

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 29
3: =====
4: Generated: 2025-12-07T06:17:37.330864
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: tests/test_resource_manager.py
11: =====
12:
13: """Tests for adaptive resource management system."""
14:
15: import asyncio
16: from datetime import datetime
17: from unittest.mock import MagicMock, patch
18:
19: import pytest
20:
21: from farfan_pipeline.core.orchestrator.core import ResourceLimits
22: from farfan_pipeline.core.orchestrator.resource_manager import (
23:     AdaptiveResourceManager,
24:     CircuitBreaker,
25:     CircuitBreakerConfig,
26:     CircuitState,
27:     DegradationStrategy,
28:     ExecutorPriority,
29:     ResourceAllocationPolicy,
30:     ResourcePressureLevel,
31: )
32: from farfan_pipeline.core.orchestrator.resource_alerts import (
33:     AlertChannel,
34:     AlertSeverity,
35:     ResourceAlertManager,
36: )
37: from farfan_pipeline.core.orchestrator.resource_integration import (
38:     create_resource_manager,
39:     register_default_policies,
40: )
41:
42:
43: @pytest.fixture
44: def resource_limits():
45:     """Create ResourceLimits instance for testing."""
46:     return ResourceLimits(
47:         max_memory_mb=1024.0,
48:         max_cpu_percent=80.0,
49:         max_workers=8,
50:         min_workers=2,
51:         hard_max_workers=16,
52:     )
53:
54: @pytest.fixture
55: def resource_manager(resource_limits):
```

```
57:     """Create AdaptiveResourceManager instance for testing."""
58:     return AdaptiveResourceManager(
59:         resource_limits=resource_limits,
60:         enable_circuit_breakers=True,
61:         enable_degradation=True,
62:     )
63:
64:
65: @pytest.fixture
66: def alert_manager():
67:     """Create ResourceAlertManager instance for testing."""
68:     return ResourceAlertManager(
69:         channels=[AlertChannel.LOG],
70:     )
71:
72:
73: class TestCircuitBreaker:
74:     """Tests for CircuitBreaker functionality."""
75:
76:     def test_initial_state_closed(self):
77:         """Circuit breaker should start in closed state."""
78:         breaker = CircuitBreaker(executor_id="test-executor")
79:         assert breaker.state == CircuitState.CLOSED
80:         assert breaker.can_execute()
81:
82:     def test_opens_after_threshold_failures(self):
83:         """Circuit breaker should open after threshold failures."""
84:         config = CircuitBreakerConfig(failure_threshold=3)
85:         breaker = CircuitBreaker(executor_id="test-executor", config=config)
86:
87:         breaker.record_failure()
88:         assert breaker.state == CircuitState.CLOSED
89:
90:         breaker.record_failure()
91:         assert breaker.state == CircuitState.CLOSED
92:
93:         breaker.record_failure()
94:         assert breaker.state == CircuitState.OPEN
95:         assert not breaker.can_execute()
96:
97:     def test_opens_on_memory_threshold(self):
98:         """Circuit breaker should open on memory threshold."""
99:         config = CircuitBreakerConfig(
100:             failure_threshold=10, memory_threshold_mb=512.0
101:         )
102:         breaker = CircuitBreaker(executor_id="test-executor", config=config)
103:
104:         breaker.record_failure(memory_mb=1024.0)
105:         assert breaker.state == CircuitState.OPEN
106:
107:     def test_reset_on_success(self):
108:         """Failure count should reset on success."""
109:         breaker = CircuitBreaker(executor_id="test-executor")
110:
111:         breaker.record_failure()
112:         breaker.record_failure()
```

