']}.{method['method']}"
393:         all_methods.add(method_id)
394:
395:     calibrated_methods = set(calibrations.keys()) - {"_metadata"}
396:
397:     missing = all_methods - calibrated_methods
398:     extra = calibrated_methods - all_methods
399:
400:     print(f"Methods in executors: {len(all_methods)}")
401:     print(f"Methods in calibrations: {len(calibrated_methods)}")
402:
403:     if missing:
404:         print(f"\u235\u214 FAIL: {len(missing)} methods missing calibration")
405:         print(f"Sample missing: {list(missing)[:5]}")
406:         return False
407:     else:
408:         print(f"\u234\u205 All executor methods have calibration entries")
409:
410:     if extra:
411:         print(f"\u232 \u217 WARNING: {len(extra)} extra calibrations")
412:         print(f"Sample extra: {list(extra)[:5]}")
413:
414:     return True
415:
416:
417: def check_layer_coverage():
418:     """Check that all executors have all 8 layers"""
419:     print("\n" + "=" * 60)
420:     print("CHECK 4: Layer Coverage")
421:     print("=" * 60)
422:
423:     required_layers = [
424:         "ingestion", "extraction", "transformation", "validation",
425:         "aggregation", "scoring", "reporting", "meta"
426:     ]
427:
428:     path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
429:     with open(path) as f:
430:         executors = json.load(f)
431:
432:     failures = []
433:
434:     for executor in executors:
435:         layers = set()
436:         for method in executor.get("methods", []):
437:             if "layer" in method:
438:                 layers.add(method["layer"])
439:
440:         missing = required_layers - layers
441:         if missing:
442:             failures.append((executor["executor_id"], missing))
443:
444:     if failures:
445:         print(f"\u235\u214 FAIL: {len(failures)} executors missing layers")
446:         for exec_id, missing in failures[:5]:
447:             print(f"  {exec_id}: missing {sorted(missing)}")
```

```
448:         return False
449:     else:
450:         print(f"\u234\u205 All {len(executors)} executors have all 8 layers")
451:         return True
452:
453:
454: def simulate_failure_detection():
455:     """Simulate a failure by temporarily modifying data"""
456:     print("\n" + "=" * 60)
457:     print("SIMULATION: Failure Detection")
458:     print("=" * 60)
459:     print("This would test if the system detects intentional failures.")
460:     print("Skipping actual modification to preserve data integrity.")
461:     print("\u234\u205 Simulation complete (dry run)")
462:     return True
463:
464:
465: def main():
466:     """Run all checks"""
467:     print("\n" + "=" * 60)
468:     print("CALIBRATION SYSTEM MANUAL VERIFICATION")
469:     print("=" * 60)
470:
471:     results = {
472:         "executors_methods": check_executors_methods(),
473:         "intrinsic_calibration": check_intrinsic_calibration(),
474:         "inventory_consistency": check_inventory_consistency(),
475:         "layer_coverage": check_layer_coverage(),
476:         "failure_simulation": simulate_failure_detection(),
477:     }
478:
479:     print("\n" + "=" * 60)
480:     print("SUMMARY")
481:     print("=" * 60)
482:
483:     for check, passed in results.items():
484:         status = "\u234\u205 PASS" if passed else "\u235\u214 FAIL"
485:         print(f"{status}: {check}")
486:
487:     all_passed = all(results.values())
488:
489:     print("\n" + "=" * 60)
490:     if all_passed:
491:         print("\u234\u205 ALL CHECKS PASSED - System appears ready")
492:     else:
493:         print("\u235\u214 SOME CHECKS FAILED - System NOT ready")
494:     print("=" * 60)
495:
496:     return 0 if all_passed else 1
497:
498:
499: if __name__ == "__main__":
500:     sys.exit(main())
501:
502:
503:
```

```
504: =====
505: FILE: tests/calibration_system/test_ast_extraction_accuracy.py
506: =====
507:
508: """
509: Test 4: AST Extraction Accuracy - Comparing Extracted Signatures vs Actual Code
510:
511: Validates that method signatures in the calibration system match actual source code:
512: - Parse Python files to extract actual method signatures
513: - Compare with signatures stored in calibration metadata
514: - Detect mismatches in parameter names, types, defaults
515:
516: FAILURE CONDITION: Signature mismatch > 5% = SYSTEM NOT READY
517: """
518: import ast
519: import json
520: import pytest
521: from pathlib import Path
522: from typing import Dict, List, Set, Any, Optional, Tuple
523: import inspect
524:
525:
526: class MethodSignatureExtractor(ast.NodeVisitor):
527:     """AST visitor to extract method signatures"""
528:
529:     def __init__(self):
530:         self.signatures: Dict[str, Dict[str, Any]] = {}
531:         self.current_class: Optional[str] = None
532:
533:     def visit_ClassDef(self, node: ast.ClassDef) -> None:
534:         """Visit class definition"""
535:         old_class = self.current_class
536:         self.current_class = node.name
537:         self.generic_visit(node)
538:         self.current_class = old_class
539:
540:     def visit_FunctionDef(self, node: ast.FunctionDef) -> None:
541:         """Visit function/method definition"""
542:         if self.current_class:
543:             method_id = f"{self.current_class}.{node.name}"
544:         else:
545:             method_id = node.name
546:
547:         args = []
548:         kwargs = []
549:         has_self = False
550:
551:         for i, arg in enumerate(node.args.args):
552:             arg_name = arg.arg
553:             if i == 0 and arg_name in ("self", "cls"):
554:                 has_self = True
555:                 continue
556:
557:             if i < len(node.args.args) - len(node.args.defaults):
558:                 args.append(arg_name)
559:             else:
```

```
560:         kwargs.append(arg_name)
561:
562:     returns = ast.unparse(node.returns) if node.returns else "Any"
563:     is_async = isinstance(node, ast.AsyncFunctionDef)
564:
565:     self.signatures[method_id] = {
566:         "args": args,
567:         "kwargs": kwargs,
568:         "returns": returns,
569:         "is_async": is_async,
570:         "has_self": has_self,
571:         "lineno": node.lineno,
572:     }
573:
574:     visit_AsyncFunctionDef = visit_FunctionDef
575:
576:
577: class TestASTExtractionAccuracy:
578:
579:     MAX_MISMATCH_PERCENT = 5.0
580:
581:     @pytest.fixture(scope="class")
582:     def executors_methods(self) -> Dict[str, Any]:
583:         """Load executors_methods.json"""
584:         path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
585:         with open(path) as f:
586:             return json.load(f)
587:
588:     @pytest.fixture(scope="class")
589:     def source_signatures(self) -> Dict[str, Dict[str, Any]]:
590:         """Extract signatures from actual source files"""
591:         signatures = {}
592:
593:         src_dirs = [
594:             Path("src/farfan_pipeline"),
595:             Path("farfan_core/farfan_core"),
596:         ]
597:
598:         for src_dir in src_dirs:
599:             if not src_dir.exists():
600:                 continue
601:
602:             for py_file in src_dir.rglob("*.py"):
603:                 if "test" in str(py_file) or "__pycache__" in str(py_file):
604:                     continue
605:
606:                 try:
607:                     tree = ast.parse(py_file.read_text(), filename=str(py_file))
608:                     extractor = MethodSignatureExtractor()
609:                     extractor.visit(tree)
610:
611:                     for method_id, sig in extractor.signatures.items():
612:                         sig["file"] = str(py_file)
613:                         signatures[method_id] = sig
614:
615:                 except (SyntaxError, UnicodeDecodeError) as e:
```



```

672:                 "method": method_id,
673:                 "stored": stored_sig,
674:                 "actual": source_sig
675:             })
676:
677:     mismatch_rate = (len(mismatches) / total_checked * 100) if total_checked > 0 else 0
678:
679:     if mismatches:
680:         msg = f"\nSignature mismatches ({len(mismatches)}/{total_checked} = {mismatch_rate:.1f}%) :\n"
681:         for item in mismatches[:5]:
682:             msg += f"    {item['method']}:\n"
683:             msg += f"        Stored: {item['stored']}\n"
684:             msg += f"        Actual: {item['actual']}\n"
685:         print(msg)
686:
687:     assert mismatch_rate <= self.MAX_MISMATCH_PERCENT, \
688:         f"Signature mismatch rate {mismatch_rate:.1f}% exceeds threshold {self.MAX_MISMATCH_PERCENT}%"
```

689:

```

690:     def _signatures_match(
691:         self, stored: Dict[str, Any], actual: Dict[str, Any]
692:     ) -> bool:
693:         """Compare two signatures for equality"""
694:         if not isinstance(stored, dict) or not isinstance(actual, dict):
695:             return False
696:
697:         stored_args = stored.get("args", [])
698:         actual_args = actual.get("args", [])
699:
700:         if len(stored_args) != len(actual_args):
701:             return False
702:
703:         for s_arg, a_arg in zip(stored_args, actual_args):
704:             if isinstance(s_arg, str) and isinstance(a_arg, str):
705:                 if s_arg != a_arg:
706:                     return False
707:
708:         return True
709:
710:     def test_private_methods_marked(self, executors_methods):
711:         """Verify private methods (starting with _) are marked"""
712:         private_methods = []
713:
714:         for executor in executors_methods:
715:             for method in executor["methods"]:
716:                 if method["method"].startswith("_"):
717:                     private_methods.append({
718:                         "executor": executor["executor_id"],
719:                         "method": f"{method['class']}.{method['method']}"
720:                     })
721:
722:         if private_methods:
723:             print(f"\nFound {len(private_methods)} private methods in executors")
724:
725:     def test_async_methods_identified(self, source_signatures):
726:         """Identify async methods in source code"""
727:         async_methods = [

```

```
728:         method_id for method_id, sig in source_signatures.items()
729:         if sig.get("is_async")
730:     ]
731:
732:     if async_methods:
733:         print(f"\nFound {len(async_methods)} async methods: {async_methods[:5]}")
734:
735: def test_method_parameter_count_reasonable(self, source_signatures):
736:     """Verify methods don't have excessive parameters"""
737:     MAX_PARAMS = 10
738:
739:     excessive_params = []
740:
741:     for method_id, sig in source_signatures.items():
742:         total_params = len(sig.get("args", [])) + len(sig.get("kwargs", []))
743:
744:         if total_params > MAX_PARAMS:
745:             excessive_params.append({
746:                 "method": method_id,
747:                 "param_count": total_params,
748:                 "file": sig.get("file")
749:             })
750:
751:     if excessive_params:
752:         msg = f"\nMethods with >{MAX_PARAMS} parameters:\n"
753:         for item in excessive_params[:5]:
754:             msg += f"  {item['method']}: {item['param_count']} params\n"
755:         print(msg)
756:
757: def test_return_type_annotations_present(self, source_signatures):
758:     """Check for return type annotations"""
759:     missing_returns = []
760:
761:     for method_id, sig in source_signatures.items():
762:         if method_id.startswith("_"):
763:             continue
764:
765:         if sig.get("returns") == "Any" or not sig.get("returns"):
766:             missing_returns.append(method_id)
767:
768:     if missing_returns:
769:         coverage = (1 - len(missing_returns) / len(source_signatures)) * 100
770:         print(
771:             f"\nReturn type annotation coverage: {coverage:.1f}% "
772:             f"({len(source_signatures) - len(missing_returns)}/{len(source_signatures)})"
773:         )
774:
775:
776:
777: =====
778: FILE: tests/calibration_system/test_canonical_id_format.py
779: =====
780:
781: """
782: test_canonical_id_format.py - Canonical ID Format Enforcement
783:
```

```
784: This test suite enforces Constraint 1: canonical_method_catalogue_v2.json is the
785: ONLY source of truth for method IDs, and all IDs must follow canonical format.
786:
787: Tests:
788: - Canonical ID format validation (regex pattern)
789: - Cross-file consistency (all JSONs reference canonical IDs only)
790: - No duplicate IDs
791: - No invalid characters (spaces, special chars)
792: """
793:
794: import json
795: import re
796: from pathlib import Path
797: from typing import Dict, List, Set
798:
799: import pytest
800:
801:
802: # Paths relative to repo root
803: REPO_ROOT = Path(__file__).parent.parent.parent
804: CATALOG_PATH = REPO_ROOT / "config" / "canonical_method_catalogue_v2.json"
805: PARAMETRIZED_PATH = REPO_ROOT / "config" / "canonic_inventory_methods_parametrized.json"
806: LAYERS_PATH = REPO_ROOT / "config" / "canonic_inventory_methods_layers.json"
807: INTRINSIC_PATH = REPO_ROOT / "system" / "config" / "calibration" / "intrinsic_calibration.json"
808:
809: # Canonical ID format: module.Class.method (with optional .linenumber)
810: # Must NOT contain: spaces, special chars except dot and underscore
811: CANONICAL_PATTERN = re.compile(r'^[a-zA-Z0-9_]+(\.[a-zA-Z0-9_]+)+$')
812:
813: # Invalid patterns that should trigger failure
814: INVALID_PATTERNS = [
815:     re.compile(r'\s'), # Spaces
816:     re.compile(r'::'), # Double colon separator
817:     re.compile(r'/'), # Forward slash
818:     re.compile(r'Copia de'), # Spanish "Copy of" indicates backup file
819: ]
820:
821:
822: def load_json(path: Path) -> Dict:
823:     """Load JSON file."""
824:     with open(path, 'r', encoding='utf-8') as f:
825:         return json.load(f)
826:
827:
828: @pytest.fixture(scope="module")
829: def canonical_ids() -> Set[str]:
830:     """Load canonical IDs from source of truth."""
831:     catalog = load_json(CATALOG_PATH)
832:     assert isinstance(catalog, list), "Catalog must be a list"
833:
834:     ids = set()
835:     for entry in catalog:
836:         if 'unique_id' in entry:
837:             ids.add(entry['unique_id'])
838:
839:     assert len(ids) > 0, "Catalog must contain at least one method"
```

```
840:     return ids
841:
842:
843: def extract_method_ids(file_path: Path, file_type: str) -> List[str]:
844:     """Extract method IDs from a JSON file."""
845:     data = load_json(file_path)
846:
847:     if file_type == 'catalog':
848:         return [entry['unique_id'] for entry in data if 'unique_id' in entry]
849:
850:     # Dict-based files (layers, intrinsic, parametrized)
851:     if isinstance(data, dict):
852:         # Skip metadata keys starting with $
853:         return [k for k in data.keys() if not k.startswith('$')]
854:
855:     return []
856:
857:
858: class TestCanonicalIDFormat:
859:     """Test suite for canonical ID format validation."""
860:
861:     def test_catalog_ids_match_canonical_pattern(self, canonical_ids):
862:         """[Constraint 1] All catalog IDs must match canonical pattern."""
863:         invalid_ids = []
864:
865:         for id_ in canonical_ids:
866:             if not CANONICAL_PATTERN.match(id_):
867:                 invalid_ids.append(id_)
868:
869:         assert len(invalid_ids) == 0, (
870:             f"Found {len(invalid_ids)} IDs not matching canonical pattern:\n" +
871:             "\n".join(f" - {id_}" for id_ in invalid_ids[:10])
872:         )
873:
874:     def test_catalog_ids_no_invalid_patterns(self, canonical_ids):
875:         """[Constraint 1] Catalog IDs must not contain invalid patterns."""
876:         violations = {}
877:
878:         for id_ in canonical_ids:
879:             for pattern in INVALID_PATTERNS:
880:                 if pattern.search(id_):
881:                     if pattern.pattern not in violations:
882:                         violations[pattern.pattern] = []
883:                         violations[pattern.pattern].append(id_)
884:
885:         assert len(violations) == 0, (
886:             "Found invalid patterns in catalog IDs:\n" +
887:             "\n".join(
888:                 f" Pattern '{p}': {len(ids)} violations\n" +
889:                 "\n".join(f" - {id_}" for id_ in ids[:5])
890:                 for p, ids in violations.items()
891:             )
892:         )
893:
894:     def test_catalog_no_duplicates(self, canonical_ids):
895:         """[Constraint 1] Catalog must not contain duplicate IDs."""
```

```
896:     catalog = load_json(CATALOG_PATH)
897:     all_ids = [entry['unique_id'] for entry in catalog if 'unique_id' in entry]
898:
899:     duplicates = [id_ for id_ in all_ids if all_ids.count(id_) > 1]
900:     unique_duplicates = list(set(duplicates))
901:
902:     assert len(unique_duplicates) == 0, (
903:         f"Found {len(unique_duplicates)} duplicate IDs in catalog:\n" +
904:         "\n".join(f" - {id_} (appears {all_ids.count(id_)}x)"
905:                  for id_ in unique_duplicates[:10])
906:     )
907:
908:
909: class TestCrossFileConsistency:
910:     """Test suite for cross-file consistency validation."""
911:
912:     def test_layers_json_uses_canonical_ids_only(self, canonical_ids):
913:         """[Constraint 1] All IDs in layers JSON must exist in canonical catalog."""
914:         layers_ids = extract_method_ids(LAYERS_PATH, 'layers')
915:         non_canonical = [id_ for id_ in layers_ids if id_ not in canonical_ids]
916:
917:         assert len(non_canonical) == 0, (
918:             f"Found {len(non_canonical)} non-canonical IDs in layers JSON:\n" +
919:             "\n".join(f" - {id_}" for id_ in non_canonical)
920:         )
921:
922:     def test_intrinsic_json_uses_canonical_ids_only(self, canonical_ids):
923:         """[Constraint 1] All IDs in intrinsic calibration JSON must exist in canonical catalog."""
924:         intrinsic_ids = extract_method_ids(INTRINSIC_PATH, 'intrinsic')
925:         non_canonical = [id_ for id_ in intrinsic_ids if id_ not in canonical_ids]
926:
927:         assert len(non_canonical) == 0, (
928:             f"Found {len(non_canonical)} non-canonical IDs in intrinsic JSON:\n" +
929:             "\n".join(f" - {id_}" for id_ in non_canonical)
930:         )
931:
932:     def test_parametrized_json_uses_canonical_ids_only(self, canonical_ids):
933:         """[Constraint 1] All IDs in parametrized JSON must exist in canonical catalog."""
934:         # Skip this test if file format is unexpected
935:         try:
936:             param_ids = extract_method_ids(PARAMETRIZED_PATH, 'parametrized')
937:         except Exception:
938:             pytest.skip("Parametrized JSON has unexpected format")
939:         return
940:
941:         non_canonical = [id_ for id_ in param_ids if id_ not in canonical_ids]
942:
943:         assert len(non_canonical) == 0, (
944:             f"Found {len(non_canonical)} non-canonical IDs in parametrized JSON:\n" +
945:             "\n".join(f" - {id_}" for id_ in non_canonical[:10])
946:         )
947:
948:
949: class TestCatalogIntegrity:
950:     """Test suite for catalog structural integrity."""
951:
```

```
952:     def test_catalog_is_list(self):
953:         """Catalog must be a JSON list."""
954:         catalog = load_json(CATALOG_PATH)
955:         assert isinstance(catalog, list), "Catalog must be a list of method entries"
956:
957:     def test_catalog_entries_have_unique_id(self):
958:         """All catalog entries must have 'unique_id' field."""
959:         catalog = load_json(CATALOG_PATH)
960:         missing_id = [i for i, entry in enumerate(catalog) if 'unique_id' not in entry]
961:
962:         assert len(missing_id) == 0, (
963:             f"Found {len(missing_id)} entries without 'unique_id' field:\n" +
964:             f"  Indices: {missing_id[:10]}")
965:
966:
967:     def test_catalog_minimum_size(self, canonical_ids):
968:         """Catalog must contain reasonable number of methods."""
969:         # Should have at least 30 executors + other methods
970:         assert len(canonical_ids) >= 30, (
971:             f"Catalog only contains {len(canonical_ids)} methods."
972:             f"Expected at least 30 (6 dimensions x 5 questions).")
973:
974:
975:
976: if __name__ == '__main__':
977:     # Allow running as script for quick validation
978:     pytest.main([__file__, '-v'])
979:
980:
981:
982: =====
983: FILE: tests/calibration_system/test_intrinsic_coverage.py
984: =====
985:
986: """
987: Test 3: Intrinsic Coverage - 211% Methods + All 30 Executors Have status='computed'
988:
989: Validates calibration completeness:
990: - At least 80% of all methods must have calibration status='computed'
991: - All 30 executors must have at least one method with status='computed'
992: - No executor should be entirely 'excluded' or 'manual'
993:
994: FAILURE CONDITION: Coverage < 80% OR any executor without computed status = NOT READY
995: """
996: import json
997: import pytest
998: from pathlib import Path
999: from typing import Dict, List, Set, Any
1000:
1001:
1002: class TestIntrinsicCoverage:
1003:
1004:     MIN_COVERAGE_PERCENT = 80.0
1005:
1006:     @pytest.fixture(scope="class")
1007:     def executors_methods(self) -> Dict[str, Any]:
```

```
1008:     """Load executors_methods.json"""
1009:     path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
1010:     with open(path) as f:
1011:         return json.load(f)
1012:
1013:     @pytest.fixture(scope="class")
1014:     def intrinsic_calibration(self) -> Dict[str, Any]:
1015:         """Load intrinsic_calibration.json"""
1016:         path = Path("system/config/calibration/intrinsic_calibration.json")
1017:         with open(path) as f:
1018:             return json.load(f)
1019:
1020:     def test_intrinsic_calibration_not_empty(self, intrinsic_calibration):
1021:         """Verify intrinsic_calibration.json is not empty"""
1022:         methods = {k: v for k, v in intrinsic_calibration.items() if k != "_metadata"}
1023:         assert len(methods) > 0, "intrinsic_calibration.json has no method entries"
1024:
1025:     def test_80_percent_coverage_with_computed_status(
1026:         self, executors_methods, intrinsic_calibration
1027):
1028:         """CRITICAL: Verify at least 80% of methods have status='computed'"""
1029:         all_methods = set()
1030:         for executor in executors_methods:
1031:             for method in executor["methods"]:
1032:                 method_id = f"{method['class']}.{method['method']}"
1033:                 all_methods.add(method_id)
1034:
1035:         total_methods = len(all_methods)
1036:         computed_methods = set()
1037:
1038:         for method_id, calibration in intrinsic_calibration.items():
1039:             if method_id == "_metadata":
1040:                 continue
1041:
1042:             if method_id in all_methods:
1043:                 status = calibration.get("status", "missing")
1044:                 if status == "computed":
1045:                     computed_methods.add(method_id)
1046:
1047:         coverage_percent = (len(computed_methods) / total_methods * 100) if total_methods > 0 else 0
1048:
1049:         missing_computed = all_methods - computed_methods
1050:
1051:         failure_msg = (
1052:             f"CRITICAL: Calibration coverage is {coverage_percent:.1f}% "
1053:             f"({len(computed_methods)})/({total_methods}), "
1054:             f"minimum required is {self.MIN_COVERAGE_PERCENT}%\n"
1055:             f"Methods without 'computed' status: {len(missing_computed)}\n"
1056:
1057:
1058:         if missing_computed and len(missing_computed) <= 20:
1059:             failure_msg += "Missing methods:\n" + "\n".join(
1060:                 f"  - {m}" for m in sorted(missing_computed)
1061:
1062:
1063:         assert coverage_percent >= self.MIN_COVERAGE_PERCENT, failure_msg
```

```
1064:  
1065:     def test_all_30_executors_have_computed_methods(  
1066:         self, executors_methods, intrinsic_calibration  
1067:     ):  
1068:         """CRITICAL: Every executor must have at least one method with status='computed'"""  
1069:         failures = []  
1070:  
1071:         for executor in executors_methods:  
1072:             executor_id = executor["executor_id"]  
1073:             has_computed = False  
1074:  
1075:             for method in executor["methods"]:  
1076:                 method_id = f"{method['class']}.{method['method']}"  
1077:                 calibration = intrinsic_calibration.get(method_id, {})  
1078:  
1079:                 if calibration.get("status") == "computed":  
1080:                     has_computed = True  
1081:                     break  
1082:  
1083:                 if not has_computed:  
1084:                     failures.append(  
1085:                         f"Executor {executor_id} has no methods with status='computed'"  
1086:                     )  
1087:  
1088:             assert not failures, \  
1089:                 f"CRITICAL: Executors without computed calibrations:\n" + "\n".join(failures)  
1090:  
1091:     def test_status_field_valid_values(self, intrinsic_calibration):  
1092:         """Verify all status fields have valid values"""  
1093:         valid_statuses = {"computed", "excluded", "manual", "error"}  
1094:  
1095:         for method_id, calibration in intrinsic_calibration.items():  
1096:             if method_id == "_metadata":  
1097:                 continue  
1098:  
1099:             status = calibration.get("status")  
1100:             assert status in valid_statuses, \  
1101:                 f"Invalid status '{status}' for {method_id}. "  
1102:                 f"Must be one of: {valid_statuses}"  
1103:  
1104:     def test_computed_methods_have_scores(self, intrinsic_calibration):  
1105:         """Verify methods with status='computed' have actual scores"""  
1106:         for method_id, calibration in intrinsic_calibration.items():  
1107:             if method_id == "_metadata":  
1108:                 continue  
1109:  
1110:             if calibration.get("status") == "computed":  
1111:                 assert "b_theory" in calibration or "scores" in calibration, \  
1112:                     f"Method {method_id} has status='computed' but no scores"  
1113:  
1114:     def test_no_fully_excluded_executors(self, executors_methods, intrinsic_calibration):  
1115:         """Warn if any executor has all methods excluded"""  
1116:         warnings = []  
1117:  
1118:         for executor in executors_methods:  
1119:             executor_id = executor["executor_id"]
```

```
1120:         all_excluded = True
1121:
1122:         for method in executor["methods"]:
1123:             method_id = f"{method['class']}.{method['method']}"
1124:             calibration = intrinsic_calibration.get(method_id, {})
1125:
1126:             if calibration.get("status") != "excluded":
1127:                 all_excluded = False
1128:                 break
1129:
1130:         if all_excluded:
1131:             warnings.append(
1132:                 f"WARNING: Executor {executor_id} has all methods excluded"
1133:             )
1134:
1135:     if warnings:
1136:         print("\n" + "\n".join(warnings))
1137:
1138: def test_excluded_methods_have_reason(self, intrinsic_calibration):
1139:     """Verify excluded methods have exclusion reason"""
1140:     for method_id, calibration in intrinsic_calibration.items():
1141:         if method_id == "_metadata":
1142:             continue
1143:
1144:         if calibration.get("status") == "excluded":
1145:             assert "exclusion_reason" in calibration or "reason" in calibration, \
1146:                 f"Excluded method {method_id} missing exclusion reason"
1147:
1148: def test_manual_methods_documented(self, intrinsic_calibration):
1149:     """Verify methods with status='manual' are documented"""
1150:     manual_methods = []
1151:
1152:     for method_id, calibration in intrinsic_calibration.items():
1153:         if method_id == "_metadata":
1154:             continue
1155:
1156:         if calibration.get("status") == "manual":
1157:             manual_methods.append(method_id)
1158:             assert "note" in calibration or "reason" in calibration, \
1159:                 f"Manual method {method_id} missing documentation"
1160:
1161:     if manual_methods:
1162:         print(f"\nFound {len(manual_methods)} manual calibrations: {manual_methods[:5]}")
1163:
1164: def test_error_status_methods_tracked(self, intrinsic_calibration):
1165:     """Track methods with status='error' for debugging"""
1166:     error_methods = []
1167:
1168:     for method_id, calibration in intrinsic_calibration.items():
1169:         if method_id == "_metadata":
1170:             continue
1171:
1172:         if calibration.get("status") == "error":
1173:             error_methods.append(method_id)
1174:
1175:     if error_methods:
```

```
1176:         print(
1177:             f"\nWARNING: {len(error_methods)} methods have status='error': "
1178:             f"{error_methods[:5]}"
1179:         )
1180:
1181:     def test_coverage_by_executor(self, executors_methods, intrinsic_calibration):
1182:         """Report coverage statistics per executor"""
1183:         stats = []
1184:
1185:         for executor in executors_methods:
1186:             executor_id = executor["executor_id"]
1187:             total = len(executor["methods"])
1188:             computed = 0
1189:
1190:             for method in executor["methods"]:
1191:                 method_id = f"{method['class']}.{method['method']}"
1192:                 calibration = intrinsic_calibration.get(method_id, {})
1193:
1194:                 if calibration.get("status") == "computed":
1195:                     computed += 1
1196:
1197:                 coverage = (computed / total * 100) if total > 0 else 0
1198:                 stats.append((executor_id, coverage, computed, total))
1199:
1200:         stats.sort(key=lambda x: x[1])
1201:
1202:         low_coverage = [s for s in stats if s[1] < self.MIN_COVERAGE_PERCENT]
1203:
1204:         if low_coverage:
1205:             msg = "\nExecutors with coverage below threshold:\n"
1206:             for executor_id, coverage, computed, total in low_coverage[:10]:
1207:                 msg += f" {executor_id}: {coverage:.1f}% ({computed}/{total})\n"
1208:             print(msg)
1209:
1210:     def test_metadata_present_in_intrinsic(self, intrinsic_calibration):
1211:         """Verify _metadata is present with version info"""
1212:         assert "_metadata" in intrinsic_calibration, \
1213:             "intrinsic_calibration.json missing _metadata"
1214:
1215:         metadata = intrinsic_calibration["_metadata"]
1216:         assert "version" in metadata, "_metadata missing version"
1217:         assert "generated" in metadata or "last_updated" in metadata, \
1218:             "_metadata missing timestamp"
1219:
1220:
1221:
1222: =====
1223: FILE: tests/calibration_system/test_inventory_consistency.py
1224: =====
1225:
1226: """
1227: Test 1: Inventory Consistency - Verifying Same Methods in All JSONs
1228:
1229: Validates that all executor method inventories are consistent across:
1230: - executors_methods.json: 30 executors with method lists
1231: - intrinsic_calibration.json: All methods have calibration entries
```

```
1232: - canonical method catalog (if exists)
1233:
1234: FAILURE CONDITION: Any method missing from any inventory = SYSTEM NOT READY
1235: """
1236: import json
1237: import pytest
1238: from pathlib import Path
1239: from typing import Dict, List, Set, Any
1240:
1241:
1242: class TestInventoryConsistency:
1243:     @pytest.fixture(scope="class")
1244:     def executors_methods(self) -> Dict[str, Any]:
1245:         """Load executors_methods.json"""
1246:         path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
1247:         assert path.exists(), f"executors_methods.json not found at {path}"
1248:         with open(path) as f:
1249:             return json.load(f)
1250:
1251:
1252:     @pytest.fixture(scope="class")
1253:     def intrinsic_calibration(self) -> Dict[str, Any]:
1254:         """Load intrinsic_calibration.json"""
1255:         path = Path("system/config/calibration/intrinsic_calibration.json")
1256:         assert path.exists(), f"intrinsic_calibration.json not found at {path}"
1257:         with open(path) as f:
1258:             return json.load(f)
1259:
1260:     @pytest.fixture(scope="class")
1261:     def calibration_rubric(self) -> Dict[str, Any]:
1262:         """Load calibration rubric"""
1263:         path = Path("config/intrinsic_calibration_rubric.json")
1264:         assert path.exists(), f"calibration rubric not found at {path}"
1265:         with open(path) as f:
1266:             return json.load(f)
1267:
1268:     def test_executors_methods_structure(self, executors_methods):
1269:         """Verify executors_methods.json has correct structure"""
1270:         assert isinstance(executors_methods, list), "executors_methods must be a list"
1271:         assert len(executors_methods) > 0, "executors_methods cannot be empty"
1272:
1273:         for executor in executors_methods:
1274:             assert "executor_id" in executor, f"Executor missing executor_id: {executor}"
1275:             assert "methods" in executor, f"Executor {executor.get('executor_id')} missing methods"
1276:             assert isinstance(executor["methods"], list), \
1277:                 f"Executor {executor['executor_id']} methods must be a list"
1278:
1279:     def test_30_executors_present(self, executors_methods):
1280:         """Verify exactly 30 executors are defined"""
1281:         executor_ids = [e["executor_id"] for e in executors_methods]
1282:         assert len(executor_ids) == 30, \
1283:             f"Expected 30 executors, found {len(executor_ids)}: {executor_ids}"
1284:
1285:         expected_pattern = set()
1286:         for d in range(1, 7):
1287:             for q in range(1, 6):
1288:                 expected_pattern.add(f"D{d}-Q{q}")
```

```
1288:
1289:     actual_set = set(executor_ids)
1290:     missing = expected_pattern - actual_set
1291:     extra = actual_set - expected_pattern
1292:
1293:     assert not missing, f"Missing executors: {sorted(missing)}"
1294:     assert not extra, f"Unexpected executors: {sorted(extra)}"
1295:
1296: def test_all_methods_unique_per_executor(self, executors_methods):
1297:     """Verify no duplicate methods within each executor"""
1298:     for executor in executors_methods:
1299:         methods = executor["methods"]
1300:         method_sigs = [f"{m['class']}.{m['method']}"} for m in methods]
1301:         duplicates = [sig for sig in method_sigs if method_sigs.count(sig) > 1]
1302:
1303:         assert not duplicates, \
1304:             f"Executor {executor['executor_id']} has duplicate methods: {set(duplicates)}"
1305:
1306: def test_method_format_consistency(self, executors_methods):
1307:     """Verify all methods have required fields"""
1308:     for executor in executors_methods:
1309:         for method in executor["methods"]:
1310:             assert "class" in method, \
1311:                 f"Method in {executor['executor_id']} missing 'class': {method}"
1312:             assert "method" in method, \
1313:                 f"Method in {executor['executor_id']} missing 'method': {method}"
1314:             assert isinstance(method["class"], str) and method["class"], \
1315:                 f"Invalid class name in {executor['executor_id']}: {method}"
1316:             assert isinstance(method["method"], str) and method["method"], \
1317:                 f"Invalid method name in {executor['executor_id']}: {method}"
1318:
1319: def test_intrinsic_calibration_covers_all_methods(
1320:     self, executors_methods, intrinsic_calibration
1321: ):
1322:     """CRITICAL: Verify all executor methods have intrinsic calibration entries"""
1323:     all_methods = set()
1324:     for executor in executors_methods:
1325:         for method in executor["methods"]:
1326:             method_id = f"{method['class']}.{method['method']}"
1327:             all_methods.add(method_id)
1328:
1329:     calibrated_methods = set(intrinsic_calibration.keys()) - {"_metadata"}
1330:
1331:     missing_calibrations = all_methods - calibrated_methods
1332:     extra_calibrations = calibrated_methods - all_methods
1333:
1334:     failure_report = []
1335:     if missing_calibrations:
1336:         failure_report.append(
1337:             f"CRITICAL: {len(missing_calibrations)} methods missing calibration:\n"
1338:             + "\n".join(f"  - {m}" for m in sorted(missing_calibrations)[:20])
1339:         )
1340:
1341:     if extra_calibrations:
1342:         failure_report.append(
1343:             f"WARNING: {len(extra_calibrations)} calibrations for unknown methods:\n"
```

```
1344:             + "\n".join(f" - {m}" for m in sorted(extra_calibrations)[:20])
1345:         )
1346:
1347:     assert not missing_calibrations, "\n".join(failure_report)
1348:
1349: def test_calibration_status_field_present(self, intrinsic_calibration):
1350:     """Verify all calibration entries have status field"""
1351:     for method_id, calibration in intrinsic_calibration.items():
1352:         if method_id == "_metadata":
1353:             continue
1354:
1355:         assert isinstance(calibration, dict), \
1356:             f"Calibration for {method_id} must be a dict, got {type(calibration)}"
1357:
1358:         assert "status" in calibration, \
1359:             f"Calibration for {method_id} missing 'status' field"
1360:
1361:         valid_statuses = {"computed", "excluded", "manual", "error"}
1362:         assert calibration["status"] in valid_statuses, \
1363:             f"Invalid status for {method_id}: {calibration['status']}"
1364:
1365: def test_no_empty_method_lists(self, executors_methods):
1366:     """Verify no executor has empty method list"""
1367:     for executor in executors_methods:
1368:         assert len(executor["methods"]) > 0, \
1369:             f"Executor {executor['executor_id']} has no methods"
1370:
1371: def test_method_naming_conventions(self, executors_methods):
1372:     """Verify method names follow Python conventions"""
1373:     import re
1374:
1375:     class_pattern = re.compile(r'^[A-Z][a-zA-Z0-9]*$')
1376:     method_pattern = re.compile(r'^[a-z_][a-zA-Z0-9_]*$')
1377:
1378:     for executor in executors_methods:
1379:         for method in executor["methods"]:
1380:             class_name = method["class"]
1381:             method_name = method["method"]
1382:
1383:             assert class_pattern.match(class_name), \
1384:                 f"Invalid class name in {executor['executor_id']}: {class_name}"
1385:             assert method_pattern.match(method_name), \
1386:                 f"Invalid method name in {executor['executor_id']}: {method_name}"
1387:
1388:
1389:
1390: =====
1391: FILE: tests/calibration_system/test_layer_correctness.py
1392: =====
1393:
1394: """
1395: Test 2: Layer Correctness - 30 Executors Have 8 Layers + Correct LAYER_REQUIREMENTS
1396:
1397: Validates architectural integrity:
1398: - All 30 executors must define exactly 8 layers
1399: - LAYER_REQUIREMENTS mapping must be correct and complete
```

```
1400: - Layer dependencies must be acyclic
1401:
1402: FAILURE CONDITION: Any layer mismatch = SYSTEM NOT READY
1403: """
1404: import json
1405: import pytest
1406: from pathlib import Path
1407: from typing import Dict, List, Set, Any
1408:
1409:
1410: class TestLayerCorrectness:
1411:
1412:     REQUIRED_LAYERS = {
1413:         "ingestion",
1414:         "extraction",
1415:         "transformation",
1416:         "validation",
1417:         "aggregation",
1418:         "scoring",
1419:         "reporting",
1420:         "meta"
1421:     }
1422:
1423:     @pytest.fixture(scope="class")
1424:     def executors_methods(self) -> Dict[str, Any]:
1425:         """Load executors_methods.json"""
1426:         path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
1427:         with open(path) as f:
1428:             return json.load(f)
1429:
1430:     @pytest.fixture(scope="class")
1431:     def layer_requirements_path(self) -> Path:
1432:         """Find LAYER_REQUIREMENTS definition"""
1433:         candidates = [
1434:             Path("src/farfan_pipeline/core/orchestrator/executors.py"),
1435:             Path("src/farfan_pipeline/core/orchestrator/base_executor_with_contract.py"),
1436:             Path("src/farfan_pipeline/core/layers.py"),
1437:         ]
1438:         for path in candidates:
1439:             if path.exists():
1440:                 return path
1441:         pytest.skip("LAYER_REQUIREMENTS definition file not found")
1442:
1443:     def test_8_layers_defined(self):
1444:         """Verify exactly 8 layers are required"""
1445:         assert len(self.REQUIRED_LAYERS) == 8, \
1446:             f"Expected 8 layers, found {len(self.REQUIRED_LAYERS)} "
1447:
1448:     def test_executor_methods_have_layer_field(self, executors_methods):
1449:         """Verify all methods specify a layer"""
1450:         for executor in executors_methods:
1451:             for method in executor["methods"]:
1452:                 method_sig = f"{method['class']}.{method['method']}"
1453:
1454:                 assert "layer" in method, \
1455:                     f"Method {method_sig} in {executor['executor_id']} missing 'layer' field"
```

```
1456:
1457:             layer = method["layer"]
1458:             assert layer in self.REQUIRED_LAYERS, \
1459:                 f"Invalid layer '{layer}' for {method_sig} in {executor['executor_id']}.\n" \
1460:                 f"Must be one of: {sorted(self.REQUIRED_LAYERS)}\""
1461:
1462:     def test_all_30_executors_have_all_8_layers(self, executors_methods):
1463:         """CRITICAL: Verify each of 30 executors has methods in all 8 layers"""
1464:         failures = []
1465:
1466:         for executor in executors_methods:
1467:             executor_id = executor["executor_id"]
1468:             layers_present = set()
1469:
1470:             for method in executor["methods"]:
1471:                 if "layer" in method:
1472:                     layers_present.add(method["layer"])
1473:
1474:             missing_layers = self.REQUIRED_LAYERS - layers_present
1475:
1476:             if missing_layers:
1477:                 failures.append(
1478:                     f"Executor {executor_id} missing layers: {sorted(missing_layers)}")
1479:             )
1480:
1481:         assert not failures,
1482:             f"CRITICAL: Executors with incomplete layer coverage:\n" + \
1483:             "\n".join(failures)
1484:
1485:     def test_layer_requirements_mapping_exists(self, layer_requirements_path):
1486:         """Verify LAYER_REQUIREMENTS mapping is defined in code"""
1487:         content = layer_requirements_path.read_text()
1488:
1489:         assert "LAYER_REQUIREMENTS" in content, \
1490:             f"LAYER_REQUIREMENTS not found in {layer_requirements_path}"
1491:
1492:     def test_layer_requirements_covers_all_layers(self, layer_requirements_path):
1493:         """Verify LAYER_REQUIREMENTS covers all 8 layers"""
1494:         import ast
1495:
1496:         content = layer_requirements_path.read_text()
1497:         tree = ast.parse(content)
1498:
1499:         layer_req_dict = None
1500:         for node in ast.walk(tree):
1501:             if isinstance(node, ast.Assign):
1502:                 for target in node.targets:
1503:                     if isinstance(target, ast.Name) and target.id == "LAYER_REQUIREMENTS":
1504:                         if isinstance(node.value, ast.Dict):
1505:                             layer_req_dict = {}
1506:                             for k, v in zip(node.value.keys, node.value.values):
1507:                                 if isinstance(k, ast.Constant):
1508:                                     layer_req_dict[k.value] = True
1509:
1510:         if layer_req_dict is None:
1511:             pytest.skip("Could not parse LAYER_REQUIREMENTS from source")
```

```
1512:
1513:     defined_layers = set(layer_req_dict.keys())
1514:     missing_layers = self.REQUIRED_LAYERS - defined_layers
1515:
1516:     assert not missing_layers, \
1517:         f"AYER_REQUIREMENTS missing layers: {sorted(missing_layers)}"
1518:
1519: def test_layer_dependency_acyclicity(self, executors_methods):
1520:     """Verify layer execution order is acyclic"""
1521:     layer_order = [
1522:         "ingestion",
1523:         "extraction",
1524:         "transformation",
1525:         "validation",
1526:         "aggregation",
1527:         "scoring",
1528:         "reporting",
1529:         "meta"
1530:     ]
1531:
1532:     layer_to_position = {layer: i for i, layer in enumerate(layer_order)}
1533:
1534:     for executor in executors_methods:
1535:         for i, method in enumerate(executor["methods"]):
1536:             if "layer" not in method:
1537:                 continue
1538:
1539:             current_layer = method["layer"]
1540:             current_pos = layer_to_position.get(current_layer, -1)
1541:
1542:             for prev_method in executor["methods"][:i]:
1543:                 if "layer" not in prev_method:
1544:                     continue
1545:                 prev_layer = prev_method["layer"]
1546:                 prev_pos = layer_to_position.get(prev_layer, -1)
1547:
1548:                 if current_pos < prev_pos:
1549:                     pytest.fail(
1550:                         f"Layer ordering violation in {executor['executor_id']}: "
1551:                         f"{prev_layer} (pos {prev_pos}) comes before "
1552:                         f"{current_layer} (pos {current_pos})"
1553:                     )
1554:
1555:     def test_method_distribution_across_layers(self, executors_methods):
1556:         """Analyze method distribution across layers"""
1557:         layer_counts = {layer: 0 for layer in self.REQUIRED_LAYERS}
1558:
1559:         for executor in executors_methods:
1560:             for method in executor["methods"]:
1561:                 if "layer" in method:
1562:                     layer_counts[method["layer"]] += 1
1563:
1564:         for layer, count in layer_counts.items():
1565:             assert count > 0, \
1566:                 f"Layer '{layer}' has no methods across all executors"
1567:
```

```
1568:     def test_layer_field_consistency_with_method_names(self, executors_methods):
1569:         """Verify layer assignment consistency with method naming patterns"""
1570:         layer_hints = {
1571:             "extract": "extraction",
1572:             "ingest": "ingestion",
1573:             "validate": "validation",
1574:             "transform": "transformation",
1575:             "aggregate": "aggregation",
1576:             "score": "scoring",
1577:             "report": "reporting",
1578:         }
1579:
1580:         warnings = []
1581:
1582:         for executor in executors_methods:
1583:             for method in executor["methods"]:
1584:                 if "layer" not in method:
1585:                     continue
1586:
1587:                 method_name = method["method"].lower()
1588:                 assigned_layer = method["layer"]
1589:
1590:                 for hint, suggested_layer in layer_hints.items():
1591:                     if hint in method_name and assigned_layer != suggested_layer:
1592:                         warnings.append(
1593:                             f"Potential layer mismatch: {method['class']}.{method['method']} "
1594:                             f"in {executor['executor_id']} assigned to '{assigned_layer}' "
1595:                             f"but name suggests '{suggested_layer}'"
1596:                         )
1597:
1598:             if warnings:
1599:                 print("\n".join(warnings[:10]))
1600:
1601:     def test_no_orphan_layers(self, executors_methods):
1602:         """Verify no executor has layer gaps in sequence"""
1603:         layer_order = [
1604:             "ingestion",
1605:             "extraction",
1606:             "transformation",
1607:             "validation",
1608:             "aggregation",
1609:             "scoring",
1610:             "reporting",
1611:             "meta"
1612:         ]
1613:
1614:         for executor in executors_methods:
1615:             layers_used = []
1616:             for method in executor["methods"]:
1617:                 if "layer" in method:
1618:                     layers_used.append(method["layer"])
1619:
1620:             layer_positions = sorted([
1621:                 layer_order.index(layer) for layer in set(layers_used)
1622:             ])
1623:
```

```
1624:         if not layer_positions:
1625:             continue
1626:
1627:         for i in range(len(layer_positions) - 1):
1628:             gap = layer_positions[i + 1] - layer_positions[i]
1629:             if gap > 2:
1630:                 pytest.fail(
1631:                     f"Large layer gap in {executor['executor_id']}: "
1632:                     f"{layer_order[layer_positions[i]]} -> "
1633:                     f"{layer_order[layer_positions[i + 1]]}"
1634:                 )
1635:
1636:
1637:
1638: =====
1639: FILE: tests/calibration_system/test_no_hardcoded_calibrations.py
1640: =====
1641:
1642: """
1643: Test 6: No Hardcoded Calibrations - Scanning for Magic Numbers in Calibration-Sensitive Areas
1644:
1645: Validates that calibration parameters are not hardcoded:
1646: - Scans source code for hardcoded thresholds, weights, scores
1647: - Excludes test files and configuration files
1648: - Fails if magic numbers found in calibration-sensitive code
1649:
1650: FAILURE CONDITION: Any hardcoded calibration values = NOT READY
1651:
1652: Integration: This test suite integrates with the pre-commit hook system.
1653: The pre-commit hook (scripts/pre_commit_validators.py) uses the same validation
1654: logic to block commits containing hardcoded calibration patterns.
1655: """
1656: import re
1657: import pytest
1658: from pathlib import Path
1659: from typing import Any, List, Dict, Tuple, Set
1660:
1661:
1662: class TestNoHardcodedCalibrations:
1663:
1664:     CALIBRATION_PATTERNS = [
1665:         (r'threshold\s*=\s*[0-9.]+', 'threshold assignment'),
1666:         (r'weight\s*=\s*[0-9.]+', 'weight assignment'),
1667:         (r'score\s*=\s*[0-9.]+', 'score assignment'),
1668:         (r'min_evidence\s*=\s*[0-9.]+', 'min_evidence assignment'),
1669:         (r'max_evidence\s*=\s*[0-9.]+', 'max_evidence assignment'),
1670:         (r'confidence\s*=\s*[0-9.]+', 'confidence assignment'),
1671:         (r'penalty\s*=\s*[0-9.]+', 'penalty assignment'),
1672:         (r'tolerance\s*=\s*[0-9.]+', 'tolerance assignment'),
1673:         (r'sensitivity\s*=\s*[0-9.]+', 'sensitivity assignment'),
1674:     ]
1675:
1676:     EXCLUDED_PATHS = [
1677:         "test",
1678:         "__pycache__",
1679:         ".git",
```