```
113:         assert breaker.failure_count == 2
114:
115:         breaker.record_success()
116:         assert breaker.failure_count == 0
117:
118:     def test_half_open_to_closed(self):
119:         """Circuit breaker should close after successes in half-open state."""
120:         config = CircuitBreakerConfig(
121:             failure_threshold=2,
122:             success_threshold=2,
123:             timeout_seconds=0.1,
124:         )
125:         breaker = CircuitBreaker(executor_id="test-executor", config=config)
126:
127:         breaker.record_failure()
128:         breaker.record_failure()
129:         assert breaker.state == CircuitState.OPEN
130:
131:         import time
132:         time.sleep(0.2)
133:
134:         assert breaker.can_execute()
135:         assert breaker.state == CircuitState.HALF_OPEN
136:
137:         breaker.record_success()
138:         breaker.record_success()
139:         assert breaker.state == CircuitState.CLOSED
140:
141:
142: class TestDegradationStrategy:
143:     """Tests for DegradationStrategy functionality."""
144:
145:     def test_should_apply_at_threshold(self):
146:         """Strategy should apply at or above threshold."""
147:         strategy = DegradationStrategy(
148:             name="test",
149:             pressure_threshold=ResourcePressureLevel.HIGH,
150:         )
151:
152:         assert not strategy.should_apply(ResourcePressureLevel.NORMAL)
153:         assert not strategy.should_apply(ResourcePressureLevel.ELEVATED)
154:         assert strategy.should_apply(ResourcePressureLevel.HIGH)
155:         assert strategy.should_apply(ResourcePressureLevel.CRITICAL)
156:         assert strategy.should_apply(ResourcePressureLevel.EMERGENCY)
157:
158:     def test_disabled_strategy_never_applies(self):
159:         """Disabled strategy should never apply."""
160:         strategy = DegradationStrategy(
161:             name="test",
162:             pressure_threshold=ResourcePressureLevel.NORMAL,
163:             enabled=False,
164:         )
165:
166:         assert not strategy.should_apply(ResourcePressureLevel.EMERGENCY)
167:
168:
```

```
169: class TestAdaptiveResourceManager:
170:     """Tests for AdaptiveResourceManager functionality."""
171:
172:     @pytest.mark.asyncio
173:     async def test_assess_pressure_normal(self, resource_manager):
174:         """Should assess normal pressure with low resource usage."""
175:         with patch.object(
176:             resource_manager.resource_limits,
177:             "get_resource_usage",
178:             return_value={
179:                 "cpu_percent": 30.0,
180:                 "memory_percent": 40.0,
181:                 "rss_mb": 200.0,
182:                 "worker_budget": 8.0,
183:             },
184:         ):
185:             pressure = await resource_manager.assess_resource_pressure()
186:             assert pressure == ResourcePressureLevel.NORMAL
187:
188:     @pytest.mark.asyncio
189:     async def test_assess_pressure_elevated(self, resource_manager):
190:         """Should assess elevated pressure with moderate resource usage."""
191:         with patch.object(
192:             resource_manager.resource_limits,
193:             "get_resource_usage",
194:             return_value={
195:                 "cpu_percent": 55.0,
196:                 "memory_percent": 66.0,
197:                 "rss_mb": 676.0,
198:                 "worker_budget": 8.0,
199:             },
200:         ):
201:             pressure = await resource_manager.assess_resource_pressure()
202:             assert pressure == ResourcePressureLevel.ELEVATED
203:
204:     @pytest.mark.asyncio
205:     async def test_assess_pressure_critical(self, resource_manager):
206:         """Should assess critical pressure with high resource usage."""
207:         with patch.object(
208:             resource_manager.resource_limits,
209:             "get_resource_usage",
210:             return_value={
211:                 "cpu_percent": 75.0,
212:                 "memory_percent": 88.0,
213:                 "rss_mb": 900.0,
214:                 "worker_budget": 8.0,
215:             },
216:         ):
217:             pressure = await resource_manager.assess_resource_pressure()
218:             assert pressure == ResourcePressureLevel.CRITICAL
219:
220:     def test_can_execute_with_closed_breaker(self, resource_manager):
221:         """Should allow execution with closed circuit breaker."""
222:         can_exec, reason = resource_manager.can_execute("test-executor")
223:         assert can_exec
224:         assert reason == "OK"
```

```
225:
226:     def test_can_execute_with_open_breaker(self, resource_manager):
227:         """Should block execution with open circuit breaker."""
228:         breaker = resource_manager.get_or_create_circuit_breaker("test-executor")
229:         breaker.state = CircuitState.OPEN
230:         breaker.last_state_change = datetime.utcnow()
231:
232:         can_exec, reason = resource_manager.can_execute("test-executor")
233:         assert not can_exec
234:         assert "open" in reason.lower()
235:
236:     def test_get_degradation_config_normal(self, resource_manager):
237:         """Should get normal config at normal pressure."""
238:         config = resource_manager.get_degradation_config("test-executor")
239:
240:         assert config["entity_limit_factor"] == 1.0
241:         assert not config["disable_expensive_computations"]
242:         assert not config["use_simplified_methods"]
243:         assert len(config["applied_strategies"]) == 0
244:
245:     def test_get_degradation_config_critical(self, resource_manager):
246:         """Should get degraded config at critical pressure."""
247:         resource_manager.current_pressure = ResourcePressureLevel.CRITICAL
248:         config = resource_manager.get_degradation_config("test-executor")
249:
250:         assert config["entity_limit_factor"] < 1.0
251:         assert config["disable_expensive_computations"]
252:         assert config["use_simplified_methods"]
253:         assert len(config["applied_strategies"]) > 0
254:
255:     @pytest.mark.asyncio
256:     async def test_allocate_resources_for_critical_executor(
257:         self, resource_manager
258):
259:         """Should allocate more resources for critical executors."""
260:         allocation = await resource_manager.allocate_resources("D3-Q3")
261:
262:         assert allocation["priority"] == ExecutorPriority.CRITICAL.value
263:         assert "max_memory_mb" in allocation
264:         assert "max_workers" in allocation
265:
266:     @pytest.mark.asyncio
267:     async def test_start_end_executor_execution(self, resource_manager):
268:         """Should track executor execution lifecycle."""
269:         allocation = await resource_manager.start_executor_execution("test-exec")
270:         assert "test-exec" in resource_manager._active_executors
271:
272:         await resource_manager.end_executor_execution(
273:             executor_id="test-exec",
274:             success=True,
275:             duration_ms=100.0,
276:             memory_mb=256.0,
277:         )
278:
279:         assert "test-exec" not in resource_manager._active_executors
280:         assert "test-exec" in resource_manager.executor_metrics
```