```
1680:         "farfan-env",
1681:         "venv",
1682:         ".venv",
1683:         "node_modules",
1684:         "config",
1685:         "system/config",
1686:     ]
1687:
1688:     ALLOWED_VALUES = {
1689:         '0', '1', '0.0', '1.0', '0.5',
1690:         '-1', '2', '10', '100',
1691:     }
1692:
1693:     @pytest.fixture(scope="class")
1694:     def source_files(self) -> List[Path]:
1695:         """Collect all Python source files to scan"""
1696:         files = []
1697:
1698:         src_dirs = [
1699:             Path("src/farfan_pipeline"),
1700:             Path("farfan_core/farfan_core"),
1701:         ]
1702:
1703:         for src_dir in src_dirs:
1704:             if not src_dir.exists():
1705:                 continue
1706:
1707:             for py_file in src_dir.rglob("*.py"):
1708:                 if self._should_exclude(py_file):
1709:                     continue
1710:
1711:                 files.append(py_file)
1712:
1713:         return files
1714:
1715:     def _should_exclude(self, path: Path) -> bool:
1716:         """Check if path should be excluded from scanning"""
1717:         path_str = str(path)
1718:
1719:         for excluded in self.EXCLUDED_PATHS:
1720:             if excluded in path_str:
1721:                 return True
1722:
1723:         return False
1724:
1725:     def test_source_files_found(self, source_files):
1726:         """Verify we have source files to scan"""
1727:         assert len(source_files) > 0, \
1728:             "No source files found to scan for hardcoded calibrations"
1729:
1730:         print(f"\nScanning {len(source_files)} source files")
1731:
1732:     def test_no_hardcoded_thresholds(self, source_files):
1733:         """CRITICAL: Verify no hardcoded threshold values in source"""
1734:         violations = self._scan_for_patterns(source_files, "threshold")
1735:
```

```
1736:         if violations:
1737:             msg = self._format_violations(violations, "threshold")
1738:             pytest.fail(msg)
1739:
1740:     def test_no_hardcoded_weights(self, source_files):
1741:         """CRITICAL: Verify no hardcoded weight values in source"""
1742:         violations = self._scan_for_patterns(source_files, "weight")
1743:
1744:         if violations:
1745:             msg = self._format_violations(violations, "weight")
1746:             pytest.fail(msg)
1747:
1748:     def test_no_hardcoded_scores(self, source_files):
1749:         """CRITICAL: Verify no hardcoded score values in source"""
1750:         violations = self._scan_for_patterns(source_files, "score")
1751:
1752:         if violations:
1753:             msg = self._format_violations(violations, "score")
1754:             pytest.fail(msg)
1755:
1756:     def test_no_hardcoded_calibration_params(self, source_files):
1757:         """CRITICAL: Comprehensive scan for all calibration patterns"""
1758:         all_violations = []
1759:
1760:         for file_path in source_files:
1761:             try:
1762:                 content = file_path.read_text()
1763:                 lines = content.split('\n')
1764:
1765:                 for lineno, line in enumerate(lines, 1):
1766:                     if self._is_comment_or_docstring(line):
1767:                         continue
1768:
1769:                     for pattern, description in self.CALIBRATION_PATTERNS:
1770:                         matches = re.finditer(pattern, line, re.IGNORECASE)
1771:
1772:                         for match in matches:
1773:                             value = match.group().split('=')[-1].strip()
1774:
1775:                             if value not in self.ALLOWED_VALUES:
1776:                                 all_violations.append({
1777:                                     'file': str(file_path),
1778:                                     'line': lineno,
1779:                                     'code': line.strip(),
1780:                                     'pattern': description,
1781:                                     'value': value
1782:                                 })
1783:
1784:             except (UnicodeDecodeError, OSError) as e:
1785:                 print(f"Warning: Could not read {file_path}: {e}")
1786:
1787:             if all_violations:
1788:                 msg = self._format_all_violations(all_violations)
1789:                 pytest.fail(msg)
1790:
1791:     def _scan_for_patterns(
```

```
1792:         self, source_files: List[Path], keyword: str
1793:     ) -> List[Dict[str, Any]]:
1794:         """Scan files for specific calibration pattern"""
1795:         violations = []
1796:         pattern = re.compile(rf'{keyword}\s*=\s*([0-9.]+)', re.IGNORECASE)
1797:
1798:         for file_path in source_files:
1799:             try:
1800:                 content = file_path.read_text()
1801:                 lines = content.split('\n')
1802:
1803:                 for lineno, line in enumerate(lines, 1):
1804:                     if self._is_comment_or_docstring(line):
1805:                         continue
1806:
1807:                     matches = pattern.finditer(line)
1808:
1809:                     for match in matches:
1810:                         value = match.group(1)
1811:
1812:                         if value not in self.ALLOWED_VALUES:
1813:                             violations.append({
1814:                                 'file': str(file_path),
1815:                                 'line': lineno,
1816:                                 'code': line.strip(),
1817:                                 'value': value
1818:                             })
1819:
1820:             except (UnicodeDecodeError, OSError):
1821:                 pass
1822:
1823:         return violations
1824:
1825:     def _is_comment_or_docstring(self, line: str) -> bool:
1826:         """Check if line is a comment or docstring"""
1827:         stripped = line.strip()
1828:         return (
1829:             stripped.startswith('#') or
1830:             stripped.startswith('"""') or
1831:             stripped.startswith("'''")
1832:         )
1833:
1834:     def _format_violations(
1835:         self, violations: List[Dict[str, Any]], keyword: str
1836:     ) -> str:
1837:         """Format violation message"""
1838:         msg = (
1839:             f"\nCRITICAL: Found {len(violations)} hardcoded {keyword} values:\n\n"
1840:         )
1841:
1842:         for v in violations[:10]:
1843:             msg += f"  {v['file']}:{v['line']}\n"
1844:             msg += f"    {v['code']}\n"
1845:             msg += f"    Value: {v['value']}\n\n"
1846:
1847:         if len(violations) > 10:
```

```
1848:             msg += f"  ... and {len(violations)} more\n"
1849:
1850:             msg += "\nAll calibration parameters must be loaded from configuration files.\n"
1851:             msg += "Move these values to intrinsic_calibration.json or calibration_registry.py\n"
1852:
1853:             return msg
1854:
1855:     def _format_all_violations(self, violations: List[Dict[str, Any]]) -> str:
1856:         """Format all violations message"""
1857:         msg = (
1858:             f"\nCRITICAL: Found {len(violations)} hardcoded calibration values:\n\n"
1859:         )
1860:
1861:         by_file = {}
1862:         for v in violations:
1863:             file_path = v['file']
1864:             if file_path not in by_file:
1865:                 by_file[file_path] = []
1866:             by_file[file_path].append(v)
1867:
1868:             for file_path, file_violations in sorted(by_file.items())[:5]:
1869:                 msg += f" {file_path}:\n"
1870:                 for v in file_violations[:3]:
1871:                     msg += f"     Line {v['line']}: {v['code']}\n"
1872:                     msg += f"     {v['pattern']}: {v['value']}\n"
1873:                 if len(file_violations) > 3:
1874:                     msg += f"     ... and {len(file_violations)} more\n"
1875:                 msg += "\n"
1876:
1877:             if len(by_file) > 5:
1878:                 msg += f"     ... and {len(by_file)} more files\n"
1879:
1880:             msg += "\nAll calibration parameters must be externalized to configuration.\n"
1881:
1882:         return msg
1883:
1884:     def test_calibration_registry_used(self, source_files):
1885:         """Verify calibration_registry is imported where needed"""
1886:         imports_found = 0
1887:
1888:         for file_path in source_files:
1889:             if "calibration" in str(file_path).lower():
1890:                 continue
1891:
1892:             try:
1893:                 content = file_path.read_text()
1894:
1895:                 if "calibration_registry" in content or "get_calibration" in content:
1896:                     imports_found += 1
1897:
1898:             except (UnicodeDecodeError, OSErr
1899:                     pass
1900:
1901:             print(f"\nFound {imports_found} files using calibration_registry")
1902:
1903:     def test_pre_commit_hook_exists(self):
```

```
1904:     """Verify pre-commit hook is installed"""
1905:     hook_path = Path(".git/hooks/pre-commit")
1906:     assert hook_path.exists(), "Pre-commit hook not installed"
1907:
1908:     content = hook_path.read_text()
1909:     assert "pre_commit_validators.py" in content, \
1910:         "Pre-commit hook does not call validators"
1911:
1912:     print("\n\u234\u223 Pre-commit hook is installed and configured")
1913:
1914: def test_pre_commitValidatorsExist(self):
1915:     """Verify pre-commit validator scripts exist"""
1916:     validators_path = Path("scripts/pre_commit_validators.py")
1917:     assert validators_path.exists(), "Pre-commit validators script missing"
1918:
1919:     update_hash_path = Path("scripts/update_hash_registry.py")
1920:     assert update_hash_path.exists(), "Hash registry update script missing"
1921:
1922:     print("\n\u234\u223 Pre-commit validator scripts exist")
1923:
1924: def test_noInlineBayesianPriors(self, source_files):
1925:     """Check for hardcoded Bayesian priors"""
1926:     violations = []
1927:     pattern = re.compile(r'prior\s*=\s*([0-9.]+)', re.IGNORECASE)
1928:
1929:     for file_path in source_files:
1930:         try:
1931:             content = file_path.read_text()
1932:             lines = content.split('\n')
1933:
1934:             for lineno, line in enumerate(lines, 1):
1935:                 if self._is_comment_or_docstring(line):
1936:                     continue
1937:
1938:                 if pattern.search(line):
1939:                     violations.append({
1940:                         'file': str(file_path),
1941:                         'line': lineno,
1942:                         'code': line.strip()
1943:                     })
1944:
1945:             except (UnicodeDecodeError, OSError):
1946:                 pass
1947:
1948:         if violations:
1949:             msg = f"\nFound {len(violations)} hardcoded prior values:\n"
1950:             for v in violations[:5]:
1951:                 msg += f"  {v['file']}:{v['line']}: {v['code']}\n"
1952:             print(msg)
1953:
1954:
1955:
1956: =====
1957: FILE: tests/calibration_system/test_orchestrator_runtime.py
1958: =====
1959:
```

```
1960: """
1961: Test 5: Orchestrator Runtime - Correct Layer Evaluation + Aggregation with Real Contexts
1962:
1963: Validates runtime behavior:
1964: - Layers execute in correct order
1965: - Method results aggregate properly
1966: - Context passing works correctly
1967: - No runtime exceptions in calibration resolution
1968:
1969: FAILURE CONDITION: Any runtime error OR incorrect aggregation = NOT READY
1970:
1971: DEPRECATED: Test assumes outdated executors_methods.json structure and layer execution order.
1972: See tests/DEPRECATED_TESTS.md for details.
1973: """
1974: import json
1975: import pytest
1976: from pathlib import Path
1977: from typing import Dict, List, Any
1978: from unittest.mock import Mock, patch, MagicMock
1979:
1980: pytestmark = pytest.mark.obsolete
1981:
1982:
1983: class TestOrchestratorRuntime:
1984:
1985:     @pytest.fixture(scope="class")
1986:     def executors_methods(self) -> Dict[str, Any]:
1987:         """Load executors_methods.json"""
1988:         path = Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
1989:         with open(path) as f:
1990:             return json.load(f)
1991:
1992:     @pytest.fixture(scope="class")
1993:     def intrinsic_calibration(self) -> Dict[str, Any]:
1994:         """Load intrinsic_calibration.json"""
1995:         path = Path("system/config/calibration/intrinsic_calibration.json")
1996:         with open(path) as f:
1997:             return json.load(f)
1998:
1999:     @pytest.fixture
2000:     def mock_context(self) -> Dict[str, Any]:
2001:         """Create a mock execution context"""
2002:         return {
2003:             "document_text": "Sample policy document for testing",
2004:             "raw_text": "Sample policy document for testing",
2005:             "document_id": "test_doc_001",
2006:             "metadata": {
2007:                 "source": "test",
2008:                 "year": 2024
2009:             }
2010:         }
2011:
2012:     def test_layer_execution_order_defined(self, executors_methods):
2013:         """Verify layer execution order is consistent"""
2014:         expected_order = [
2015:             "ingestion",
```

```
2016:         "extraction",
2017:         "transformation",
2018:         "validation",
2019:         "aggregation",
2020:         "scoring",
2021:         "reporting",
2022:         "meta"
2023:     ]
2024:
2025:     for executor in executors_methods:
2026:         layers_sequence = []
2027:
2028:         for method in executor["methods"]:
2029:             if "layer" in method:
2030:                 layer = method["layer"]
2031:                 if not layers_sequence or layers_sequence[-1] != layer:
2032:                     layers_sequence.append(layer)
2033:
2034:             for i, layer in enumerate(layers_sequence):
2035:                 expected_pos = expected_order.index(layer)
2036:
2037:                 for j in range(i + 1, len(layers_sequence)):
2038:                     next_layer = layers_sequence[j]
2039:                     next_pos = expected_order.index(next_layer)
2040:
2041:                     assert expected_pos <= next_pos, \
2042:                         f"Layer order violation in {executor['executor_id']}:\n" \
2043:                         f"{layer} (pos {expected_pos}) followed by {next_layer} (pos {next_pos})"
2044:
2045:     def test_calibration_context_resolution(self, intrinsic_calibration):
2046:         """Test that calibration context can be resolved without errors"""
2047:         try:
2048:             from src.farfan_pipeline.core.orchestrator.calibration_context import (
2049:                 CalibrationContext,
2050:                 resolve_contextual_calibration,
2051:             )
2052:         except ImportError:
2053:             pytest.skip("calibration_context module not available")
2054:
2055:         test_question_ids = ["D1Q1", "D3Q5", "D6Q3"]
2056:
2057:         for qid in test_question_ids:
2058:             try:
2059:                 context = CalibrationContext.from_question_id(qid)
2060:                 assert context.question_id == qid
2061:                 assert context.dimension > 0
2062:                 assert context.question_num > 0
2063:             except Exception as e:
2064:                 pytest.fail(f"Failed to create context for {qid}: {e}")
2065:
2066:     def test_method_execution_simulation(self, executors_methods, mock_context):
2067:         """Simulate method execution to verify no obvious runtime errors"""
2068:         sample_executor = executors_methods[0]
2069:
2070:         for method in sample_executor["methods"][:3]:
2071:             method_id = f"{method['class']}.{method['method']}
```

```
2072:
2073:         try:
2074:             result = self._simulate_method_call(
2075:                 method["class"],
2076:                 method["method"],
2077:                 mock_context
2078:             )
2079:
2080:             assert result is not None or result == {}, \
2081:                 f"Method {method_id} returned None unexpectedly"
2082:
2083:         except NotImplementedError:
2084:             pass
2085:         except Exception as e:
2086:             pytest.fail(
2087:                 f"Unexpected error simulating {method_id}: {type(e).__name__}: {e}"
2088:             )
2089:
2090:     def _simulate_method_call(
2091:         self, class_name: str, method_name: str, context: Dict[str, Any]
2092:     ) -> Any:
2093:         """Simulate a method call (mock implementation)"""
2094:         if method_name.startswith("_"):
2095:             return {}
2096:
2097:         return {"status": "simulated", "method": f"{class_name}.{method_name}"}
2098:
2099:     def test_aggregation_logic_validation(self, executors_methods):
2100:         """Verify aggregation layer methods are present"""
2101:         aggregation_methods = []
2102:
2103:         for executor in executors_methods:
2104:             for method in executor["methods"]:
2105:                 if method.get("layer") == "aggregation":
2106:                     aggregation_methods.append({
2107:                         "executor": executor["executor_id"],
2108:                         "method": f"{method['class']}.{method['method']}"
2109:                     })
2110:
2111:         assert len(aggregation_methods) > 0, \
2112:             "No aggregation methods found across all executors"
2113:
2114:         print(f"\nFound {len(aggregation_methods)} aggregation methods")
2115:
2116:     def test_scoring_methods_present(self, executors_methods):
2117:         """Verify scoring layer methods are present"""
2118:         scoring_methods = []
2119:
2120:         for executor in executors_methods:
2121:             for method in executor["methods"]:
2122:                 if method.get("layer") == "scoring":
2123:                     scoring_methods.append({
2124:                         "executor": executor["executor_id"],
2125:                         "method": f"{method['class']}.{method['method']}"
2126:                     })
2127:
```

```
2128:         assert len(scoring_methods) > 0, \
2129:             "No scoring methods found across all executors"
2130:
2131:         print(f"\nFound {len(scoring_methods)} scoring methods")
2132:
2133:     def test_reporting_layer_completeness(self, executors_methods):
2134:         """Verify all executors have reporting capability"""
2135:         executors_without_reporting = []
2136:
2137:         for executor in executors_methods:
2138:             has_reporting = any(
2139:                 method.get("layer") == "reporting"
2140:                 for method in executor["methods"]
2141:             )
2142:
2143:             if not has_reporting:
2144:                 executors_without_reporting.append(executor["executor_id"])
2145:
2146:         if executors_without_reporting:
2147:             print(
2148:                 f"\nWARNING: {len(executors_without_reporting)} executors "
2149:                 f"without reporting layer: {executors_without_reporting[:5]}"
2150:             )
2151:
2152:     def test_context_propagation_structure(self, mock_context):
2153:         """Verify context has expected structure for method execution"""
2154:         required_fields = ["document_text", "raw_text"]
2155:
2156:         for field in required_fields:
2157:             assert field in mock_context, \
2158:                 f"Mock context missing required field: {field}"
2159:
2160:     def test_no_circular_dependencies_in_layers(self, executors_methods):
2161:         """Verify no circular dependencies between layers"""
2162:         layer_dependencies = {}
2163:
2164:         for executor in executors_methods:
2165:             for i, method in enumerate(executor["methods"]):
2166:                 if "layer" not in method:
2167:                     continue
2168:
2169:                 current_layer = method["layer"]
2170:
2171:                 if current_layer not in layer_dependencies:
2172:                     layer_dependencies[current_layer] = set()
2173:
2174:                     for prev_method in executor["methods"][:i]:
2175:                         if "layer" in prev_method:
2176:                             layer_dependencies[current_layer].add(prev_method["layer"])
2177:
2178:         for layer, deps in layer_dependencies.items():
2179:             assert layer not in deps, \
2180:                 f"Circular dependency detected: layer {layer} depends on itself"
2181:
2182:     def test_calibration_values_resolvable(self, intrinsic_calibration):
2183:         """Verify calibration values can be accessed without errors"""
```

```
2184:         for method_id, calibration in intrinsic_calibration.items():
2185:             if method_id == "_metadata":
2186:                 continue
2187:
2188:             if calibration.get("status") == "computed":
2189:                 assert isinstance(calibration, dict), \
2190:                     f"Calibration for {method_id} must be a dict"
2191:
2192:
2193:
2194: =====
2195: FILE: tests/calibration_system/test_performance_benchmarks.py
2196: =====
2197:
2198: """
2199: Test 7: Performance Benchmarks - Load Times and Calibration Speed
2200:
2201: Validates system performance:
2202: - Load intrinsic.json: < 1 second
2203: - Calibrate 30 executors: < 5 seconds
2204: - Calibrate 200 methods: < 30 seconds
2205:
2206: FAILURE CONDITION: Any performance threshold exceeded = NOT READY
2207:
2208: DEPRECATED: Test uses hardcoded paths to system/config/calibration/intrinsic_calibration.json.
2209: See tests/DEPRECATED_TESTS.md for details.
2210: """
2211: import json
2212: import time
2213: import pytest
2214: from pathlib import Path
2215: from typing import Dict, Any, List
2216:
2217: pytestmark = pytest.mark.obsolete
2218:
2219:
2220: class TestPerformanceBenchmarks:
2221:
2222:     MAX_LOAD_TIME_SECONDS = 1.0
2223:     MAX_30_EXECUTORS_SECONDS = 5.0
2224:     MAX_200_METHODS_SECONDS = 30.0
2225:
2226:     @pytest.fixture(scope="class")
2227:     def intrinsic_path(self) -> Path:
2228:         """Path to intrinsic_calibration.json"""
2229:         return Path("system/config/calibration/intrinsic_calibration.json")
2230:
2231:     @pytest.fixture(scope="class")
2232:     def executors_path(self) -> Path:
2233:         """Path to executors_methods.json"""
2234:         return Path("src/farfan_pipeline/core/orchestrator/executors_methods.json")
2235:
2236:     def test_load_intrinsic_json_performance(self, intrinsic_path):
2237:         """CRITICAL: Load intrinsic_calibration.json in < 1 second"""
2238:         assert intrinsic_path.exists(), f"File not found: {intrinsic_path}"
2239:
```

```
2240:         start_time = time.time()
2241:
2242:         with open(intrinsic_path) as f:
2243:             data = json.load(f)
2244:
2245:         load_time = time.time() - start_time
2246:
2247:         assert load_time < self.MAX_LOAD_TIME_SECONDS, \
2248:             f"Loading intrinsic_calibration.json took {load_time:.3f}s, " \
2249:             f"exceeds limit of {self.MAX_LOAD_TIME_SECONDS}s"
2250:
2251:         print(f"\nLoaded intrinsic_calibration.json in {load_time:.3f}s")
2252:         print(f"File size: {intrinsic_path.stat().st_size / 1024:.1f} KB")
2253:         print(f"Number of methods: {len([k for k in data.keys() if k != '_metadata'])}")
2254:
2255:     def test_load_executors_methods_performance(self, executors_path):
2256:         """Load executors_methods.json performance"""
2257:         assert executors_path.exists(), f"File not found: {executors_path}"
2258:
2259:         start_time = time.time()
2260:
2261:         with open(executors_path) as f:
2262:             data = json.load(f)
2263:
2264:         load_time = time.time() - start_time
2265:
2266:         assert load_time < self.MAX_LOAD_TIME_SECONDS, \
2267:             f"Loading executors_methods.json took {load_time:.3f}s, " \
2268:             f"exceeds limit of {self.MAX_LOAD_TIME_SECONDS}s"
2269:
2270:         print(f"\nLoaded executors_methods.json in {load_time:.3f}s")
2271:         print(f"Number of executors: {len(data)}")
2272:
2273:     def test_calibrate_30_executors_performance(
2274:         self, intrinsic_path, executors_path
2275:     ):
2276:         """CRITICAL: Calibrate all 30 executors in < 5 seconds"""
2277:         with open(intrinsic_path) as f:
2278:             calibrations = json.load(f)
2279:
2280:             with open(executors_path) as f:
2281:                 executors = json.load(f)
2282:
2283:             assert len(executors) == 30, f"Expected 30 executors, found {len(executors)}"
2284:
2285:             start_time = time.time()
2286:
2287:             calibrated_count = 0
2288:             for executor in executors:
2289:                 for method in executor["methods"]:
2290:                     method_id = f"{method['class']}.{method['method']}"
2291:
2292:                     if method_id in calibrations:
2293:                         calibration = calibrations[method_id]
2294:
2295:                         _ = calibration.get("status")
```

```
2296:             _ = calibration.get("b_theory")
2297:             _ = calibration.get("b_impl")
2298:             _ = calibration.get("b_deploy")
2299:
2300:         calibrated_count += 1
2301:
2302:     calibration_time = time.time() - start_time
2303:
2304:     assert calibration_time < self.MAX_30_EXECUTORS_SECONDS, \
2305:           f"Calibrating 30 executors took {calibration_time:.3f}s, " \
2306:           f"exceeds limit of {self.MAX_30_EXECUTORS_SECONDS}s"
2307:
2308:     print(f"\nCalibrated {calibrated_count} methods across 30 executors in {calibration_time:.3f}s")
2309:     print(f"Average: {calibration_time / 30:.3f}s per executor")
2310:
2311: def test_calibrate_200_methods_performance(self, intrinsic_path):
2312:     """CRITICAL: Calibrate 200 methods in < 30 seconds"""
2313:     with open(intrinsic_path) as f:
2314:         calibrations = json.load(f)
2315:
2316:     methods = [k for k in calibrations.keys() if k != "_metadata"]
2317:
2318:     if len(methods) < 200:
2319:         pytest.skip(f"Only {len(methods)} methods available, need 200 for benchmark")
2320:
2321:     test_methods = methods[:200]
2322:
2323:     start_time = time.time()
2324:
2325:     for method_id in test_methods:
2326:         calibration = calibrations[method_id]
2327:
2328:         _ = calibration.get("status")
2329:         _ = calibration.get("b_theory", 0.0)
2330:         _ = calibration.get("b_impl", 0.0)
2331:         _ = calibration.get("b_deploy", 0.0)
2332:
2333:         if calibration.get("status") == "computed":
2334:             _ = calibration.get("evidence", {})
2335:
2336:     calibration_time = time.time() - start_time
2337:
2338:     assert calibration_time < self.MAX_200_METHODS_SECONDS, \
2339:       f"Calibrating 200 methods took {calibration_time:.3f}s, " \
2340:       f"exceeds limit of {self.MAX_200_METHODS_SECONDS}s"
2341:
2342:     print(f"\nCalibrated 200 methods in {calibration_time:.3f}s")
2343:     print(f"Average: {calibration_time / 200 * 1000:.3f}ms per method")
2344:
2345: def test_json_parsing_overhead(self, intrinsic_path):
2346:     """Measure JSON parsing overhead"""
2347:     content = intrinsic_path.read_text()
2348:
2349:     iterations = 10
2350:     start_time = time.time()
2351:
```

```
2352:     for _ in range(iterations):
2353:         _ = json.loads(content)
2354:
2355:     total_time = time.time() - start_time
2356:     avg_time = total_time / iterations
2357:
2358:     print(f"\nJSON parsing: {avg_time * 1000:.3f}ms per parse (avg of {iterations})")
2359:
2360: def test_calibration_lookup_performance(self, intrinsic_path):
2361:     """Measure calibration lookup performance"""
2362:     with open(intrinsic_path) as f:
2363:         calibrations = json.load(f)
2364:
2365:     methods = [k for k in calibrations.keys() if k != "_metadata"]
2366:
2367:     if not methods:
2368:         pytest.skip("No methods in calibration file")
2369:
2370:     iterations = 10000
2371:     test_method = methods[0]
2372:
2373:     start_time = time.time()
2374:
2375:     for _ in range(iterations):
2376:         _ = calibrations.get(test_method)
2377:
2378:     total_time = time.time() - start_time
2379:     avg_time = total_time / iterations
2380:
2381:     print(f"\nDictionary lookup: {avg_time * 1000000:.3f}µs per lookup (avg of {iterations})")
2382:
2383: def test_file_size_reasonable(self, intrinsic_path):
2384:     """Verify intrinsic_calibration.json size is reasonable"""
2385:     MAX_SIZE_MB = 10
2386:
2387:     size_bytes = intrinsic_path.stat().st_size
2388:     size_mb = size_bytes / (1024 * 1024)
2389:
2390:     assert size_mb < MAX_SIZE_MB, \
2391:         f"intrinsic_calibration.json is {size_mb:.2f}MB, exceeds {MAX_SIZE_MB}MB limit"
2392:
2393:     print(f"\nFile size: {size_mb:.2f}MB")
2394:
2395: def test_concurrent_access_simulation(self, intrinsic_path):
2396:     """Simulate concurrent access to calibration data"""
2397:     import threading
2398:
2399:     results = []
2400:
2401:     def load_calibrations():
2402:         with open(intrinsic_path) as f:
2403:             data = json.load(f)
2404:             results.append(len(data))
2405:
2406:     threads = []
2407:     num_threads = 5
```

```
2408:  
2409:     start_time = time.time()  
2410:  
2411:     for _ in range(num_threads):  
2412:         thread = threading.Thread(target=load_calibrations)  
2413:         threads.append(thread)  
2414:         thread.start()  
2415:  
2416:     for thread in threads:  
2417:         thread.join()  
2418:  
2419:     total_time = time.time() - start_time  
2420:  
2421:     assert len(results) == num_threads, \  
2422:         f"Expected {num_threads} results, got {len(results)}"  
2423:  
2424:     print(f"\n{n} concurrent loads completed in {total_time:.3f}s")  
2425:  
2426: def test_memory_footprint(self, intrinsic_path):  
2427:     """Estimate memory footprint of calibration data"""  
2428:     import sys  
2429:  
2430:     with open(intrinsic_path) as f:  
2431:         data = json.load(f)  
2432:  
2433:     size_estimate = sys.getsizeof(data)  
2434:  
2435:     for key, value in data.items():  
2436:         size_estimate += sys.getsizeof(key)  
2437:         size_estimate += sys.getsizeof(value)  
2438:  
2439:         if isinstance(value, dict):  
2440:             for k, v in value.items():  
2441:                 size_estimate += sys.getsizeof(k) + sys.getsizeof(v)  
2442:  
2443:     size_mb = size_estimate / (1024 * 1024)  
2444:  
2445:     print(f"\nEstimated memory footprint: {size_mb:.2f}MB")  
2446:  
2447:  
2448:  
2449: =====  
2450: FILE: tests/calibration_system/test_pre_commit_validators.py  
2451: =====  
2452:  
2453: """  
2454: Test Pre-Commit Validators Integration  
2455:  
2456: Tests the pre-commit hook system that enforces:  
2457: 1. No hardcoded calibration patterns  
2458: 2. JSON schema compliance  
2459: 3. SHA256 hash verification  
2460: 4. YAML prohibition  
2461: """  
2462:  
2463: import json
```

```
2464: import sys
2465: from pathlib import Path
2466:
2467: sys.path.insert(0, str(Path(__file__).parent.parent / "scripts"))
2468:
2469: from pre_commit_validators import (
2470:     ALLOWED_VALUES,
2471:     CALIBRATION_PATTERNS,
2472:     validate_json_schema,
2473:     validate_no_hardcoded_calibrations,
2474:     validate_no_yaml_files,
2475: )
2476:
2477:
2478: class TestPreCommitValidators:
2479:
2480:     def test_hardcoded_calibration_detection(self, tmp_path):
2481:         """Test detection of hardcoded calibration values"""
2482:         test_file = tmp_path / "test_module.py"
2483:         test_file.write_text(
2484:             """
2485: def calculate_score():
2486:     weight = 0.75
2487:     threshold = 0.8
2488:     score = 0.9
2489:     return weight * threshold * score
2490: """
2491:     )
2492:
2493:         is_valid, errors = validate_no_hardcoded_calibrations([test_file])
2494:
2495:         assert not is_valid, "Should detect hardcoded calibrations"
2496:         assert len(errors) > 0
2497:         assert any("weight" in str(e) for e in errors)
2498:         assert any("threshold" in str(e) for e in errors)
2499:         assert any("score" in str(e) for e in errors)
2500:
2501:     def test_allowed_values_pass(self, tmp_path):
2502:         """Test that allowed literal values pass validation"""
2503:         test_file = tmp_path / "test_module.py"
2504:         test_file.write_text(
2505:             """
2506: def initialize():
2507:     weight = 1.0
2508:     threshold = 0.5
2509:     score = 0
2510:     min_count = 2
2511:     max_count = 10
2512:     return weight, threshold, score
2513: """
2514:     )
2515:
2516:         is_valid, errors = validate_no_hardcoded_calibrations([test_file])
2517:
2518:         assert is_valid, f"Allowed values should pass: {errors}"
2519:
```

```
2520:     def test_comment_exclusion(self, tmp_path):
2521:         """Test that comments are excluded from validation"""
2522:         test_file = tmp_path / "test_module.py"
2523:         test_file.write_text(
2524:             """
2525: def example():
2526:     # weight = 0.75 # This is a comment
2527:     return load_weight_from_config()
2528: """
2529:     )
2530:
2531:     is_valid, errors = validate_no_hardcoded_calibrations([test_file])
2532:
2533:     assert is_valid, "Comments should be excluded"
2534:
2535:     def test_json_validation_valid(self, tmp_path):
2536:         """Test valid JSON passes validation"""
2537:         test_file = tmp_path / "config.json"
2538:         test_file.write_text(
2539:             json.dumps({"version": "1.0", "settings": {"key": "value"}}, indent=2)
2540:         )
2541:
2542:         is_valid, errors = validate_json_schema([test_file])
2543:
2544:         assert is_valid, f"Valid JSON should pass: {errors}"
2545:
2546:     def test_json_validation_invalid(self, tmp_path):
2547:         """Test invalid JSON is detected"""
2548:         test_file = tmp_path / "config.json"
2549:         test_file.write_text("{invalid json")
2550:
2551:         is_valid, errors = validate_json_schema([test_file])
2552:
2553:         assert not is_valid, "Invalid JSON should be detected"
2554:         assert len(errors) > 0
2555:
2556:     def test_yaml_prohibition(self, tmp_path):
2557:         """Test that YAML files are blocked"""
2558:         yaml_file = tmp_path / "config.yaml"
2559:         yaml_file.write_text("key: value")
2560:
2561:         is_valid, errors = validate_no_yaml_files([yaml_file])
2562:
2563:         assert not is_valid, "YAML files should be blocked"
2564:         assert len(errors) > 0
2565:         assert any("YAML" in str(e) or "yaml" in str(e) for e in errors)
2566:
2567:     def test_yml_extension_blocked(self, tmp_path):
2568:         """Test that .yml extension is also blocked"""
2569:         yml_file = tmp_path / "config.yml"
2570:         yml_file.write_text("key: value")
2571:
2572:         is_valid, errors = validate_no_yaml_files([yml_file])
2573:
2574:         assert not is_valid, ".yml files should be blocked"
2575:
```

```
2576:     def test_calibration_patterns_comprehensive(self):
2577:         """Test that all calibration patterns are defined"""
2578:         expected_params = [
2579:             "weight",
2580:             "score",
2581:             "threshold",
2582:             "min_evidence",
2583:             "max_evidence",
2584:             "confidence",
2585:             "penalty",
2586:             "tolerance",
2587:             "sensitivity",
2588:             "prior",
2589:         ]
2590:
2591:         pattern_params = [param for _, param in CALIBRATION_PATTERNS]
2592:
2593:         for param in expected_params:
2594:             assert (
2595:                 param in pattern_params
2596:             ), f"Calibration parameter '{param}' not in patterns"
2597:
2598:     def test_allowed_values_defined(self):
2599:         """Test that allowed literal values are properly defined"""
2600:         assert "0" in ALLOWED_VALUES
2601:         assert "1" in ALLOWED_VALUES
2602:         assert "0.0" in ALLOWED_VALUES
2603:         assert "1.0" in ALLOWED_VALUES
2604:         assert "0.5" in ALLOWED_VALUES
2605:         assert "-1" in ALLOWED_VALUES
2606:
2607:     def test_config_path_exclusion(self):
2608:         """Test that config/ paths are excluded from calibration checks"""
2609:         config_file = Path("config/test_settings.py")
2610:
2611:         if config_file.exists():
2612:             content = config_file.read_text()
2613:             original_content = content
2614:         else:
2615:             config_file.parent.mkdir(parents=True, exist_ok=True)
2616:             original_content = None
2617:
2618:         try:
2619:             config_file.write_text("weight = 0.75\nthreshold = 0.8\n")
2620:
2621:             is_valid, errors = validate_no_hardcoded_calibrations(
2622:                 [config_file.absolute()]
2623:             )
2624:
2625:             assert is_valid, "Config files should be excluded"
2626:         finally:
2627:             if original_content is not None:
2628:                 config_file.write_text(original_content)
2629:             elif config_file.exists():
2630:                 config_file.unlink()
2631:
```

```
2632:     def test_multiple_violations_reported(self, tmp_path):
2633:         """Test that multiple violations are all reported"""
2634:         test_file = tmp_path / "violations.py"
2635:         test_file.write_text(
2636:             """
2637: def bad_code():
2638:     weight = 0.75
2639:     score = 0.85
2640:     threshold = 0.9
2641:     penalty = 0.15
2642:     sensitivity = 0.65
2643:     return weight + score + threshold + penalty + sensitivity
2644: """
2645:     )
2646:
2647:     is_valid, errors = validate_no_hardcoded_calibrations([test_file])
2648:
2649:     assert not is_valid
2650:     error_text = "\n".join(errors)
2651:     assert "weight" in error_text
2652:     assert "score" in error_text
2653:     assert "threshold" in error_text
2654:     assert "penalty" in error_text
2655:     assert "sensitivity" in error_text
2656:
2657:     def test_string_literals_excluded(self, tmp_path):
2658:         """Test that string literals with patterns are excluded"""
2659:         test_file = tmp_path / "strings.py"
2660:         test_file.write_text(
2661:             """
2662: def example():
2663:     message = "weight = 0.75"
2664:     description = "Set threshold = 0.8 for best results"
2665:     return message, description
2666: """
2667:     )
2668:
2669:     is_valid, errors = validate_no_hardcoded_calibrations([test_file])
2670:
2671:     assert is_valid, "String literals should be excluded"
2672:
2673:     def test_pre_commit_hook_integration(self):
2674:         """Test that pre-commit hook file exists and is executable"""
2675:         hook_path = Path(".git/hooks/pre-commit")
2676:
2677:         assert hook_path.exists(), "Pre-commit hook not installed"
2678:
2679:         content = hook_path.read_text()
2680:         assert "pre_commit_validators.py" in content
2681:         assert "bash" in content or "sh" in content
2682:
2683:     def test_hash_registry_script_exists(self):
2684:         """Test that hash registry update script exists"""
2685:         script_path = Path("scripts/update_hash_registry.py")
2686:
2687:         assert script_path.exists(), "Hash registry script missing"
```

```
2688:
2689:         content = script_path.read_text()
2690:         assert "sha256" in content.lower()
2691:         assert "hash" in content.lower()
2692:
2693:     def test_calibration_registry_pattern_consistency(self):
2694:         """Test that patterns match calibration_registry.py structure"""
2695:         registry_path = Path(
2696:             "src/farfan_pipeline/core/calibration/calibration_registry.py"
2697:         )
2698:
2699:         if registry_path.exists():
2700:             content = registry_path.read_text()
2701:
2702:             for _pattern, param_name in CALIBRATION_PATTERNS:
2703:                 if param_name in ["weight", "score"]:
2704:                     assert (
2705:                         param_name in content
2706:                     ), f"Calibration registry should reference {param_name}"
2707:
2708:
2709:
2710: =====
2711: FILE: tests/core/orchestrator/test_chunk_matrix_builder.py
2712: =====
2713:
2714: """Unit tests for chunk matrix builder module."""
2715:
2716: import pytest
2717: from pathlib import Path
2718:
2719: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance
2720: from farfan_pipeline.core.orchestrator.chunk_matrix_builder import (
2721:     build_chunk_matrix,
2722:     EXPECTED_CHUNK_COUNT,
2723:     POLICY AREAS,
2724:     DIMENSIONS,
2725: )
2726:
2727:
2728: def create_chunk(
2729:     chunk_id: int,
2730:     policy_area_id: str,
2731:     dimension_id: str,
2732:     text: str = "Test chunk content",
2733: ) -> ChunkData:
2734:     """Helper to create a valid ChunkData instance."""
2735:     return ChunkData(
2736:         id=chunk_id,
2737:         text=text,
2738:         chunk_type="diagnostic",
2739:         sentences=[],
2740:         tables=[],
2741:         start_pos=0,
2742:         end_pos=len(text),
2743:         confidence=0.9,
```

```
2744:     chunk_id=f"policy_area_id}-{dimension_id}",
2745:     policy_area_id=policy_area_id,
2746:     dimension_id=dimension_id,
2747:     provenance=Provenance(page_number=1),
2748: )
2749:
2750:
2751: def create_full_document() -> PreprocessedDocument:
2752:     """Create a valid PreprocessedDocument with all 60 chunks."""
2753:     chunks = []
2754:     idx = 0
2755:     for pa in POLICY AREAS:
2756:         for dim in DIMENSIONS:
2757:             chunks.append(create_chunk(idx, pa, dim, f"Content for {pa}-{dim}"))
2758:             idx += 1
2759:
2760:     return PreprocessedDocument(
2761:         document_id="test-doc-001",
2762:         raw_text="Complete policy document text",
2763:         sentences=[],
2764:         tables=[],
2765:         metadata={"chunk_count": 60},
2766:         chunks=chunks,
2767:         processing_mode="chunked",
2768:     )
2769:
2770:
2771: def test_build_chunk_matrix_success():
2772:     """build_chunk_matrix should accept exactly 60 valid chunks."""
2773:     doc = create_full_document()
2774:     matrix, keys = build_chunk_matrix(doc)
2775:
2776:     assert len(matrix) == EXPECTED_CHUNK_COUNT
2777:     assert len(keys) == EXPECTED_CHUNK_COUNT
2778:     assert keys[0] == ("PA01", "DIM01")
2779:     assert keys[-1] == ("PA10", "DIM06")
2780:
2781:     chunk = matrix[("PA05", "DIM03")]
2782:     assert chunk.policy_area_id == "PA05"
2783:     assert chunk.dimension_id == "DIM03"
2784:     assert chunk.text == "Content for PA05-DIM03"
2785:
2786:
2787: def test_build_chunk_matrix_sorted_keys():
2788:     """Keys should be sorted deterministically by PA then DIM."""
2789:     doc = create_full_document()
2790:     _, keys = build_chunk_matrix(doc)
2791:
2792:     assert keys[0] == ("PA01", "DIM01")
2793:     assert keys[5] == ("PA01", "DIM06")
2794:     assert keys[6] == ("PA02", "DIM01")
2795:     assert keys[-1] == ("PA10", "DIM06")
2796:
2797:     for i in range(len(keys) - 1):
2798:         assert keys[i] < keys[i + 1], f"Keys not sorted: {keys[i]} >= {keys[i+1]}"
2799:
```

```
2800:  
2801: def test_build_chunk_matrix_rejects_59_chunks():  
2802:     """build_chunk_matrix should reject documents with 59 chunks."""  
2803:     chunks = []  
2804:     idx = 0  
2805:     for pa in POLICY AREAS:  
2806:         for dim in DIMENSIONS:  
2807:             if pa == "PA10" and dim == "DIM06":  
2808:                 break  
2809:             chunks.append(create_chunk(idx, pa, dim))  
2810:             idx += 1  
2811:  
2812:     doc = PreprocessedDocument(  
2813:         document_id="test-doc-002",  
2814:         raw_text="Incomplete document",  
2815:         sentences=[],  
2816:         tables=[],  
2817:         metadata={},  
2818:         chunks=chunks,  
2819:     )  
2820:  
2821:     with pytest.raises(ValueError, match=r"Missing chunk combinations: .*PA10', 'DIM06' .*"):  
2822:         build_chunk_matrix(doc)  
2823:  
2824:  
2825: def test_build_chunk_matrix_rejects_61_chunks():  
2826:     """build_chunk_matrix should reject documents with 61 chunks."""  
2827:     chunks = []  
2828:     idx = 0  
2829:     for pa in POLICY AREAS:  
2830:         for dim in DIMENSIONS:  
2831:             chunks.append(create_chunk(idx, pa, dim))  
2832:             idx += 1  
2833:  
2834:     chunks.append(create_chunk(idx, "PA01", "DIM01", "Duplicate chunk"))  
2835:  
2836:     doc = PreprocessedDocument(  
2837:         document_id="test-doc-003",  
2838:         raw_text="Document with duplicate",  
2839:         sentences=[],  
2840:         tables=[],  
2841:         metadata={},  
2842:         chunks=chunks,  
2843:     )  
2844:  
2845:     with pytest.raises(ValueError, match="Duplicate.*PA01-DIM01"):  
2846:         build_chunk_matrix(doc)  
2847:  
2848:  
2849: def test_build_chunk_matrix_detects_duplicate_keys():  
2850:     """build_chunk_matrix should reject duplicate (PA, DIM) combinations."""  
2851:     chunks = []  
2852:     idx = 0  
2853:     for pa in POLICY AREAS:  
2854:         for dim in DIMENSIONS:  
2855:             if pa == "PA05" and dim == "DIM03":
```

```
2856:             continue
2857:             chunks.append(create_chunk(idx, pa, dim))
2858:             idx += 1
2859:
2860:             chunks.append(create_chunk(idx, "PA01", "DIM01", "Duplicate chunk"))
2861:
2862:             doc = PreprocessedDocument(
2863:                 document_id="test-doc-004",
2864:                 raw_text="Document with duplicate key",
2865:                 sentences=[],
2866:                 tables=[],
2867:                 metadata={},
2868:                 chunks=chunks,
2869:             )
2870:
2871:             with pytest.raises(ValueError, match="Duplicate.*PA01-DIM01"):
2872:                 build_chunk_matrix(doc)
2873:
2874:
2875: def test_build_chunk_matrix_detects_missing_combinations():
2876:     """build_chunk_matrix should detect missing PA\227DIM combinations."""
2877:     chunks = []
2878:     idx = 0
2879:     for pa in POLICY AREAS:
2880:         for dim in DIMENSIONS:
2881:             if pa == "PA05" and dim == "DIM03":
2882:                 continue
2883:             chunks.append(create_chunk(idx, pa, dim))
2884:             idx += 1
2885:
2886:     chunks.append(create_chunk(idx, "PA05", "DIM04"))
2887:
2888:     doc = PreprocessedDocument(
2889:         document_id="test-doc-005",
2890:         raw_text="Document with duplicate and missing",
2891:         sentences=[],
2892:         tables=[],
2893:         metadata={},
2894:         chunks=chunks,
2895:     )
2896:
2897:     with pytest.raises(ValueError, match="Duplicate.*PA05-DIM04"):
2898:         build_chunk_matrix(doc)
2899:
2900:
2901: def test_build_chunk_matrix_reports_specific_missing_key():
2902:     """build_chunk_matrix should report specific missing PA-DIM combination."""
2903:     chunks = []
2904:     idx = 0
2905:     for pa in POLICY AREAS:
2906:         for dim in DIMENSIONS:
2907:             if pa == "PA05" and dim == "DIM03":
2908:                 continue
2909:             if pa == "PA07" and dim == "DIM02":
2910:                 continue
2911:             chunks.append(create_chunk(idx, pa, dim))
```

```
2912:             idx += 1
2913:
2914:     doc = PreprocessedDocument(
2915:         document_id="test-doc-006",
2916:         raw_text="Document missing PA05-DIM03 and PA07-DIM02",
2917:         sentences=[],
2918:         tables=[],
2919:         metadata={},
2920:         chunks=chunks,
2921:     )
2922:
2923:     with pytest.raises(ValueError) as exc_info:
2924:         build_chunk_matrix(doc)
2925:     assert (
2926:         "Missing chunk combinations: [('PA05', 'DIM03'), ('PA07', 'DIM02')]" in str(exc_info.value)
2927:     )
2928:
2929:
2930:
2931: def test_build_chunk_matrix_rejects_null_policy_area():
2932:     """build_chunk_matrix should reject chunks with null policy_area_id."""
2933:     chunks = []
2934:     idx = 0
2935:     for pa in POLICY AREAS:
2936:         for dim in DIMENSIONS:
2937:             if pa == "PA01" and dim == "DIM01":
2938:                 chunk = ChunkData(
2939:                     id=idx,
2940:                     text="Test",
2941:                     chunk_type="diagnostic",
2942:                     sentences=[],
2943:                     tables=[],
2944:                     start_pos=0,
2945:                     end_pos=4,
2946:                     confidence=0.9,
2947:                     chunk_id="PA01-DIM01",
2948:                     policy_area_id=None,
2949:                     dimension_id=dim,
2950:                 )
2951:                 chunks.append(chunk)
2952:             else:
2953:                 chunks.append(create_chunk(idx, pa, dim))
2954:             idx += 1
2955:
2956:     doc = PreprocessedDocument(
2957:         document_id="test-doc-007",
2958:         raw_text="Document with null policy_area_id",
2959:         sentences=[],
2960:         tables=[],
2961:         metadata={},
2962:         chunks=chunks,
2963:     )
2964:
2965:     with pytest.raises(ValueError, match="null policy_area_id"):
2966:         build_chunk_matrix(doc)
2967:
```