```
281:
282:     metrics = resource_manager.get_executor_metrics("test-exec")
283:     assert metrics["total_executions"] == 1
284:     assert metrics["successful_executions"] == 1
285:
286:     @pytest.mark.asyncio
287:     async def test_circuit_breaker_on_failure(self, resource_manager):
288:         """Circuit breaker should open after repeated failures."""
289:         executor_id = "failing-executor"
290:
291:         await resource_manager.start_executor_execution(executor_id)
292:
293:         for _ in range(5):
294:             await resource_manager.end_executor_execution(
295:                 executor_id=executor_id,
296:                 success=False,
297:                 duration_ms=100.0,
298:                 memory_mb=512.0,
299:             )
300:
301:         breaker = resource_manager.circuit_breakers[executor_id]
302:         assert breaker.state == CircuitState.OPEN
303:
304:     def test_register_allocation_policy(self, resource_manager):
305:         """Should register custom allocation policy."""
306:         policy = ResourceAllocationPolicy(
307:             executor_id="custom-exec",
308:             priority=ExecutorPriority.HIGH,
309:             min_memory_mb=128.0,
310:             max_memory_mb=512.0,
311:             min_workers=1,
312:             max_workers=4,
313:         )
314:
315:         resource_manager.register_allocation_policy(policy)
316:         assert "custom-exec" in resource_manager.allocation_policies
317:
318:     def test_reset_circuit_breaker(self, resource_manager):
319:         """Should reset circuit breaker manually."""
320:         breaker = resource_manager.get_or_create_circuit_breaker("test-exec")
321:         breaker.state = CircuitState.OPEN
322:         breaker.failure_count = 10
323:
324:         success = resource_manager.reset_circuit_breaker("test-exec")
325:         assert success
326:         assert breaker.state == CircuitState.CLOSED
327:         assert breaker.failure_count == 0
328:
329:     def test_get_resource_status(self, resource_manager):
330:         """Should get comprehensive resource status."""
331:         status = resource_manager.get_resource_status()
332:
333:         assert "timestamp" in status
334:         assert "current_pressure" in status
335:         assert "resource_usage" in status
336:         assert "executor_metrics" in status
```

```
337:         assert "circuit_breakers" in status
338:
339:
340: class TestResourceAlertManager:
341:     """Tests for ResourceAlertManager functionality."""
342:
343:     def test_memory_warning_alert(self, alert_manager):
344:         """Should generate memory warning alert."""
345:         from farfan_pipeline.core.orchestrator.resource_manager import (
346:             ResourcePressureEvent,
347:         )
348:
349:         event = ResourcePressureEvent(
350:             timestamp=datetime.utcnow(),
351:             pressure_level=ResourcePressureLevel.HIGH,
352:             cpu_percent=50.0,
353:             memory_mb=800.0,
354:             memory_percent=78.0,
355:             worker_count=8,
356:             active_executors=5,
357:             degradation_applied=[],
358:             circuit_breakers_open=[],
359:             message="High memory usage",
360:         )
361:
362:         alerts = alert_manager.process_event(event)
363:         assert len(alerts) > 0
364:
365:         memory_alerts = [
366:             a for a in alerts if "memory" in a.title.lower()
367:         ]
368:         assert len(memory_alerts) > 0
369:
370:     def test_circuit_breaker_alert(self, alert_manager):
371:         """Should generate circuit breaker alert."""
372:         from farfan_pipeline.core.orchestrator.resource_manager import (
373:             ResourcePressureEvent,
374:         )
375:
376:         event = ResourcePressureEvent(
377:             timestamp=datetime.utcnow(),
378:             pressure_level=ResourcePressureLevel.HIGH,
379:             cpu_percent=50.0,
380:             memory_mb=500.0,
381:             memory_percent=50.0,
382:             worker_count=8,
383:             active_executors=5,
384:             degradation_applied=[],
385:             circuit_breakers_open=["D3-Q3", "D4-Q2"],
386:             message="Circuit breakers opened",
387:         )
388:
389:         alerts = alert_manager.process_event(event)
390:
391:         cb_alerts = [
392:             a for a in alerts if "circuit" in a.title.lower()
```