```
2968:
2969: def test_build_chunk_matrix_rejects_null_dimension():
2970:     """build_chunk_matrix should reject chunks with null dimension_id."""
2971:     chunks = []
2972:     idx = 0
2973:     for pa in POLICY AREAS:
2974:         for dim in DIMENSIONS:
2975:             if pa == "PA01" and dim == "DIM01":
2976:                 chunk = ChunkData(
2977:                     id=idx,
2978:                     text="Test",
2979:                     chunk_type="diagnostic",
2980:                     sentences=[],
2981:                     tables=[],
2982:                     start_pos=0,
2983:                     end_pos=4,
2984:                     confidence=0.9,
2985:                     chunk_id="PA01-DIM01",
2986:                     policy_area_id=pa,
2987:                     dimension_id=None,
2988:                 )
2989:                 chunks.append(chunk)
2990:             else:
2991:                 chunks.append(create_chunk(idx, pa, dim))
2992:             idx += 1
2993:
2994:     doc = PreprocessedDocument(
2995:         document_id="test-doc-008",
2996:         raw_text="Document with null dimension_id",
2997:         sentences=[],
2998:         tables=[],
2999:         metadata={},
3000:         chunks=chunks,
3001:     )
3002:
3003:     with pytest.raises(ValueError, match="null dimension_id"):
3004:         build_chunk_matrix(doc)
3005:
3006:
3007: def test_build_chunk_matrix_rejects_invalid_policy_area():
3008:     """build_chunk_matrix should reject invalid policy area format."""
3009:     with pytest.raises(ValueError, match="Invalid chunk_id 'PA00-DIM01'"):
3010:         create_chunk(0, "PA00", "DIM01")
3011:
3012:
3013: def test_build_chunk_matrix_rejects_invalid_dimension():
3014:     """build_chunk_matrix should reject invalid dimension format."""
3015:     with pytest.raises(ValueError, match="Invalid chunk_id 'PA01-DIM07'"):
3016:         create_chunk(0, "PA01", "DIM07")
3017:
3018:
3019: def test_build_chunk_matrix_rejects_malformed_chunk_id():
3020:     """build_chunk_matrix should reject malformed chunk IDs."""
3021:     with pytest.raises(ValueError, match="Invalid chunk_id 'P01-DIM01'"):
3022:         create_chunk(0, "P01", "DIM01")
3023:
```

```
3024:
3025: def test_build_chunk_matrix_preserves_content():
3026:     """build_chunk_matrix should preserve chunk content and metadata."""
3027:     doc = create_full_document()
3028:     matrix, _ = build_chunk_matrix(doc)
3029:
3030:     chunk = matrix[("PA07", "DIM04")]
3031:     assert chunk.text == "Content for PA07-DIM04"
3032:     assert chunk.policy_area_id == "PA07"
3033:     assert chunk.dimension_id == "DIM04"
3034:     assert chunk.chunk_type == "diagnostic"
3035:     assert chunk.confidence == 0.9
3036:     assert chunk.provenance is not None
3037:     assert chunk.provenance.page_number == 1
3038:
3039:
3040: def test_build_chunk_matrix_deterministic_ordering():
3041:     """Multiple calls should produce identical key ordering."""
3042:     doc = create_full_document()
3043:
3044:     _, keys1 = build_chunk_matrix(doc)
3045:     _, keys2 = build_chunk_matrix(doc)
3046:
3047:     assert keys1 == keys2
3048:
3049:
3050: def test_build_chunk_matrix_all_combinations_present():
3051:     """Matrix should contain all 60 PA\227DIM combinations."""
3052:     doc = create_full_document()
3053:     matrix, _ = build_chunk_matrix(doc)
3054:
3055:     for pa in POLICY AREAS:
3056:         for dim in DIMENSIONS:
3057:             assert (pa, dim) in matrix
3058:             chunk = matrix[(pa, dim)]
3059:             assert chunk.policy_area_id == pa
3060:             assert chunk.dimension_id == dim
3061:
3062:
3063:
3064: =====
3065: FILE: tests/core/orchestrator/test_contract_verification.py
3066: =====
3067:
3068: """Tests for pre-execution contract verification in BaseExecutorWithContract.
3069:
3070: This test module validates the contract verification functionality that ensures
3071: all 30 base executor contracts are valid before execution begins.
3072: """
3073: from __future__ import annotations
3074:
3075: import pytest
3076: from pathlib import Path
3077: from unittest.mock import MagicMock, patch
3078:
3079: from farfan_pipeline.core.orchestrator.base_executor_with_contract import (
```

```
3080:     BaseExecutorWithContract,
3081: )
3082:
3083:
3084: class TestContractVerification:
3085:     """Test suite for contract verification functionality."""
3086:
3087:     def test_verify_all_base_contracts_calls_verify_single(self):
3088:         """Test that verify_all_base_contracts attempts to verify all 30 contracts."""
3089:         with patch.object(
3090:             BaseExecutorWithContract,
3091:             "_verify_single_contract",
3092:             return_value={
3093:                 "passed": True,
3094:                 "errors": [],
3095:                 "warnings": [],
3096:                 "contract_version": "v3",
3097:                 "contract_path": "/fake/path",
3098:             },
3099:         ) as mock_verify:
3100:             result = BaseExecutorWithContract.verify_all_base_contracts(
3101:                 class_registry={}
3102:             )
3103:
3104:             assert mock_verify.call_count == 30
3105:             assert result["total_contracts"] == 30
3106:             assert result["passed"] is True
3107:             assert len(result["verified_contracts"]) == 30
3108:
3109:     def test_verify_all_base_contracts_with_errors(self):
3110:         """Test that errors are accumulated correctly."""
3111:
3112:         def mock_verify_single(base_slot, class_registry=None):
3113:             if base_slot in ["D1-Q1", "D2-Q2"]:
3114:                 return {
3115:                     "passed": False,
3116:                     "errors": [f"Error in {base_slot}"],
3117:                     "warnings": [],
3118:                     "contract_version": "v3",
3119:                     "contract_path": "/fake/path",
3120:                 }
3121:             return {
3122:                 "passed": True,
3123:                 "errors": [],
3124:                 "warnings": [],
3125:                 "contract_version": "v3",
3126:                 "contract_path": "/fake/path",
3127:             }
3128:
3129:             with patch.object(
3130:                 BaseExecutorWithContract,
3131:                 "_verify_single_contract",
3132:                 side_effect=mock_verify_single,
3133:             ):
3134:                 result = BaseExecutorWithContract.verify_all_base_contracts(
3135:                     class_registry={}
```

```
3136:         )
3137:
3138:         assert result["total_contracts"] == 30
3139:         assert result["passed"] is False
3140:         assert len(result["errors"]) == 2
3141:         assert len(result["verified_contracts"]) == 28
3142:         assert any("D1-Q1" in err for err in result["errors"])
3143:         assert any("D2-Q2" in err for err in result["errors"])
3144:
3145:     def test_verify_all_base_contracts_caches_result(self):
3146:         """Test that subsequent calls use cached result."""
3147:         BaseExecutorWithContract._factory_contracts_verified = False
3148:         BaseExecutorWithContract._factory_verification_errors = []
3149:
3150:         with patch.object(
3151:             BaseExecutorWithContract,
3152:             "_verify_single_contract",
3153:             return_value={
3154:                 "passed": True,
3155:                 "errors": [],
3156:                 "warnings": [],
3157:                 "contract_version": "v3",
3158:                 "contract_path": "/fake/path",
3159:             },
3160:         ) as mock_verify:
3161:             result1 = BaseExecutorWithContract.verify_all_base_contracts(
3162:                 class_registry={}
3163:             )
3164:             result2 = BaseExecutorWithContract.verify_all_base_contracts(
3165:                 class_registry={}
3166:             )
3167:
3168:             assert mock_verify.call_count == 30
3169:             assert result1 == result2
3170:
3171:     def test_verify_single_contract_missing_file(self):
3172:         """Test verification fails when contract file doesn't exist."""
3173:         BaseExecutorWithContract._factory_contracts_verified = False
3174:
3175:         result = BaseExecutorWithContract._verify_single_contract(
3176:             "D99-Q99", class_registry={}
3177:         )
3178:
3179:         assert result["passed"] is False
3180:         assert len(result["errors"]) > 0
3181:         assert any("not found" in err.lower() for err in result["errors"])
3182:         assert result["contract_path"] is None
3183:
3184:     def test_verify_v2_contract_fields(self):
3185:         """Test v2 contract field validation."""
3186:         valid_contract = {
3187:             "method_inputs": [
3188:                 {"class": "TestClass", "method": "test_method"}
3189:             ],
3190:             "assembly_rules": [],
3191:             "validation_rules": [],
```

```
3192:         }
3193:
3194:     errors = BaseExecutorWithContract._verify_v2_contract_fields(
3195:         valid_contract, "D1-Q1", class_registry={"TestClass": object}
3196:     )
3197:     assert len(errors) == 0
3198:
3199: def test_verify_v2_contract_missing_fields(self):
3200:     """Test v2 contract validation catches missing fields."""
3201:     invalid_contract = {
3202:         "method_inputs": []
3203:     }
3204:
3205:     errors = BaseExecutorWithContract._verify_v2_contract_fields(
3206:         invalid_contract, "D1-Q1", class_registry={}
3207:     )
3208:
3209:     assert len(errors) >= 2
3210:     assert any("assembly_rules" in err for err in errors)
3211:     assert any("validation_rules" in err for err in errors)
3212:
3213: def test_verify_v2_contract_method_class_not_in_registry(self):
3214:     """Test v2 contract validation catches missing method classes."""
3215:     contract = {
3216:         "method_inputs": [
3217:             {"class": "MissingClass", "method": "test_method"}
3218:         ],
3219:         "assembly_rules": [],
3220:         "validation_rules": []
3221:     }
3222:
3223:     errors = BaseExecutorWithContract._verify_v2_contract_fields(
3224:         contract, "D1-Q1", class_registry={}
3225:     )
3226:
3227:     assert len(errors) >= 1
3228:     assert any("MissingClass" in err and "not found" in err for err in errors)
3229:
3230: def test_verify_v3_contract_fields(self):
3231:     """Test v3 contract field validation."""
3232:     valid_contract = {
3233:         "identity": {
3234:             "base_slot": "D1-Q1"
3235:         },
3236:         "method_binding": {
3237:             "orchestration_mode": "single_method",
3238:             "class_name": "TestClass",
3239:             "method_name": "test_method",
3240:         },
3241:         "evidence_assembly": {
3242:             "assembly_rules": []
3243:         },
3244:         "validation_rules": {},
3245:         "question_context": {
3246:             "expected_elements": []
3247:         },
3248:         "expected_elements": []
3249:     }
```

```
3248:         "error_handling": {},
3249:     }
3250:
3251:     errors = BaseExecutorWithContract._verify_v3_contract_fields(
3252:         valid_contract, "D1-Q1", class_registry={"TestClass": object}
3253:     )
3254:     assert len(errors) == 0
3255:
3256:     def test_verify_v3_contract_missing_identity(self):
3257:         """Test v3 contract validation catches missing identity."""
3258:         invalid_contract = {
3259:             "method_binding": {},
3260:             "evidence_assembly": {"assembly_rules": []},
3261:             "validation_rules": {},
3262:             "question_context": {"expected_elements": []},
3263:             "error_handling": {}
3264:         }
3265:
3266:         errors = BaseExecutorWithContract._verify_v3_contract_fields(
3267:             invalid_contract, "D1-Q1", class_registry={}
3268:         )
3269:
3270:         assert any("identity" in err.lower() for err in errors)
3271:
3272:     def test_verify_v3_contract_base_slot_mismatch(self):
3273:         """Test v3 contract validation catches base_slot mismatch."""
3274:         contract = {
3275:             "identity": {
3276:                 "base_slot": "D2-Q1"
3277:             },
3278:             "method_binding": {
3279:                 "class_name": "TestClass",
3280:                 "method_name": "test_method",
3281:             },
3282:             "evidence_assembly": {"assembly_rules": []},
3283:             "validation_rules": {},
3284:             "question_context": {"expected_elements": []},
3285:             "error_handling": {}
3286:         }
3287:
3288:         errors = BaseExecutorWithContract._verify_v3_contract_fields(
3289:             contract, "D1-Q1", class_registry={"TestClass": object}
3290:         )
3291:
3292:         assert any("mismatch" in err.lower() and "D1-Q1" in err for err in errors)
3293:
3294:     def test_verify_v3_contract_multi_method_pipeline(self):
3295:         """Test v3 contract validation for multi_method_pipeline mode."""
3296:         contract = {
3297:             "identity": {"base_slot": "D1-Q1"},
3298:             "method_binding": {
3299:                 "orchestration_mode": "multi_method_pipeline",
3300:                 "methods": [
3301:                     {"class_name": "Class1", "method_name": "method1"},
3302:                     {"class_name": "Class2", "method_name": "method2"},
3303:                 ]
3304:             }
3305:         }
```

```
3304:         },
3305:         "evidence_assembly": {"assembly_rules": []},
3306:         "validation_rules": {},
3307:         "question_context": {"expected_elements": []},
3308:         "error_handling": {},
3309:     }
3310:
3311:     errors = BaseExecutorWithContract._verify_v3_contract_fields(
3312:         contract,
3313:         "D1-Q1",
3314:         class_registry={"Class1": object, "Class2": object}
3315:     )
3316:
3317:     assert len(errors) == 0
3318:
3319: def test_verify_v3_contract_multi_method_missing_class(self):
3320:     """Test v3 multi-method validation catches missing classes."""
3321:     contract = {
3322:         "identity": {"base_slot": "D1-Q1"},
3323:         "method_binding": {
3324:             "orchestration_mode": "multi_method_pipeline",
3325:             "methods": [
3326:                 {"class_name": "Class1", "method_name": "method1"},
3327:                 {"class_name": "MissingClass", "method_name": "method2"},
3328:             ]
3329:         },
3330:         "evidence_assembly": {"assembly_rules": []},
3331:         "validation_rules": {},
3332:         "question_context": {"expected_elements": []},
3333:         "error_handling": {}
3334:     }
3335:
3336:     errors = BaseExecutorWithContract._verify_v3_contract_fields(
3337:         contract,
3338:         "D1-Q1",
3339:         class_registry={"Class1": object}
3340:     )
3341:
3342:     assert any("MissingClass" in err and "not found" in err for err in errors)
3343:
3344: def test_verify_v3_contract_missing_expected_elements(self):
3345:     """Test v3 contract validation catches missing expected_elements."""
3346:     contract = {
3347:         "identity": {"base_slot": "D1-Q1"},
3348:         "method_binding": {
3349:             "class_name": "TestClass",
3350:             "method_name": "test_method",
3351:         },
3352:         "evidence_assembly": {"assembly_rules": []},
3353:         "validation_rules": {},
3354:         "question_context": {},
3355:         "error_handling": {}
3356:     }
3357:
3358:     errors = BaseExecutorWithContract._verify_v3_contract_fields(
3359:         contract, "D1-Q1", class_registry={"TestClass": object}
```

```
3360:         )
3361:
3362:         assert any("expected_elements" in err for err in errors)
3363:
3364:     def test_base_slot_to_q_number_calculation(self):
3365:         """Test that base slot to Q number calculation is correct."""
3366:         test_cases = [
3367:             ("D1-Q1", "Q001"),
3368:             ("D1-Q5", "Q005"),
3369:             ("D2-Q1", "Q006"),
3370:             ("D3-Q3", "Q013"),
3371:             ("D6-Q5", "Q030"),
3372:         ]
3373:
3374:         for base_slot, expected_q_id in test_cases:
3375:             dimension = int(base_slot[1])
3376:             question = int(base_slot[4])
3377:             q_number = (dimension - 1) * 5 + question
3378:             q_id = f"Q{q_number:03d}"
3379:             assert q_id == expected_q_id, f"Failed for {base_slot}: got {q_id}, expected {expected_q_id}"
3380:
3381:
3382:
3383: =====
3384: FILE: tests/core/test_chunk_data_extensions.py
3385: =====
3386:
3387: """
3388: Tests for ChunkData extended fields and validation.
3389:
3390: Validates expected_elements schema expectations and document_position
3391: byte offset tracking with comprehensive __post_init__ validation.
3392: """
3393:
3394: import logging
3395:
3396: import pytest
3397:
3398: from farfan_pipeline.core.types import ChunkData
3399:
3400:
3401: @pytest.fixture
3402: def minimal_valid_chunk():
3403:     """Minimal valid chunk with empty expected_elements and no document_position."""
3404:     return ChunkData(
3405:         id=100,
3406:         text="Minimal chunk text",
3407:         chunk_type="diagnostic",
3408:         sentences=[],
3409:         tables=[],
3410:         start_pos=0,
3411:         end_pos=50,
3412:         confidence=0.9,
3413:         policy_area_id="PA01",
3414:         dimension_id="DIM01",
3415:     )
```

```
3416:  
3417:  
3418: @pytest.fixture  
3419: def chunk_with_position():  
3420:     """Chunk with document_position range."""  
3421:     return ChunkData(  
3422:         id=101,  
3423:         text="Chunk with position",  
3424:         chunk_type="activity",  
3425:         sentences=[0],  
3426:         tables=[],  
3427:         start_pos=0,  
3428:         end_pos=20,  
3429:         confidence=0.85,  
3430:         policy_area_id="PA02",  
3431:         dimension_id="DIM02",  
3432:         document_position=(1000, 2000),  
3433:     )  
3434:  
3435:  
3436: @pytest.fixture  
3437: def chunk_with_expected_elements():  
3438:     """Chunk with populated expected_elements schema."""  
3439:     return ChunkData(  
3440:         id=102,  
3441:         text="Chunk with schema expectations",  
3442:         chunk_type="indicator",  
3443:         sentences=[0, 1],  
3444:         tables=[],  
3445:         start_pos=0,  
3446:         end_pos=30,  
3447:         confidence=0.92,  
3448:         policy_area_id="PA03",  
3449:         dimension_id="DIM03",  
3450:         expected_elements=[  
3451:             {"type": "table", "required": True, "minimum": 1},  
3452:             {"type": "numeric_data", "required": False},  
3453:             {"type": "citation", "minimum": 0},  
3454:         ],  
3455:     )  
3456:  
3457:  
3458: @pytest.fixture  
3459: def fully_populated_chunk():  
3460:     """Chunk with both expected_elements and document_position."""  
3461:     return ChunkData(  
3462:         id=103,  
3463:         text="Fully populated chunk",  
3464:         chunk_type="resource",  
3465:         sentences=[0, 1, 2],  
3466:         tables=[0],  
3467:         start_pos=0,  
3468:         end_pos=25,  
3469:         confidence=0.95,  
3470:         policy_area_id="PA04",  
3471:         dimension_id="DIM04",
```

```
3472:         expected_elements=[
3473:             {"type": "budget_allocation", "required": True, "minimum": 1},
3474:             {"type": "timeframe"}, 
3475:         ],
3476:         document_position=(5000, 5500),
3477:     )
3478:
3479:
3480: class TestChunkDataValidFields:
3481:     """Test suite for valid ChunkData configurations."""
3482:
3483:     def test_minimal_valid_chunk(self, minimal_valid_chunk):
3484:         """Minimal chunk with empty fields should be valid."""
3485:         assert minimal_valid_chunk.expected_elements == []
3486:         assert minimal_valid_chunk.document_position is None
3487:         assert minimal_valid_chunk.text == "Minimal chunk text"
3488:
3489:     def test_chunk_with_position_range(self, chunk_with_position):
3490:         """Chunk with valid position range should be accepted."""
3491:         assert chunk_with_position.document_position == (1000, 2000)
3492:         assert chunk_with_position.text == "Chunk with position"
3493:
3494:     def test_chunk_with_expected_elements_schema(self, chunk_with_expected_elements):
3495:         """Chunk with populated expected_elements should be valid."""
3496:         assert len(chunk_with_expected_elements.expected_elements) == 3
3497:         assert chunk_with_expected_elements.expected_elements[0]["type"] == "table"
3498:         assert chunk_with_expected_elements.expected_elements[0]["required"] is True
3499:         assert chunk_with_expected_elements.expected_elements[0]["minimum"] == 1
3500:         assert (
3501:             chunk_with_expected_elements.expected_elements[1]["type"] == "numeric_data"
3502:         )
3503:         assert chunk_with_expected_elements.expected_elements[1]["required"] is False
3504:         assert chunk_with_expected_elements.expected_elements[2]["type"] == "citation"
3505:         assert chunk_with_expected_elements.expected_elements[2]["minimum"] == 0
3506:
3507:     def test_fully_populated_chunk_both_fields(self, fully_populated_chunk):
3508:         """Chunk with both new fields populated should be valid."""
3509:         assert len(fully_populated_chunk.expected_elements) == 2
3510:         assert fully_populated_chunk.document_position == (5000, 5500)
3511:         assert fully_populated_chunk.expected_elements[0]["type"] == "budget_allocation"
3512:         assert fully_populated_chunk.expected_elements[1]["type"] == "timeframe"
3513:
3514:     def test_zero_length_position_triggers_warning(self, caplog):
3515:         """Zero-length position range should trigger warning log entry."""
3516:         with caplog.at_level(logging.WARNING):
3517:             chunk = ChunkData(
3518:                 id=999,
3519:                 text="Test zero-length position",
3520:                 chunk_type="diagnostic",
3521:                 sentences=[],
3522:                 tables=[],
3523:                 start_pos=0,
3524:                 end_pos=26,
3525:                 confidence=0.8,
3526:                 policy_area_id="PA01",
3527:                 dimension_id="DIM01",
```

```
3528:             document_position=(1500, 1500),
3529:         )
3530:     assert "zero-length document_position" in caplog.text
3531:     assert chunk.document_position == (1500, 1500)
3532:
3533:
3534: class TestChunkDataTextValidation:
3535:     """Test suite for text field validation."""
3536:
3537:     def test_empty_text_raises_error(self):
3538:         """Empty text should raise ValueError."""
3539:         with pytest.raises(ValueError, match="text cannot be empty"):
3540:             ChunkData(
3541:                 id=1,
3542:                 text="",
3543:                 chunk_type="diagnostic",
3544:                 sentences=[],
3545:                 tables=[],
3546:                 start_pos=0,
3547:                 end_pos=10,
3548:                 confidence=0.9,
3549:                 policy_area_id="PA01",
3550:                 dimension_id="DIM01",
3551:             )
3552:
3553:     def test_whitespace_only_text_raises_error(self):
3554:         """Whitespace-only text should raise ValueError."""
3555:         with pytest.raises(ValueError, match="text cannot be empty"):
3556:             ChunkData(
3557:                 id=1,
3558:                 text=" \n\t ",
3559:                 chunk_type="diagnostic",
3560:                 sentences=[],
3561:                 tables=[],
3562:                 start_pos=0,
3563:                 end_pos=10,
3564:                 confidence=0.9,
3565:                 policy_area_id="PA01",
3566:                 dimension_id="DIM01",
3567:             )
3568:
3569:
3570: class TestChunkDataDocumentPositionValidation:
3571:     """Test suite for document_position field validation."""
3572:
3573:     def test_inverted_position_range_raises_error(self):
3574:         """Position range with end < start should raise ValueError."""
3575:         with pytest.raises(ValueError, match="end offset.*must be >= start offset"):
3576:             ChunkData(
3577:                 id=1,
3578:                 text="Valid text",
3579:                 chunk_type="diagnostic",
3580:                 sentences=[],
3581:                 tables=[],
3582:                 start_pos=0,
3583:                 end_pos=10,
```

```
3584:             confidence=0.9,
3585:             policy_area_id="PA01",
3586:             dimension_id="DIM01",
3587:             document_position=(2000, 1000),
3588:         )
3589:
3590:     def test_negative_start_offset_raises_error(self):
3591:         """Negative start offset should raise ValueError."""
3592:         with pytest.raises(ValueError, match="start offset must be non-negative"):
3593:             ChunkData(
3594:                 id=1,
3595:                 text="Valid text",
3596:                 chunk_type="diagnostic",
3597:                 sentences=[],
3598:                 tables=[],
3599:                 start_pos=0,
3600:                 end_pos=10,
3601:                 confidence=0.9,
3602:                 policy_area_id="PA01",
3603:                 dimension_id="DIM01",
3604:                 document_position=(-100, 1000),
3605:             )
3606:
3607:     def test_non_tuple_document_position_raises_error(self):
3608:         """Non-tuple document_position should raise ValueError."""
3609:         with pytest.raises(ValueError, match="document_position must be a tuple"):
3610:             ChunkData(
3611:                 id=1,
3612:                 text="Valid text",
3613:                 chunk_type="diagnostic",
3614:                 sentences=[],
3615:                 tables=[],
3616:                 start_pos=0,
3617:                 end_pos=10,
3618:                 confidence=0.9,
3619:                 policy_area_id="PA01",
3620:                 dimension_id="DIM01",
3621:                 document_position=[1000, 2000],
3622:             )
3623:
3624:     def test_wrong_arity_document_position_raises_error(self):
3625:         """document_position with != 2 elements should raise ValueError."""
3626:         with pytest.raises(ValueError, match="must have exactly 2 elements"):
3627:             ChunkData(
3628:                 id=1,
3629:                 text="Valid text",
3630:                 chunk_type="diagnostic",
3631:                 sentences=[],
3632:                 tables=[],
3633:                 start_pos=0,
3634:                 end_pos=10,
3635:                 confidence=0.9,
3636:                 policy_area_id="PA01",
3637:                 dimension_id="DIM01",
3638:                 document_position=(1000, 2000, 3000),
3639:             )
```

```
3640:
3641:     def test_non_integer_position_start_raises_error(self):
3642:         """Non-integer start position element should raise ValueError."""
3643:         with pytest.raises(ValueError, match="start.*must be an integer"):
3644:             ChunkData(
3645:                 id=1,
3646:                 text="Valid text",
3647:                 chunk_type="diagnostic",
3648:                 sentences=[],
3649:                 tables=[],
3650:                 start_pos=0,
3651:                 end_pos=10,
3652:                 confidence=0.9,
3653:                 policy_area_id="PA01",
3654:                 dimension_id="DIM01",
3655:                 document_position=("1000", 2000),
3656:             )
3657:
3658:     def test_non_integer_position_end_raises_error(self):
3659:         """Non-integer end position element should raise ValueError."""
3660:         with pytest.raises(ValueError, match="end.*must be an integer"):
3661:             ChunkData(
3662:                 id=1,
3663:                 text="Valid text",
3664:                 chunk_type="diagnostic",
3665:                 sentences=[],
3666:                 tables=[],
3667:                 start_pos=0,
3668:                 end_pos=10,
3669:                 confidence=0.9,
3670:                 policy_area_id="PA01",
3671:                 dimension_id="DIM01",
3672:                 document_position=(1000, 2000.5),
3673:             )
3674:
3675:
3676: class TestChunkDataExpectedElementsValidation:
3677:     """Test suite for expected_elements field validation."""
3678:
3679:     def test_non_list_expected_elements_raises_error(self):
3680:         """Non-list expected_elements should raise ValueError."""
3681:         with pytest.raises(ValueError, match="expected_elements must be a list"):
3682:             ChunkData(
3683:                 id=1,
3684:                 text="Valid text",
3685:                 chunk_type="diagnostic",
3686:                 sentences=[],
3687:                 tables=[],
3688:                 start_pos=0,
3689:                 end_pos=10,
3690:                 confidence=0.9,
3691:                 policy_area_id="PA01",
3692:                 dimension_id="DIM01",
3693:                 expected_elements="not a list",
3694:             )
3695:
```

```
3696:     def test_non_dict_expected_element_raises_error(self):
3697:         """Non-dict element in expected_elements should raise ValueError."""
3698:         with pytest.raises(ValueError, match="must be a dict"):
3699:             ChunkData(
3700:                 id=1,
3701:                 text="Valid text",
3702:                 chunk_type="diagnostic",
3703:                 sentences=[],
3704:                 tables=[],
3705:                 start_pos=0,
3706:                 end_pos=10,
3707:                 confidence=0.9,
3708:                 policy_area_id="PA01",
3709:                 dimension_id="DIM01",
3710:                 expected_elements=["not a dict"],
3711:             )
3712:
3713:     def test_missing_type_key_raises_error(self):
3714:         """expected_elements dict without 'type' key should raise ValueError."""
3715:         with pytest.raises(ValueError, match="missing required 'type' key"):
3716:             ChunkData(
3717:                 id=1,
3718:                 text="Valid text",
3719:                 chunk_type="diagnostic",
3720:                 sentences=[],
3721:                 tables=[],
3722:                 start_pos=0,
3723:                 end_pos=10,
3724:                 confidence=0.9,
3725:                 policy_area_id="PA01",
3726:                 dimension_id="DIM01",
3727:                 expected_elements=[{"required": True}],
3728:             )
3729:
3730:     def test_non_string_type_value_raises_error(self):
3731:         """Non-string 'type' value should raise ValueError."""
3732:         with pytest.raises(ValueError, match=r"\['type'\] must be a string"):
3733:             ChunkData(
3734:                 id=1,
3735:                 text="Valid text",
3736:                 chunk_type="diagnostic",
3737:                 sentences=[],
3738:                 tables=[],
3739:                 start_pos=0,
3740:                 end_pos=10,
3741:                 confidence=0.9,
3742:                 policy_area_id="PA01",
3743:                 dimension_id="DIM01",
3744:                 expected_elements=[{"type": 123}],
3745:             )
3746:
3747:     def test_non_boolean_required_value_raises_error(self):
3748:         """Non-boolean 'required' value should raise ValueError."""
3749:         with pytest.raises(ValueError, match=r"\['required'\] must be a boolean"):
3750:             ChunkData(
3751:                 id=1,
```

```
3752:             text="Valid text",
3753:             chunk_type="diagnostic",
3754:             sentences=[],
3755:             tables=[],
3756:             start_pos=0,
3757:             end_pos=10,
3758:             confidence=0.9,
3759:             policy_area_id="PA01",
3760:             dimension_id="DIM01",
3761:             expected_elements=[{"type": "table", "required": "yes"}],
3762:         )
3763:
3764:     def test_non_integer_minimum_value_raises_error(self):
3765:         """Non-integer 'minimum' value should raise ValueError."""
3766:         with pytest.raises(ValueError, match=r"\['minimum'\] must be an integer"):
3767:             ChunkData(
3768:                 id=1,
3769:                 text="Valid text",
3770:                 chunk_type="diagnostic",
3771:                 sentences=[],
3772:                 tables=[],
3773:                 start_pos=0,
3774:                 end_pos=10,
3775:                 confidence=0.9,
3776:                 policy_area_id="PA01",
3777:                 dimension_id="DIM01",
3778:                 expected_elements=[{"type": "table", "minimum": 1.5}],
3779:             )
3780:
3781:     def test_negative_minimum_value_raises_error(self):
3782:         """Negative 'minimum' value should raise ValueError."""
3783:         with pytest.raises(ValueError, match=r"\['minimum'\] must be non-negative"):
3784:             ChunkData(
3785:                 id=1,
3786:                 text="Valid text",
3787:                 chunk_type="diagnostic",
3788:                 sentences=[],
3789:                 tables=[],
3790:                 start_pos=0,
3791:                 end_pos=10,
3792:                 confidence=0.9,
3793:                 policy_area_id="PA01",
3794:                 dimension_id="DIM01",
3795:                 expected_elements=[{"type": "table", "minimum": -1}],
3796:             )
3797:
3798:
3799: class TestChunkDataMultipleElementValidation:
3800:     """Test validation across multiple expected_elements."""
3801:
3802:     def test_multiple_valid_elements(self):
3803:         """Multiple valid expected_elements should all be accepted."""
3804:         chunk = ChunkData(
3805:             id=200,
3806:             text="Complex schema",
3807:             chunk_type="indicator",
```

```
3808:         sentences=[],
3809:         tables=[],
3810:         start_pos=0,
3811:         end_pos=15,
3812:         confidence=0.9,
3813:         policy_area_id="PA05",
3814:         dimension_id="DIM05",
3815:         expected_elements=[
3816:             {"type": "table", "required": True, "minimum": 2},
3817:             {"type": "figure", "required": False, "minimum": 0},
3818:             {"type": "citation", "minimum": 1},
3819:             {"type": "formula"}},
3820:         ],
3821:     )
3822:     assert len(chunk.expected_elements) == 4
3823:
3824: def test_error_in_second_element_reports_correct_index(self):
3825:     """Error in second element should report index 1."""
3826:     with pytest.raises(ValueError, match=r"expected_elements\[1\]"):
3827:         ChunkData(
3828:             id=1,
3829:             text="Valid text",
3830:             chunk_type="diagnostic",
3831:             sentences=[],
3832:             tables=[],
3833:             start_pos=0,
3834:             end_pos=10,
3835:             confidence=0.9,
3836:             policy_area_id="PA01",
3837:             dimension_id="DIM01",
3838:             expected_elements=[
3839:                 {"type": "valid_type"},
3840:                 {"type": 123}, # Invalid
3841:             ],
3842:         )
3843:
3844:
3845: class TestChunkDataFieldInteraction:
3846:     """Test interaction between new fields and existing validation."""
3847:
3848:     def test_new_fields_with_invalid_chunk_id_still_fails(self):
3849:         """Invalid chunk_id should still fail even with valid new fields."""
3850:         with pytest.raises(ValueError, match="chunk_id"):
3851:             ChunkData(
3852:                 id=1,
3853:                 text="Valid text",
3854:                 chunk_type="diagnostic",
3855:                 sentences=[],
3856:                 tables=[],
3857:                 start_pos=0,
3858:                 end_pos=10,
3859:                 confidence=0.9,
3860:                 policy_area_id="PA99",
3861:                 dimension_id="DIM99",
3862:                 expected_elements=[{"type": "table"}],
3863:                 document_position=(1000, 2000),
```

```
3864:         )
3865:
3866:     def test_new_fields_preserved_after_chunk_id_derivation(self):
3867:         """New fields should be preserved when chunk_id is derived."""
3868:         chunk = ChunkData(
3869:             id=1,
3870:             text="Valid text",
3871:             chunk_type="diagnostic",
3872:             sentences=[],
3873:             tables=[],
3874:             start_pos=0,
3875:             end_pos=10,
3876:             confidence=0.9,
3877:             policy_area_id="PA01",
3878:             dimension_id="DIM01",
3879:             expected_elements=[{"type": "table"}],
3880:             document_position=(1000, 2000),
3881:         )
3882:         assert chunk.chunk_id == "PA01-DIM01"
3883:         assert len(chunk.expected_elements) == 1
3884:         assert chunk.document_position == (1000, 2000)
3885:
3886:
3887:
3888: =====
3889: FILE: tests/core/test_chunk_matrix_builder.py
3890: =====
3891:
3892: """
3893: Comprehensive test suite for ChunkMatrix validation and construction.
3894:
3895: Tests cover:
3896: - Deterministic ordering (multi-run stability)
3897: - 60-chunk invariant enforcement
3898: - Missing/duplicate chunk detection
3899: - chunk_id validation (PA01-PA10, DIM01-DIM06 format)
3900: - ValueError message verification
3901: - Audit log structure validation
3902: - Property-based tests using Hypothesis
3903: """
3904:
3905: from datetime import datetime
3906: from typing import Any
3907:
3908: import pytest
3909: from hypothesis import HealthCheck, given, settings
3910: from hypothesis import strategies as st
3911:
3912: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
3913: from farfan_pipeline.synchronization.irrigation_synchronizer import ChunkMatrix
3914:
3915:
3916: def create_chunk(
3917:     chunk_id: int,
3918:     policy_area_id: str,
3919:     dimension_id: str,
```

```
3920:     text: str = "test content",
3921:     chunk_type: str = "diagnostic",
3922: ) -> ChunkData:
3923:     """Factory for creating test chunks with configurable attributes."""
3924:     return ChunkData(
3925:         id=chunk_id,
3926:         text=text,
3927:         chunk_type=chunk_type, # type: ignore[arg-type]
3928:         sentences=[],
3929:         tables=[],
3930:         start_pos=0,
3931:         end_pos=len(text),
3932:         confidence=0.95,
3933:         policy_area_id=policy_area_id,
3934:         dimension_id=dimension_id,
3935:     )
3936:
3937:
3938: def create_complete_document(chunk_order: str = "sequential") -> PreprocessedDocument:
3939:     """Create a valid document with all 60 required chunks.
3940:
3941:     Args:
3942:         chunk_order: How to order chunks - 'sequential', 'reversed', or 'shuffled'
3943:     """
3944:     chunks = []
3945:     chunk_id = 0
3946:     for pa_num in range(1, 11):
3947:         for dim_num in range(1, 7):
3948:             pa_id = f"PA{pa_num:02d}"
3949:             dim_id = f"DIM{dim_num:02d}"
3950:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
3951:             chunk_id += 1
3952:
3953:     if chunk_order == "reversed":
3954:         chunks = list(reversed(chunks))
3955:     elif chunk_order == "shuffled":
3956:         import random
3957:
3958:         random.seed(42)
3959:         random.shuffle(chunks)
3960:
3961:     return PreprocessedDocument(
3962:         document_id="test-doc",
3963:         raw_text="test",
3964:         sentences=[],
3965:         tables=[],
3966:         metadata={},
3967:         chunks=chunks,
3968:         ingested_at=datetime.now(),
3969:     )
3970:
3971:
3972: class TestChunkMatrixDeterministicOrdering:
3973:     """Test deterministic ordering and multi-run stability."""
3974:
3975:     def test_matrix_construction_is_deterministic_sequential_order(self) -> None:
```

```
3976:     """ChunkMatrix construction should be deterministic across multiple runs with sequential input."""
3977:     doc = create_complete_document("sequential")
3978:
3979:     matrices = [ChunkMatrix(doc) for _ in range(5)]
3980:
3981:     for pa_num in range(1, 11):
3982:         for dim_num in range(1, 7):
3983:             pa_id = f"PA{pa_num:02d}"
3984:             dim_id = f"DIM{dim_num:02d}"
3985:
3986:             chunks = [m.get_chunk(pa_id, dim_id) for m in matrices]
3987:             first_chunk = chunks[0]
3988:
3989:             for chunk in chunks[1:]:
3990:                 assert chunk.id == first_chunk.id
3991:                 assert chunk.text == first_chunk.text
3992:                 assert chunk.policy_area_id == first_chunk.policy_area_id
3993:                 assert chunk.dimension_id == first_chunk.dimension_id
3994:
3995:     def test_matrix_construction_is_deterministic_reversed_order(self) -> None:
3996:         """ChunkMatrix should produce same results regardless of input chunk ordering."""
3997:         doc_sequential = create_complete_document("sequential")
3998:         doc_reversed = create_complete_document("reversed")
3999:
4000:         matrix_seq = ChunkMatrix(doc_sequential)
4001:         matrix_rev = ChunkMatrix(doc_reversed)
4002:
4003:         for pa_num in range(1, 11):
4004:             for dim_num in range(1, 7):
4005:                 pa_id = f"PA{pa_num:02d}"
4006:                 dim_id = f"DIM{dim_num:02d}"
4007:
4008:                 chunk_seq = matrix_seq.get_chunk(pa_id, dim_id)
4009:                 chunk_rev = matrix_rev.get_chunk(pa_id, dim_id)
4010:
4011:                 assert chunk_seq.policy_area_id == chunk_rev.policy_area_id
4012:                 assert chunk_seq.dimension_id == chunk_rev.dimension_id
4013:
4014:     def test_matrix_construction_is_deterministic_shuffled_order(self) -> None:
4015:         """ChunkMatrix should handle shuffled input order deterministically."""
4016:         doc_sequential = create_complete_document("sequential")
4017:         doc_shuffled = create_complete_document("shuffled")
4018:
4019:         matrix_seq = ChunkMatrix(doc_sequential)
4020:         matrix_shuf = ChunkMatrix(doc_shuffled)
4021:
4022:         for pa_num in range(1, 11):
4023:             for dim_num in range(1, 7):
4024:                 pa_id = f"PA{pa_num:02d}"
4025:                 dim_id = f"DIM{dim_num:02d}"
4026:
4027:                 chunk_seq = matrix_seq.get_chunk(pa_id, dim_id)
4028:                 chunk_shuf = matrix_shuf.get_chunk(pa_id, dim_id)
4029:
4030:                 assert chunk_seq.policy_area_id == chunk_shuf.policy_area_id
4031:                 assert chunk_seq.dimension_id == chunk_shuf.dimension_id
```

```
4032:
4033:     def test_multi_run_stability_with_identical_input(self) -> None:
4034:         """Multiple ChunkMatrix constructions from same document should be identical."""
4035:         doc = create_complete_document()
4036:
4037:         runs = 10
4038:         matrices = [ChunkMatrix(doc) for _ in range(runs)]
4039:
4040:         test_keys = [("PA01", "DIM01"), ("PA05", "DIM03"), ("PA10", "DIM06")]
4041:
4042:         for pa_id, dim_id in test_keys:
4043:             chunks = [m.get_chunk(pa_id, dim_id) for m in matrices]
4044:             first = chunks[0]
4045:
4046:             for chunk in chunks[1:]:
4047:                 assert chunk.id == first.id
4048:                 assert chunk.policy_area_id == first.policy_area_id
4049:                 assert chunk.dimension_id == first.dimension_id
4050:
4051:
4052: class TestSixtyChunkInvariantEnforcement:
4053:     """Test strict enforcement of 60-chunk requirement."""
4054:
4055:     def test_accepts_exactly_60_chunks(self) -> None:
4056:         """ChunkMatrix should accept exactly 60 valid chunks."""
4057:         doc = create_complete_document()
4058:         matrix = ChunkMatrix(doc)
4059:
4060:         chunk = matrix.get_chunk("PA01", "DIM01")
4061:         assert chunk.policy_area_id == "PA01"
4062:         assert chunk.dimension_id == "DIM01"
4063:
4064:     def test_rejects_59_chunks(self) -> None:
4065:         """ChunkMatrix should reject documents with 59 chunks by detecting missing combination."""
4066:         doc = create_complete_document()
4067:         doc.chunks = doc.chunks[:59]
4068:
4069:         with pytest.raises(ValueError) as exc_info:
4070:             ChunkMatrix(doc)
4071:
4072:         error_msg = str(exc_info.value)
4073:         assert "Missing chunk combinations" in error_msg
4074:
4075:     def test_rejects_61_chunks(self) -> None:
4076:         """ChunkMatrix should reject documents with 61 chunks."""
4077:         doc = create_complete_document()
4078:         first_chunk = doc.chunks[0]
4079:         assert first_chunk.policy_area_id is not None
4080:         assert first_chunk.dimension_id is not None
4081:         duplicate_chunk = create_chunk(
4082:             60, first_chunk.policy_area_id, first_chunk.dimension_id
4083:         )
4084:         doc.chunks.append(duplicate_chunk)
4085:
4086:         with pytest.raises(ValueError) as exc_info:
4087:             ChunkMatrix(doc)
```

```
4088:  
4089:         assert "Duplicate (PA, DIM) combination detected" in str(exc_info.value)  
4090:  
4091:     def test_rejects_0_chunks(self) -> None:  
4092:         """ChunkMatrix should reject documents with no chunks."""  
4093:         doc = PreprocessedDocument(  
4094:             document_id="test-doc",  
4095:             raw_text="test",  
4096:             sentences=[],  
4097:             tables=[],  
4098:             metadata={},  
4099:             chunks=[],  
4100:             ingested_at=datetime.now(),  
4101:         )  
4102:  
4103:         with pytest.raises(ValueError) as exc_info:  
4104:             ChunkMatrix(doc)  
4105:  
4106:             error_msg = str(exc_info.value)  
4107:             assert "Missing chunk combinations" in error_msg  
4108:  
4109:     def test_rejects_arbitrary_incorrect_count(self) -> None:  
4110:         """ChunkMatrix should reject any count other than 60."""  
4111:         for count in [1, 10, 30, 45, 58, 59, 62, 70]:  
4112:             if count > 60:  
4113:                 continue  
4114:             doc = create_complete_document()  
4115:             doc.chunks = doc.chunks[:count]  
4116:  
4117:             with pytest.raises(ValueError) as exc_info:  
4118:                 ChunkMatrix(doc)  
4119:  
4120:                 error_msg = str(exc_info.value)  
4121:                 assert (  
4122:                     "Missing chunk combinations" in error_msg  
4123:                     or "Chunk Matrix Invariant Violation" in error_msg  
4124:                 )  
4125:  
4126:  
4127: class TestMissingChunkDetection:  
4128:     """Test detection and reporting of missing chunk combinations."""  
4129:  
4130:     def test_detects_single_missing_chunk(self) -> None:  
4131:         """ChunkMatrix should detect and report a single missing PA-DIM combination."""  
4132:         chunks = []  
4133:         chunk_id = 0  
4134:         for pa_num in range(1, 11):  
4135:             for dim_num in range(1, 7):  
4136:                 if pa_num == 5 and dim_num == 3:  
4137:                     continue  
4138:                 pa_id = f"PA{pa_num:02d}"  
4139:                 dim_id = f"DIM{dim_num:02d}"  
4140:                 chunks.append(create_chunk(chunk_id, pa_id, dim_id))  
4141:                 chunk_id += 1  
4142:  
4143:         doc = PreprocessedDocument(
```

```
4144:         document_id="test-doc",
4145:         raw_text="test",
4146:         sentences=[],
4147:         tables=[],
4148:         metadata={},
4149:         chunks=chunks,
4150:         ingested_at=datetime.now(),
4151:     )
4152:
4153:     with pytest.raises(ValueError) as exc_info:
4154:         ChunkMatrix(doc)
4155:
4156:     error_msg = str(exc_info.value)
4157:     assert "Missing chunk combinations" in error_msg
4158:     assert "PA05", 'DIM03' in error_msg
4159:
4160:     def test_detects_multiple_missing_chunks(self) -> None:
4161:         """ChunkMatrix should detect and report multiple missing combinations."""
4162:         chunks = []
4163:         chunk_id = 0
4164:         missing_keys = [("PA02", "DIM01"), ("PA07", "DIM04"), ("PA10", "DIM06")]
4165:
4166:         for pa_num in range(1, 11):
4167:             for dim_num in range(1, 7):
4168:                 pa_id = f"PA{pa_num:02d}"
4169:                 dim_id = f"DIM{dim_num:02d}"
4170:                 if (pa_id, dim_id) in missing_keys:
4171:                     continue
4172:                 chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4173:                 chunk_id += 1
4174:
4175:         doc = PreprocessedDocument(
4176:             document_id="test-doc",
4177:             raw_text="test",
4178:             sentences=[],
4179:             tables=[],
4180:             metadata={},
4181:             chunks=chunks,
4182:             ingested_at=datetime.now(),
4183:         )
4184:
4185:         with pytest.raises(ValueError) as exc_info:
4186:             ChunkMatrix(doc)
4187:
4188:         error_msg = str(exc_info.value)
4189:         assert "Missing chunk combinations" in error_msg
4190:         for pa_id, dim_id in missing_keys:
4191:             assert f"{pa_id}-{dim_id}" in error_msg
4192:
4193:     def test_missing_chunks_reported_in_sorted_order(self) -> None:
4194:         """Missing chunk combinations should be reported in sorted order."""
4195:         chunks = []
4196:         chunk_id = 0
4197:         missing_keys = [("PA10", "DIM06"), ("PA01", "DIM01"), ("PA05", "DIM03")]
4198:
4199:         for pa_num in range(1, 11):
```

```
4200:         for dim_num in range(1, 7):
4201:             pa_id = f"PA(pa_num:02d)"
4202:             dim_id = f"DIM{dim_num:02d}"
4203:             if (pa_id, dim_id) in missing_keys:
4204:                 continue
4205:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4206:             chunk_id += 1
4207:
4208:     doc = PreprocessedDocument(
4209:         document_id="test-doc",
4210:         raw_text="test",
4211:         sentences=[],
4212:         tables=[],
4213:         metadata={},
4214:         chunks=chunks,
4215:         ingested_at=datetime.now(),
4216:     )
4217:
4218:     with pytest.raises(ValueError) as exc_info:
4219:         ChunkMatrix(doc)
4220:
4221:     error_msg = str(exc_info.value)
4222:     assert "Missing chunk combinations" in error_msg
4223:
4224:     missing_str = error_msg.split("Missing chunk combinations: ")[1]
4225:     missing_items = missing_str.strip("[]")
4226:     missing_list = [m.strip() for m in missing_items.split(", ")]
4227:     assert missing_list == sorted(missing_list)
4228:
4229:
4230: class TestDuplicateChunkDetection:
4231:     """Test detection and reporting of duplicate chunk combinations."""
4232:
4233:     def test_detects_duplicate_pa_dim_combination(self) -> None:
4234:         """ChunkMatrix should detect duplicate PA-DIM combinations."""
4235:         chunks = []
4236:         chunks.append(create_chunk(0, "PA01", "DIM01"))
4237:         chunks.append(create_chunk(1, "PA01", "DIM01"))
4238:
4239:         chunk_id = 2
4240:         for pa_num in range(1, 11):
4241:             for dim_num in range(1, 7):
4242:                 if pa_num == 1 and dim_num == 1:
4243:                     continue
4244:                 pa_id = f"PA(pa_num:02d)"
4245:                 dim_id = f"DIM{dim_num:02d}"
4246:                 chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4247:                 chunk_id += 1
4248:
4249:         doc = PreprocessedDocument(
4250:             document_id="test-doc",
4251:             raw_text="test",
4252:             sentences=[],
4253:             tables=[],
4254:             metadata={},
4255:             chunks=chunks,
```

```
4256:             ingested_at=datetime.now(),
4257:         )
4258:
4259:     with pytest.raises(ValueError) as exc_info:
4260:         ChunkMatrix(doc)
4261:
4262:     error_msg = str(exc_info.value)
4263:     assert "Duplicate" in error_msg
4264:     assert "PA01-DIM01" in error_msg
4265:
4266: def test_detects_duplicate_with_60_chunks(self) -> None:
4267:     """ChunkMatrix should detect duplicates even when chunk count is 60."""
4268:     doc = create_complete_document()
4269:     doc.chunks[0] = create_chunk(0, "PA01", "DIM02")
4270:
4271:     with pytest.raises(ValueError) as exc_info:
4272:         ChunkMatrix(doc)
4273:
4274:     error_msg = str(exc_info.value)
4275:     assert (
4276:         "Duplicate" in error_msg
4277:         or "Missing chunk combinations" in error_msg
4278:     )
4279:
4280: def test_detects_multiple_duplicates(self) -> None:
4281:     """ChunkMatrix should detect first duplicate in sequence."""
4282:     chunks = []
4283:     chunks.append(create_chunk(0, "PA01", "DIM01"))
4284:     chunks.append(create_chunk(1, "PA01", "DIM01"))
4285:     chunks.append(create_chunk(2, "PA02", "DIM02"))
4286:     chunks.append(create_chunk(3, "PA02", "DIM02"))
4287:
4288:     chunk_id = 4
4289:     for pa_num in range(1, 11):
4290:         for dim_num in range(1, 7):
4291:             if (pa_num == 1 and dim_num == 1) or (pa_num == 2 and dim_num == 2):
4292:                 continue
4293:             pa_id = f"PA{pa_num:02d}"
4294:             dim_id = f"DIM{dim_num:02d}"
4295:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4296:             chunk_id += 1
4297:
4298:     doc = PreprocessedDocument(
4299:         document_id="test-doc",
4300:         raw_text="test",
4301:         sentences=[],
4302:         tables=[],
4303:         metadata={},
4304:         chunks=chunks,
4305:         ingested_at=datetime.now(),
4306:     )
4307:
4308:     with pytest.raises(ValueError) as exc_info:
4309:         ChunkMatrix(doc)
4310:
4311:     error_msg = str(exc_info.value)
```

```
4312:         assert "Duplicate" in error_msg
4313:
4314:
4315: class TestChunkIdValidation:
4316:     """Test validation of chunk_id format (PA01-PA10, DIM01-DIM06)."""
4317:
4318:     def test_accepts_valid_pa_range(self) -> None:
4319:         """ChunkMatrix should accept all valid PA values (PA01-PA10)."""
4320:         for pa_num in range(1, 11):
4321:             pa_id = f"PA{pa_num:02d}"
4322:             chunks = []
4323:             chunk_id = 0
4324:
4325:             for dim_num in range(1, 7):
4326:                 dim_id = f"DIM{dim_num:02d}"
4327:                 chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4328:                 chunk_id += 1
4329:
4330:             for other_pa_num in range(1, 11):
4331:                 if other_pa_num == pa_num:
4332:                     continue
4333:                 other_pa_id = f"PA{other_pa_num:02d}"
4334:                 for dim_num in range(1, 7):
4335:                     dim_id = f"DIM{dim_num:02d}"
4336:                     chunks.append(create_chunk(chunk_id, other_pa_id, dim_id))
4337:                     chunk_id += 1
4338:
4339:             doc = PreprocessedDocument(
4340:                 document_id="test-doc",
4341:                 raw_text="test",
4342:                 sentences=[],
4343:                 tables=[],
4344:                 metadata={},
4345:                 chunks=chunks,
4346:                 ingested_at=datetime.now(),
4347:             )
4348:
4349:             matrix = ChunkMatrix(doc)
4350:             assert matrix.get_chunk(pa_id, "DIM01").policy_area_id == pa_id
4351:
4352:     def test_accepts_valid_dim_range(self) -> None:
4353:         """ChunkMatrix should accept all valid DIM values (DIM01-DIM06)."""
4354:         for dim_num in range(1, 7):
4355:             dim_id = f"DIM{dim_num:02d}"
4356:             doc = create_complete_document()
4357:             matrix = ChunkMatrix(doc)
4358:
4359:             for pa_num in range(1, 11):
4360:                 pa_id = f"PA{pa_num:02d}"
4361:                 assert matrix.get_chunk(pa_id, dim_id).dimension_id == dim_id
4362:
4363:     def test_rejects_pa00(self) -> None:
4364:         """ChunkData should reject PA00 at creation time."""
4365:         with pytest.raises(ValueError) as exc_info:
4366:             create_chunk(0, "PA00", "DIM01")
4367:
```

```
4368:         error_msg = str(exc_info.value)
4369:         assert "Invalid chunk_id" in error_msg
4370:         assert "PA00-DIM01" in error_msg
4371:
4372:     def test_rejects_pa11(self) -> None:
4373:         """ChunkData should reject PA11 at creation time."""
4374:         with pytest.raises(ValueError) as exc_info:
4375:             create_chunk(0, "PA11", "DIM01")
4376:
4377:             error_msg = str(exc_info.value)
4378:             assert "Invalid chunk_id" in error_msg
4379:             assert "PA11-DIM01" in error_msg
4380:
4381:     def test_rejects_dim00(self) -> None:
4382:         """ChunkData should reject DIM00 at creation time."""
4383:         with pytest.raises(ValueError) as exc_info:
4384:             create_chunk(0, "PA01", "DIM00")
4385:
4386:             error_msg = str(exc_info.value)
4387:             assert "Invalid chunk_id" in error_msg
4388:             assert "PA01-DIM00" in error_msg
4389:
4390:     def test_rejects_dim07(self) -> None:
4391:         """ChunkData should reject DIM07 at creation time."""
4392:         with pytest.raises(ValueError) as exc_info:
4393:             create_chunk(0, "PA01", "DIM07")
4394:
4395:             error_msg = str(exc_info.value)
4396:             assert "Invalid chunk_id" in error_msg
4397:             assert "PA01-DIM07" in error_msg
4398:
4399:     def test_rejects_malformed_pa_format(self) -> None:
4400:         """ChunkData should reject malformed PA identifiers at creation time."""
4401:         invalid_formats = ["P01", "PA1", "PA001", "pa01", "XA01", "PA-01"]
4402:
4403:         for invalid_pa in invalid_formats:
4404:             with pytest.raises(ValueError) as exc_info:
4405:                 create_chunk(0, invalid_pa, "DIM01")
4406:
4407:                 error_msg = str(exc_info.value)
4408:                 assert "Invalid chunk_id" in error_msg
4409:
4410:     def test_rejects_malformed_dim_format(self) -> None:
4411:         """ChunkData should reject malformed DIM identifiers at creation time."""
4412:         invalid_formats = ["D01", "DIM1", "DIM001", "dim01", "XIM01", "DIM-01"]
4413:
4414:         for invalid_dim in invalid_formats:
4415:             with pytest.raises(ValueError) as exc_info:
4416:                 create_chunk(0, "PA01", invalid_dim)
4417:
4418:                 error_msg = str(exc_info.value)
4419:                 assert "Invalid chunk_id" in error_msg
4420:
4421:
4422: class TestNullMetadataValidation:
4423:     """Test validation of null policy_area_id and dimension_id."""
```

```
4424:  
4425:     def test_rejects_null_policy_area_id(self) -> None:  
4426:         """ChunkData should reject chunks with null policy_area_id at creation time."""  
4427:         with pytest.raises(ValueError) as exc_info:  
4428:             ChunkData(  
4429:                 id=0,  
4430:                 text="test",  
4431:                 chunk_type="diagnostic",  
4432:                 sentences=[],  
4433:                 tables=[],  
4434:                 start_pos=0,  
4435:                 end_pos=4,  
4436:                 confidence=0.95,  
4437:                 policy_area_id=None,  
4438:                 dimension_id="DIM01",  
4439:             )  
4440:  
4441:             error_msg = str(exc_info.value)  
4442:             assert "chunk_id is required" in error_msg  
4443:  
4444:     def test_rejects_null_dimension_id(self) -> None:  
4445:         """ChunkData should reject chunks with null dimension_id at creation time."""  
4446:         with pytest.raises(ValueError) as exc_info:  
4447:             ChunkData(  
4448:                 id=0,  
4449:                 text="test",  
4450:                 chunk_type="diagnostic",  
4451:                 sentences=[],  
4452:                 tables=[],  
4453:                 start_pos=0,  
4454:                 end_pos=4,  
4455:                 confidence=0.95,  
4456:                 policy_area_id="PA01",  
4457:                 dimension_id=None,  
4458:             )  
4459:  
4460:             error_msg = str(exc_info.value)  
4461:             assert "chunk_id is required" in error_msg  
4462:  
4463:     def test_rejects_multiple_null_metadata(self) -> None:  
4464:         """ChunkData should detect missing chunk_id when both PA and DIM are null."""  
4465:         with pytest.raises(ValueError) as exc_info:  
4466:             ChunkData(  
4467:                 id=0,  
4468:                 text="test",  
4469:                 chunk_type="diagnostic",  
4470:                 sentences=[],  
4471:                 tables=[],  
4472:                 start_pos=0,  
4473:                 end_pos=4,  
4474:                 confidence=0.95,  
4475:                 policy_area_id=None,  
4476:                 dimension_id=None,  
4477:             )  
4478:  
4479:             error_msg = str(exc_info.value)
```

```
4480:         assert "chunk_id is required" in error_msg
4481:
4482:
4483: class TestValueErrorMessages:
4484:     """Test clarity and completeness of ValueError messages."""
4485:
4486:     def test_missing_chunk_error_includes_expected_format(self) -> None:
4487:         """Missing chunk error should explain expected format."""
4488:         doc = create_complete_document()
4489:         doc.chunks = doc.chunks[:59]
4490:
4491:         with pytest.raises(ValueError) as exc_info:
4492:             ChunkMatrix(doc)
4493:
4494:         error_msg = str(exc_info.value)
4495:         assert "PA" in error_msg and "DIM" in error_msg
4496:
4497:     def test_invalid_format_error_includes_expected_pattern(self) -> None:
4498:         """Invalid format error should show expected PA{01-10}-DIM{01-06} pattern."""
4499:         with pytest.raises(ValueError) as exc_info:
4500:             create_chunk(0, "PA99", "DIM01")
4501:
4502:         error_msg = str(exc_info.value)
4503:         assert (
4504:             "expected format PA{01-10}-DIM{01-06}" in error_msg.lower()
4505:             or "invalid chunk_id" in error_msg.lower()
4506:         )
4507:
4508:     def test_duplicate_error_includes_specific_key(self) -> None:
4509:         """Duplicate error should include the specific duplicate key."""
4510:         chunks = [create_chunk(0, "PA05", "DIM03"), create_chunk(1, "PA05", "DIM03")]
4511:
4512:         chunk_id = 2
4513:         for pa_num in range(1, 11):
4514:             for dim_num in range(1, 7):
4515:                 if pa_num == 5 and dim_num == 3:
4516:                     continue
4517:                 pa_id = f"PA{pa_num:02d}"
4518:                 dim_id = f"DIM{dim_num:02d}"
4519:                 chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4520:                 chunk_id += 1
4521:
4522:         doc = PreprocessedDocument(
4523:             document_id="test-doc",
4524:             raw_text="test",
4525:             sentences=[],
4526:             tables=[],
4527:             metadata={},
4528:             chunks=chunks,
4529:             ingested_at=datetime.now(),
4530:         )
4531:
4532:         with pytest.raises(ValueError) as exc_info:
4533:             ChunkMatrix(doc)
4534:
4535:         error_msg = str(exc_info.value)
```

```
4536:         assert "PA05" in error_msg and "DIM03" in error_msg
4537:
4538:     def test_null_metadata_error_message(self) -> None:
4539:         """Null metadata error should have clear message."""
4540:         with pytest.raises(ValueError) as exc_info:
4541:             ChunkData(
4542:                 id=42,
4543:                 text="test",
4544:                 chunk_type="diagnostic",
4545:                 sentences=[],
4546:                 tables=[],
4547:                 start_pos=0,
4548:                 end_pos=4,
4549:                 confidence=0.95,
4550:                 policy_area_id=None,
4551:                 dimension_id="DIM01",
4552:             )
4553:
4554:         error_msg = str(exc_info.value)
4555:         assert "chunk_id is required" in error_msg
4556:
4557:
4558: class TestChunkMatrixAccess:
4559:     """Test ChunkMatrix access patterns and guarantees."""
4560:
4561:     def test_get_chunk_returns_correct_chunk(self) -> None:
4562:         """get_chunk should return the chunk with matching PA and DIM."""
4563:         doc = create_complete_document()
4564:         matrix = ChunkMatrix(doc)
4565:
4566:         chunk = matrix.get_chunk("PA07", "DIM04")
4567:         assert chunk.policy_area_id == "PA07"
4568:         assert chunk.dimension_id == "DIM04"
4569:
4570:     def test_get_chunk_all_60_combinations(self) -> None:
4571:         """get_chunk should work for all 60 valid PA-DIM combinations."""
4572:         doc = create_complete_document()
4573:         matrix = ChunkMatrix(doc)
4574:
4575:         for pa_num in range(1, 11):
4576:             for dim_num in range(1, 7):
4577:                 pa_id = f"PA{pa_num:02d}"
4578:                 dim_id = f"DIM{dim_num:02d}"
4579:                 chunk = matrix.get_chunk(pa_id, dim_id)
4580:                 assert chunk.policy_area_id == pa_id
4581:                 assert chunk.dimension_id == dim_id
4582:
4583:     def test_get_chunk_raises_keyerror_for_invalid_combination(self) -> None:
4584:         """get_chunk should raise KeyError for invalid combinations."""
4585:         doc = create_complete_document()
4586:         matrix = ChunkMatrix(doc)
4587:
4588:         with pytest.raises(KeyError) as exc_info:
4589:             matrix.get_chunk("PA11", "DIM01")
4590:
4591:         error_msg = str(exc_info.value)
```

```
4592:         assert "Chunk not found" in error_msg
4593:         assert "PA11-DIM01" in error_msg
4594:
4595:     def test_get_chunk_preserves_chunk_metadata(self) -> None:
4596:         """get_chunk should preserve all chunk metadata."""
4597:         doc = create_complete_document()
4598:         doc.chunks[0] = ChunkData(
4599:             id=999,
4600:             text="specific content",
4601:             chunk_type="activity",
4602:             sentences=[1, 2, 3],
4603:             tables=[0],
4604:             start_pos=100,
4605:             end_pos=200,
4606:             confidence=0.87,
4607:             policy_area_id="PA01",
4608:             dimension_id="DIM01",
4609:         )
4610:
4611:         matrix = ChunkMatrix(doc)
4612:         chunk = matrix.get_chunk("PA01", "DIM01")
4613:
4614:         assert chunk.id == 999
4615:         assert chunk.text == "specific content"
4616:         assert chunk.chunk_type == "activity"
4617:         assert chunk.sentences == [1, 2, 3]
4618:         assert chunk.tables == [0]
4619:         assert chunk.start_pos == 100
4620:         assert chunk.end_pos == 200
4621:         assert chunk.confidence == 0.87
4622:
4623:
4624:     @st.composite
4625:     def valid_chunk_id_strategy(draw: Any) -> int: # type: ignore[misc]
4626:         """Strategy for generating valid chunk IDs."""
4627:         return draw(st.integers(min_value=0, max_value=10000)) # type: ignore[no-any-return]
4628:
4629:
4630:     @st.composite
4631:     def valid_pa_id_strategy(draw: Any) -> str: # type: ignore[misc]
4632:         """Strategy for generating valid PA identifiers (PA01-PA10)."""
4633:         pa_num: int = draw(st.integers(min_value=1, max_value=10))
4634:         return f"PA{pa_num:02d}"
4635:
4636:
4637:     @st.composite
4638:     def valid_dim_id_strategy(draw: Any) -> str: # type: ignore[misc]
4639:         """Strategy for generating valid DIM identifiers (DIM01-DIM06)."""
4640:         dim_num: int = draw(st.integers(min_value=1, max_value=6))
4641:         return f"DIM{dim_num:02d}"
4642:
4643:
4644:     @st.composite
4645:     def invalid_pa_id_strategy(draw: Any) -> str: # type: ignore[misc]
4646:         """Strategy for generating invalid PA identifiers."""
4647:         return draw( # type: ignore[no-any-return]
```

```
4648:         st.one_of(
4649:             st.just("PA00"),
4650:             st.just("PA11"),
4651:             st.just("PA99"),
4652:             st.just("P01"),
4653:             st.just("PA1"),
4654:             st.just("pa01"),
4655:         )
4656:     )
4657:
4658:
4659: @st.composite
4660: def invalid_dim_id_strategy(draw: Any) -> str: # type: ignore[misc]
4661:     """Strategy for generating invalid DIM identifiers."""
4662:     return draw( # type: ignore[no-any-return]
4663:         st.one_of(
4664:             st.just("DIM00"),
4665:             st.just("DIM07"),
4666:             st.just("DIM99"),
4667:             st.just("D01"),
4668:             st.just("DIM1"),
4669:             st.just("dim01"),
4670:         )
4671:     )
4672:
4673:
4674: class TestChunkMatrixPropertyBased:
4675:     """Property-based tests using Hypothesis for ChunkMatrix validation."""
4676:
4677:     @given(chunk_order=st.sampled_from(["sequential", "reversed", "shuffled"]))
4678:     @settings(max_examples=20, deadline=2000)
4679:     def test_property_ordering_invariance(self, chunk_order: str) -> None: # type: ignore[misc]
4680:         """Property: ChunkMatrix construction is invariant to input chunk ordering."""
4681:         doc = create_complete_document(chunk_order)
4682:         matrix = ChunkMatrix(doc)
4683:
4684:         for pa_num in range(1, 11):
4685:             for dim_num in range(1, 7):
4686:                 pa_id = f"PA{pa_num:02d}"
4687:                 dim_id = f"DIM{dim_num:02d}"
4688:                 chunk = matrix.get_chunk(pa_id, dim_id)
4689:                 assert chunk.policy_area_id == pa_id
4690:                 assert chunk.dimension_id == dim_id
4691:
4692:         @given(
4693:             pa_id=valid_pa_id_strategy(),
4694:             dim_id=valid_dim_id_strategy(),
4695:         )
4696:         @settings(max_examples=50, deadline=1000)
4697:         def test_property_valid_ids_accepted(self, pa_id: str, dim_id: str) -> None: # type: ignore[misc]
4698:             """Property: All valid PA-DIM combinations are accepted."""
4699:             doc = create_complete_document()
4700:             matrix = ChunkMatrix(doc)
4701:
4702:             chunk = matrix.get_chunk(pa_id, dim_id)
4703:             assert chunk.policy_area_id == pa_id
```

```
4704:         assert chunk.dimension_id == dim_id
4705:
4706:     @given(invalid_pa=invalid_pa_id_strategy())
4707:     @settings(
4708:         max_examples=20,
4709:         deadline=1000,
4710:         suppress_health_check=[HealthCheck.filter_too_much],
4711:     )
4712:     def test_property_invalid_pa_rejected(self, invalid_pa: str) -> None: # type: ignore[misc]
4713:         """Property: Invalid PA identifiers are rejected at ChunkData creation."""
4714:         with pytest.raises(ValueError) as exc_info:
4715:             create_chunk(0, invalid_pa, "DIM01")
4716:
4717:         assert "Invalid chunk_id" in str(exc_info.value)
4718:
4719:     @given(invalid_dim=invalid_dim_id_strategy())
4720:     @settings(
4721:         max_examples=20,
4722:         deadline=1000,
4723:         suppress_health_check=[HealthCheck.filter_too_much],
4724:     )
4725:     def test_property_invalid_dim_rejected(self, invalid_dim: str) -> None: # type: ignore[misc]
4726:         """Property: Invalid DIM identifiers are rejected at ChunkData creation."""
4727:         with pytest.raises(ValueError) as exc_info:
4728:             create_chunk(0, "PA01", invalid_dim)
4729:
4730:         assert "Invalid chunk_id" in str(exc_info.value)
4731:
4732:     @given(
4733:         pa_num=st.integers(min_value=1, max_value=10),
4734:         dim_num=st.integers(min_value=1, max_value=6),
4735:     )
4736:     @settings(max_examples=60, deadline=1000)
4737:     def test_property_all_combinations_accessible( # type: ignore[misc]
4738:         self, pa_num: int, dim_num: int
4739:     ) -> None:
4740:         """Property: All 60 PA-DIM combinations are accessible via get_chunk."""
4741:         doc = create_complete_document()
4742:         matrix = ChunkMatrix(doc)
4743:
4744:         pa_id = f"PA{pa_num:02d}"
4745:         dim_id = f"DIM{dim_num:02d}"
4746:
4747:         chunk = matrix.get_chunk(pa_id, dim_id)
4748:         assert chunk is not None
4749:         assert chunk.policy_area_id == pa_id
4750:         assert chunk.dimension_id == dim_id
4751:
4752:     @given(
4753:         missing_pa=st.integers(min_value=1, max_value=10),
4754:         missing_dim=st.integers(min_value=1, max_value=6),
4755:     )
4756:     @settings(max_examples=30, deadline=1000)
4757:     def test_property_missing_chunks_detected( # type: ignore[misc]
4758:         self, missing_pa: int, missing_dim: int
4759:     ) -> None:
```

```
4760:     """Property: Missing any PA-DIM combination is detected."""
4761:     chunks = []
4762:     chunk_id = 0
4763:     missing_pa_id = f"PA{missing_pa:02d}"
4764:     missing_dim_id = f"DIM{missing_dim:02d}"
4765:
4766:     for pa_num in range(1, 11):
4767:         for dim_num in range(1, 7):
4768:             pa_id = f"PA{pa_num:02d}"
4769:             dim_id = f"DIM{dim_num:02d}"
4770:             if pa_id == missing_pa_id and dim_id == missing_dim_id:
4771:                 continue
4772:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4773:             chunk_id += 1
4774:
4775:     doc = PreprocessedDocument(
4776:         document_id="test-doc",
4777:         raw_text="test",
4778:         sentences=[],
4779:         tables=[],
4780:         metadata={},
4781:         chunks=chunks,
4782:         ingested_at=datetime.now(),
4783:     )
4784:
4785:     with pytest.raises(ValueError) as exc_info:
4786:         ChunkMatrix(doc)
4787:
4788:     error_msg = str(exc_info.value)
4789:     assert "Missing chunk combinations" in error_msg
4790:     assert f"{missing_pa_id}-{missing_dim_id}" in error_msg
4791:
4792:     @given(
4793:         dup_pa=st.integers(min_value=1, max_value=10),
4794:         dup_dim=st.integers(min_value=1, max_value=6),
4795:     )
4796:     @settings(max_examples=30, deadline=1000)
4797:     def test_property_duplicates_detected(self, dup_pa: int, dup_dim: int) -> None: # type: ignore[misc]
4798:         """Property: Duplicate PA-DIM combinations are always detected."""
4799:         dup_pa_id = f"PA{dup_pa:02d}"
4800:         dup_dim_id = f"DIM{dup_dim:02d}"
4801:
4802:         chunks = [
4803:             create_chunk(0, dup_pa_id, dup_dim_id),
4804:             create_chunk(1, dup_pa_id, dup_dim_id),
4805:         ]
4806:
4807:         chunk_id = 2
4808:         for pa_num in range(1, 11):
4809:             for dim_num in range(1, 7):
4810:                 pa_id = f"PA{pa_num:02d}"
4811:                 dim_id = f"DIM{dim_num:02d}"
4812:                 if pa_id == dup_pa_id and dim_id == dup_dim_id:
4813:                     continue
4814:                 chunks.append(create_chunk(chunk_id, pa_id, dim_id))
4815:                 chunk_id += 1
```

```
4816:  
4817:     doc = PreprocessedDocument(  
4818:         document_id="test-doc",  
4819:         raw_text="test",  
4820:         sentences=[],  
4821:         tables=[],  
4822:         metadata={},  
4823:         chunks=chunks,  
4824:         ingested_at=datetime.now(),  
4825:     )  
4826:  
4827:     with pytest.raises(ValueError) as exc_info:  
4828:         ChunkMatrix(doc)  
4829:  
4830:     error_msg = str(exc_info.value)  
4831:     assert "Duplicate" in error_msg  
4832:     assert dup_pa_id in error_msg and dup_dim_id in error_msg  
4833:  
4834:     @given(text=st.text(min_size=1, max_size=1000))  
4835:     @settings(max_examples=30, deadline=1000)  
4836:     def test_property_chunk_content_preserved(self, text: str) -> None: # type: ignore[misc]  
4837:         """Property: Chunk text content is preserved through matrix construction."""  
4838:         doc = create_complete_document()  
4839:         doc.chunks[0] = create_chunk(0, "PA01", "DIM01", text=text)  
4840:  
4841:         matrix = ChunkMatrix(doc)  
4842:         chunk = matrix.get_chunk("PA01", "DIM01")  
4843:  
4844:         assert chunk.text == text  
4845:  
4846:  
4847: class TestChunkMatrixAuditLog:  
4848:     """Test audit log structure and validation metadata."""  
4849:  
4850:     def test_matrix_internal_structure_is_dict(self) -> None:  
4851:         """ChunkMatrix internal structure should be accessible as dict."""  
4852:         doc = create_complete_document()  
4853:         matrix = ChunkMatrix(doc)  
4854:  
4855:         assert hasattr(matrix, "chunks")  
4856:         assert isinstance(matrix.chunks, dict)  
4857:  
4858:     def test_matrix_stores_all_60_chunks(self) -> None:  
4859:         """ChunkMatrix should store exactly 60 chunks internally."""  
4860:         doc = create_complete_document()  
4861:         matrix = ChunkMatrix(doc)  
4862:  
4863:         assert len(matrix.chunks) == ChunkMatrix.EXPECTED_CHUNK_COUNT  
4864:  
4865:     def test_matrix_keys_are_tuples(self) -> None:  
4866:         """ChunkMatrix keys should be (policy_area_id, dimension_id) tuples."""  
4867:         doc = create_complete_document()  
4868:         matrix = ChunkMatrix(doc)  
4869:  
4870:         for key in matrix.chunks.keys():  
4871:             assert isinstance(key, tuple)
```

```
4872:         assert len(key) == 2
4873:         assert isinstance(key[0], str)
4874:         assert isinstance(key[1], str)
4875:
4876:     def test_matrix_values_are_chunk_data(self) -> None:
4877:         """ChunkMatrix values should be ChunkData instances."""
4878:         doc = create_complete_document()
4879:         matrix = ChunkMatrix(doc)
4880:
4881:         for value in matrix.chunks.values():
4882:             assert isinstance(value, ChunkData)
4883:
4884:     def test_matrix_contains_all_expected_keys(self) -> None:
4885:         """ChunkMatrix should contain all 60 expected PA-DIM combinations as keys."""
4886:         doc = create_complete_document()
4887:         matrix = ChunkMatrix(doc)
4888:
4889:         expected_keys = {(pa, dim) for pa in ChunkMatrix.POLICY AREAS for dim in ChunkMatrix.DIMENSIONS}
4890:
4891:         assert set(matrix.chunks.keys()) == expected_keys
4892:
4893:     def test_matrix_key_to_chunk_mapping_is_correct(self) -> None:
4894:         """ChunkMatrix keys should correctly map to their corresponding chunks."""
4895:         doc = create_complete_document()
4896:         matrix = ChunkMatrix(doc)
4897:
4898:         for (pa_id, dim_id), chunk in matrix.chunks.items():
4899:             assert chunk.policy_area_id == pa_id
4900:             assert chunk.dimension_id == dim_id
4901:
4902:
4903: class TestChunkMatrixConstants:
4904:     """Test ChunkMatrix class constants and configuration."""
4905:
4906:     def test_policy_areas_constant(self) -> None:
4907:         """ChunkMatrix.POLICY AREAS should contain PA01-PA10."""
4908:         expected = ChunkMatrix.POLICY AREAS
4909:         assert expected == ChunkMatrix.POLICY AREAS
4910:
4911:     def test_dimensions_constant(self) -> None:
4912:         """ChunkMatrix.DIMENSIONS should contain DIM01-DIM06."""
4913:         expected = ChunkMatrix.DIMENSIONS
4914:         assert expected == ChunkMatrix.DIMENSIONS
4915:
4916:     def test_expected_chunk_count_constant(self) -> None:
4917:         """ChunkMatrix.EXPECTED CHUNK COUNT should match PA\227DIM cardinality."""
4918:         assert ChunkMatrix.EXPECTED CHUNK COUNT == len(ChunkMatrix.POLICY AREAS) * len(
4919:             ChunkMatrix.DIMENSIONS
4920:         )
4921:
4922:     def test_chunk_id_pattern_validates_correctly(self) -> None:
4923:         """ChunkMatrix.CHUNK_ID_PATTERN should validate correct formats."""
4924:         pattern = ChunkMatrix.CHUNK_ID_PATTERN
4925:
4926:         valid_ids = [
4927:             f"{pa}-{dim[-2:]}" if dim.startswith("DIM") else f"{pa}-{dim}"
```

```
4928:         for pa in ChunkMatrix.POLICY AREAS
4929:             for dim in ChunkMatrix.DIMENSIONS
4930:                 ]
4931:
4932:             for chunk_id in valid_ids:
4933:                 assert pattern.match(chunk_id), f"Valid ID rejected: {chunk_id}"
4934:
4935:     def test_chunk_id_pattern_rejects_invalid(self) -> None:
4936:         """ChunkMatrix.CHUNK_ID_PATTERN should reject invalid formats."""
4937:         pattern = ChunkMatrix.CHUNK_ID_PATTERN
4938:
4939:         invalid_ids = [
4940:             "PA0-DIM01",
4941:             "PA001-DIM01",
4942:             "P01-DIM01",
4943:             "PA01-DIM1",
4944:             "PA01-DIM001",
4945:             "pa01-dim01",
4946:             "PA01_DIM01",
4947:             "PA01-DIM0A",
4948:             "XX01-DIM01",
4949:         ]
4950:
4951:         for chunk_id in invalid_ids:
4952:             assert not pattern.match(chunk_id), f"Invalid ID accepted: {chunk_id}"
4953:
4954:
4955: if __name__ == "__main__":
4956:     pytest.main([__file__, "-v", "--tb=short"])
4957:
4958:
4959:
4960: =====
4961: FILE: tests/core/test_chunk_router.py
4962: =====
4963:
4964: """
4965: Tests for chunk_router.py - Routing logic formalization and contracts.
4966:
4967: Validates the ExecutionMap contract, serialization/deserialization,
4968: and deterministic behavior of routing logic.
4969: """
4970:
4971: import hashlib
4972: import json
4973:
4974: import pytest
4975: from pydantic import ValidationError
4976:
4977: pytestmark = pytest.mark.obsolete
4978:
4979: from farfan_pipeline.core.orchestrator.chunk_router import (
4980:     ChunkRouter,
4981:     ExecutionMap,
4982:     deserialize_execution_map,
4983:     serialize_execution_map,
```

```
4984: )
4985: from farfan_pipeline.core.types import ChunkData
4986:
4987:
4988: @pytest.fixture
4989: def sample_chunks():
4990:     """Create sample chunks with policy area and dimension assignments."""
4991:     return [
4992:         ChunkData(
4993:             id=1,
4994:             text="Diagnostic text about women's rights",
4995:             chunk_type="diagnostic",
4996:             sentences=[0, 1, 2],
4997:             tables=[],
4998:             start_pos=0,
4999:             end_pos=100,
5000:             confidence=0.95,
5001:             policy_area_id="PA01",
5002:             dimension_id="DIM01",
5003:         ),
5004:         ChunkData(
5005:             id=2,
5006:             text="Activity description for violence prevention",
5007:             chunk_type="activity",
5008:             sentences=[3, 4, 5],
5009:             tables=[],
5010:             start_pos=101,
5011:             end_pos=200,
5012:             confidence=0.92,
5013:             policy_area_id="PA02",
5014:             dimension_id="DIM02",
5015:         ),
5016:         ChunkData(
5017:             id=3,
5018:             text="Indicator for environmental metrics",
5019:             chunk_type="indicator",
5020:             sentences=[6, 7],
5021:             tables=[],
5022:             start_pos=201,
5023:             end_pos=300,
5024:             confidence=0.88,
5025:             policy_area_id="PA03",
5026:             dimension_id="DIM03",
5027:         ),
5028:     ]
5029:
5030:
5031: @pytest.fixture
5032: def router():
5033:     """Create a ChunkRouter instance."""
5034:     return ChunkRouter()
5035:
5036:
5037: class TestExecutionMapContract:
5038:     """Test suite for ExecutionMap contract validation."""
5039:
```

```
5040:     def test_valid_execution_map(self):
5041:         """Valid ExecutionMap should be created successfully."""
5042:         routing_rules = {
5043:             "PA01:DIM01": "D1Q1",
5044:             "PA02:DIM02": "D2Q1",
5045:         }
5046:         canonical_repr = json.dumps(
5047:             routing_rules, sort_keys=True, separators=(", ", ":"))  

5048:         )
5049:         map_hash = hashlib.sha256(canonical_repr.encode("utf-8")).hexdigest()
5050:
5051:         execution_map = ExecutionMap(
5052:             version="1.0.0",
5053:             map_hash=map_hash,
5054:             routing_rules=routing_rules,
5055:         )
5056:
5057:         assert execution_map.version == "1.0.0"
5058:         assert execution_map.map_hash == map_hash
5059:         assert execution_map.routing_rules == routing_rules
5060:
5061:     def test_invalid_version_format(self):
5062:         """Version must match semantic versioning format."""
5063:         with pytest.raises(ValidationError, match="version"):
5064:             ExecutionMap(
5065:                 version="v1", # Invalid format
5066:                 map_hash="a" * 64,
5067:                 routing_rules={"PA01:DIM01": "D1Q1"},  

5068:             )
5069:
5070:     def test_invalid_hash_length(self):
5071:         """map_hash must be exactly 64 characters."""
5072:         with pytest.raises(ValidationError, match="map_hash"):
5073:             ExecutionMap(
5074:                 version="1.0.0",
5075:                 map_hash="abc123", # Too short
5076:                 routing_rules={"PA01:DIM01": "D1Q1"},  

5077:             )
5078:
5079:     def test_invalid_hash_format(self):
5080:         """map_hash must be valid hexadecimal."""
5081:         with pytest.raises(ValidationError, match="hexadecimal"):
5082:             ExecutionMap(
5083:                 version="1.0.0",
5084:                 map_hash="z" * 64, # Invalid hex characters
5085:                 routing_rules={"PA01:DIM01": "D1Q1"},  

5086:             )
5087:
5088:     def test_empty_routing_rules(self):
5089:         """routing_rules cannot be empty."""
5090:         with pytest.raises(ValidationError, match="routing_rules cannot be empty"):
5091:             ExecutionMap(
5092:                 version="1.0.0",
5093:                 map_hash="a" * 64,
5094:                 routing_rules={},  

5095:             )
```

```
5096:
5097:     def test_invalid_routing_key_format(self):
5098:         """Routing keys must be in 'policy_area_id:dimension_id' format."""
5099:         with pytest.raises(ValidationError, match="policy_area_id:dimension_id"):
5100:             ExecutionMap(
5101:                 version="1.0.0",
5102:                 map_hash="a" * 64,
5103:                 routing_rules={"INVALID_KEY": "D1Q1"},
5104:             )
5105:
5106:     def test_invalid_routing_key_missing_parts(self):
5107:         """Routing keys must have both policy_area_id and dimension_id."""
5108:         with pytest.raises(ValidationError, match="non-empty"):
5109:             ExecutionMap(
5110:                 version="1.0.0",
5111:                 map_hash="a" * 64,
5112:                 routing_rules={"PA01": "D1Q1"}, # Missing dimension_id
5113:             )
5114:
5115:     def test_empty_executor_class(self):
5116:         """Executor class must be non-empty string."""
5117:         with pytest.raises(ValidationError, match="non-empty string"):
5118:             ExecutionMap(
5119:                 version="1.0.0",
5120:                 map_hash="a" * 64,
5121:                 routing_rules={"PA01:DIM01": ""},
5122:             )
5123:
5124:     def test_get_executor(self):
5125:         """get_executor should retrieve correct executor for given area/dimension."""
5126:         execution_map = ExecutionMap(
5127:             version="1.0.0",
5128:             map_hash="a" * 64,
5129:             routing_rules={
5130:                 "PA01:DIM01": "D1Q1",
5131:                 "PA02:DIM02": "D2Q1",
5132:             },
5133:         )
5134:
5135:         assert execution_map.get_executor("PA01", "DIM01") == "D1Q1"
5136:         assert execution_map.get_executor("PA02", "DIM02") == "D2Q1"
5137:         assert execution_map.get_executor("PA99", "DIM99") is None
5138:
5139:
5140: class TestGenerateExecutionMap:
5141:     """Test suite for generate_execution_map functionality."""
5142:
5143:     def test_generate_execution_map_basic(self, router, sample_chunks):
5144:         """Generate execution map from valid chunks."""
5145:         execution_map = router.generate_execution_map(sample_chunks)
5146:
5147:         assert execution_map.version == "1.0.0"
5148:         expected_rules_count = 3
5149:         assert len(execution_map.routing_rules) == expected_rules_count
5150:         assert "PA01:DIM01" in execution_map.routing_rules
5151:         assert "PA02:DIM02" in execution_map.routing_rules
```

```
5152:         assert "PA03:DIM03" in execution_map.routing_rules
5153:
5154:         assert execution_map.routing_rules["PA01:DIM01"] == "D1Q1"
5155:         assert execution_map.routing_rules["PA02:DIM02"] == "D2Q1"
5156:         assert execution_map.routing_rules["PA03:DIM03"] == "D3Q1"
5157:
5158:     def test_generate_execution_map_deterministic(self, router, sample_chunks):
5159:         """Execution map generation must be deterministic."""
5160:         map1 = router.generate_execution_map(sample_chunks)
5161:         map2 = router.generate_execution_map(sample_chunks)
5162:
5163:         assert map1.version == map2.version
5164:         assert map1.map_hash == map2.map_hash
5165:         assert map1.routing_rules == map2.routing_rules
5166:
5167:     def test_generate_execution_map_order_independent(self, router, sample_chunks):
5168:         """Execution map should be same regardless of chunk order."""
5169:         import random
5170:
5171:         chunks_copy = sample_chunks.copy()
5172:         random.shuffle(chunks_copy)
5173:
5174:         map1 = router.generate_execution_map(sample_chunks)
5175:         map2 = router.generate_execution_map(chunks_copy)
5176:
5177:         assert map1.map_hash == map2.map_hash
5178:         assert map1.routing_rules == map2.routing_rules
5179:
5180:     def test_generate_execution_map_missing_policy_area(self, router):
5181:         """Should raise error if chunk missing policy_area_id."""
5182:         chunks = [
5183:             ChunkData(
5184:                 id=1,
5185:                 text="Test",
5186:                 chunk_type="diagnostic",
5187:                 sentences=[],
5188:                 tables=[],
5189:                 start_pos=0,
5190:                 end_pos=10,
5191:                 confidence=0.9,
5192:                 policy_area_id=None, # Missing
5193:                 dimension_id="DIM01",
5194:             )
5195:         ]
5196:
5197:         with pytest.raises(ValueError, match="policy_area_id"):
5198:             router.generate_execution_map(chunks)
5199:
5200:     def test_generate_execution_map_missing_dimension(self, router):
5201:         """Should raise error if chunk missing dimension_id."""
5202:         chunks = [
5203:             ChunkData(
5204:                 id=1,
5205:                 text="Test",
5206:                 chunk_type="diagnostic",
5207:                 sentences=[],
```

```
5208:         tables=[],
5209:         start_pos=0,
5210:         end_pos=10,
5211:         confidence=0.9,
5212:         policy_area_id="PA01",
5213:         dimension_id=None, # Missing
5214:     )
5215: ]
5216:
5217:     with pytest.raises(ValueError, match="dimension_id"):
5218:         router.generate_execution_map(chunks)
5219:
5220: def test_generate_execution_map_unknown_chunk_type(self, router):
5221:     """Unknown chunk types should be marked as UNROUTED."""
5222:     chunks = [
5223:         ChunkData(
5224:             id=1,
5225:             text="Test",
5226:             chunk_type="unknown_type",
5227:             sentences=[],
5228:             tables=[],
5229:             start_pos=0,
5230:             end_pos=10,
5231:             confidence=0.9,
5232:             policy_area_id="PA01",
5233:             dimension_id="DIM01",
5234:         )
5235:     ]
5236:
5237:     execution_map = router.generate_execution_map(chunks)
5238:     assert execution_map.routing_rules["PA01:DIM01"] == "UNROUTED_UNKNOWN_TYPE"
5239:
5240: def test_hash_verification(self, router, sample_chunks):
5241:     """Generated hash should match computed hash of routing rules."""
5242:     execution_map = router.generate_execution_map(sample_chunks)
5243:
5244:     canonical_repr = json.dumps(
5245:         dict(sorted(execution_map.routing_rules.items())),
5246:         sort_keys=True,
5247:         separators=(", ", ":"), )
5248:     )
5249:     expected_hash = hashlib.sha256(canonical_repr.encode("utf-8")).hexdigest()
5250:
5251:     assert execution_map.map_hash == expected_hash
5252:
5253: def test_custom_version(self, router, sample_chunks):
5254:     """Should accept custom version string."""
5255:     execution_map = router.generate_execution_map(sample_chunks, version="2.1.5")
5256:     assert execution_map.version == "2.1.5"
5257:
5258:
5259: class TestSerialization:
5260:     """Test suite for serialization and deserialization."""
5261:
5262:     def test_serialize_execution_map(self, router, sample_chunks):
5263:         """Serialize ExecutionMap to JSON string."""
```

```
5264:     execution_map = router.generate_execution_map(sample_chunks)
5265:     serialized = serialize_execution_map(execution_map)
5266:
5267:     assert isinstance(serialized, str)
5268:     data = json.loads(serialized)
5269:
5270:     assert data["version"] == execution_map.version
5271:     assert data["map_hash"] == execution_map.map_hash
5272:     assert data["routing_rules"] == execution_map.routing_rules
5273:
5274:     def test_deserialize_execution_map(self, router, sample_chunks):
5275:         """Deserialize JSON string back to ExecutionMap."""
5276:         execution_map = router.generate_execution_map(sample_chunks)
5277:         serialized = serialize_execution_map(execution_map)
5278:         deserialized = deserialize_execution_map(serialized)
5279:
5280:         assert deserialized.version == execution_map.version
5281:         assert deserialized.map_hash == execution_map.map_hash
5282:         assert deserialized.routing_rules == execution_map.routing_rules
5283:
5284:     def test_roundtrip_serialization(self, router, sample_chunks):
5285:         """Roundtrip serialization should preserve data exactly."""
5286:         original = router.generate_execution_map(sample_chunks)
5287:         serialized = serialize_execution_map(original)
5288:         deserialized = deserialize_execution_map(serialized)
5289:         reserialized = serialize_execution_map(deserialized)
5290:
5291:         assert serialized == reserialized
5292:         assert original.version == deserialized.version
5293:         assert original.map_hash == deserialized.map_hash
5294:         assert original.routing_rules == deserialized.routing_rules
5295:
5296:     def test_deserialize_invalid_json(self):
5297:         """Should raise error for invalid JSON."""
5298:         with pytest.raises(ValueError, match="Invalid JSON"):
5299:             deserialize_execution_map("not valid json")
5300:
5301:     def test_deserialize_missing_fields(self):
5302:         """Should raise validation error for missing required fields."""
5303:         incomplete_json = json.dumps({"version": "1.0.0"})
5304:
5305:         with pytest.raises(ValidationError):
5306:             deserialize_execution_map(incomplete_json)
5307:
5308:     def test_serialization_deterministic(self, router, sample_chunks):
5309:         """Serialization must be deterministic."""
5310:         execution_map = router.generate_execution_map(sample_chunks)
5311:
5312:         serialized1 = serialize_execution_map(execution_map)
5313:         serialized2 = serialize_execution_map(execution_map)
5314:
5315:         assert serialized1 == serialized2
5316:
5317:
5318: class TestChunkRouterCompatibility:
5319:     """Test suite for backward compatibility with existing ChunkRouter methods."""
```

```
5320:
5321:     def test_route_chunk_backward_compat(self, router, sample_chunks):
5322:         """route_chunk should continue to work as before."""
5323:         chunk = sample_chunks[0]
5324:         route = router.route_chunk(chunk)
5325:
5326:         assert route.chunk_id == chunk.id
5327:         assert route.chunk_type == chunk.chunk_type
5328:         assert route.executor_class == "D1Q1"
5329:         assert route.skip_reason is None
5330:
5331:     def test_should_use_full_graph(self, router):
5332:         """should_use_full_graph should identify graph methods correctly."""
5333:         assert router.should_use_full_graph("construir_grafo_causal", "TeoriaCambio")
5334:         assert router.should_use_full_graph(
5335:             "extract_causal_hierarchy", "CausalExtractor"
5336:         )
5337:         assert not router.should_use_full_graph("some_regular_method")
5338:
5339:     def test_get_relevant_executors(self, router):
5340:         """get_relevant_executors should return executor list for chunk type."""
5341:         assert router.get_relevant_executors("diagnostic") == ["D1Q1", "D1Q2", "D1Q5"]
5342:         assert router.get_relevant_executors("activity") == [
5343:             "D2Q1",
5344:             "D2Q2",
5345:             "D2Q3",
5346:             "D2Q4",
5347:             "D2Q5",
5348:         ]
5349:         assert router.get_relevant_executors("unknown") == []
5350:
5351:
5352: class TestDeterminism:
5353:     """Property-based tests for determinism guarantees."""
5354:
5355:     def test_same_input_same_output(self, router):
5356:         """Multiple calls with identical input must produce identical output."""
5357:         chunks = [
5358:             ChunkData(
5359:                 id=i,
5360:                 text=f"Chunk {i}",
5361:                 chunk_type="diagnostic",
5362:                 sentences=[i],
5363:                 tables=[],
5364:                 start_pos=i * 100,
5365:                 end_pos=(i + 1) * 100,
5366:                 confidence=0.9,
5367:                 policy_area_id=f"PA{i:02d}",
5368:                 dimension_id=f"DIM{(i % 6) + 1:02d}",
5369:             )
5370:             for i in range(1, 11)
5371:         ]
5372:
5373:         results = [router.generate_execution_map(chunks) for _ in range(10)]
5374:
5375:         first_hash = results[0].map_hash
```

```
5376:         first_rules = results[0].routing_rules
5377:
5378:     for result in results[1:]:
5379:         assert result.map_hash == first_hash
5380:         assert result.routing_rules == first_rules
5381:
5382:     def test_hash_collision_resistance(self, router):
5383:         """Different routing rules should produce different hashes."""
5384:         chunks1 = [
5385:             ChunkData(
5386:                 id=1,
5387:                 text="Test",
5388:                 chunk_type="diagnostic",
5389:                 sentences=[],
5390:                 tables=[],
5391:                 start_pos=0,
5392:                 end_pos=10,
5393:                 confidence=0.9,
5394:                 policy_area_id="PA01",
5395:                 dimension_id="DIM01",
5396:             )
5397:         ]
5398:
5399:         chunks2 = [
5400:             ChunkData(
5401:                 id=1,
5402:                 text="Test",
5403:                 chunk_type="diagnostic",
5404:                 sentences=[],
5405:                 tables=[],
5406:                 start_pos=0,
5407:                 end_pos=10,
5408:                 confidence=0.9,
5409:                 policy_area_id="PA02",
5410:                 dimension_id="DIM01",
5411:             )
5412:         ]
5413:
5414:         map1 = router.generate_execution_map(chunks1)
5415:         map2 = router.generate_execution_map(chunks2)
5416:
5417:         assert map1.map_hash != map2.map_hash
5418:         assert map1.routing_rules != map2.routing_rules
5419:
5420:
```