```
393:         ]
394:         assert len(cb_alerts) > 0
395:
396:     def test_alert_rate_limiting(self, alert_manager):
397:         """Should rate limit repeated alerts."""
398:         from farfan_pipeline.core.orchestrator.resource_manager import (
399:             ResourcePressureEvent,
400:         )
401:
402:         event = ResourcePressureEvent(
403:             timestamp=datetime.utcnow(),
404:             pressure_level=ResourcePressureLevel.HIGH,
405:             cpu_percent=78.0,
406:             memory_mb=500.0,
407:             memory_percent=50.0,
408:             worker_count=8,
409:             active_executors=5,
410:             degradation_applied=[],
411:             circuit_breakers_open=[],
412:             message="High CPU",
413:         )
414:
415:         alerts1 = alert_manager.process_event(event)
416:         alerts2 = alert_manager.process_event(event)
417:
418:         cpu_alerts1 = [a for a in alerts1 if "cpu" in a.title.lower()]
419:         cpu_alerts2 = [a for a in alerts2 if "cpu" in a.title.lower()]
420:
421:         assert len(cpu_alerts1) >= len(cpu_alerts2)
422:
423:     def test_get_alert_summary(self, alert_manager):
424:         """Should get alert summary."""
425:         summary = alert_manager.get_alert_summary()
426:
427:         assert "total_alerts" in summary
428:         assert "last_hour" in summary
429:         assert "last_24_hours" in summary
430:         assert "by_severity" in summary
431:
432:
433: class TestResourceIntegration:
434:     """Tests for resource integration functions."""
435:
436:     def test_create_resource_manager(self, resource_limits):
437:         """Should create resource manager with alerts."""
438:         manager, alerts = create_resource_manager(
439:             resource_limits=resource_limits,
440:             enable_circuit_breakers=True,
441:             enable_degradation=True,
442:             enable_alerts=True,
443:         )
444:
445:         assert manager is not None
446:         assert alerts is not None
447:         assert manager.enable_circuit_breakers
448:         assert manager.enable_degradation
```

```
449:
450:     def test_register_default_policies(self, resource_manager):
451:         """Should register default policies for critical executors."""
452:         register_default_policies(resource_manager)
453:
454:         assert "D3-Q3" in resource_manager.allocation_policies
455:         assert "D4-Q2" in resource_manager.allocation_policies
456:
457:         d3q3_policy = resource_manager.allocation_policies["D3-Q3"]
458:         assert d3q3_policy.priority == ExecutorPriority.CRITICAL
459:
460:
461:
462: =====
463: FILE: tests/test_signal_intelligence_standalone.py
464: =====
465:
466: """
467: Standalone Signal Intelligence Test
468: =====
469:
470: Direct test of signal intelligence modules without triggering full orchestrator import.
471:
472: This bypasses the aggregation.py import error while testing the core functionality.
473: """
474:
475: import json
476: import sys
477: from pathlib import Path
478:
479: # Add src to path
480: sys.path.insert(0, str(Path(__file__).parent.parent / "src"))
481:
482: # Import only what we need (avoid orchestrator.__init__.py chain)
483: from farfan_pipeline.core.orchestrator.signal_semantic_expander import (
484:     expand_pattern_semantically,
485:     expand_all_patterns,
486:     extract_core_term
487: )
488:
489: from farfan_pipeline.core.orchestrator.signal_context_scoper import (
490:     filter_patterns_by_context,
491:     create_document_context,
492:     context_matches
493: )
494:
495: from farfan_pipeline.core.orchestrator.signal_contract_validator import (
496:     validate_with_contract,
497:     execute_failure_contract,
498:     ValidationResult
499: )
500:
501: from farfan_pipeline.core.orchestrator.signal_evidence_extractor import (
502:     extract_structured_evidence,
503:     EvidenceExtractionResult
504: )
```

```
505:  
506:  
507: def load_questionnaire_direct():  
508:     """Load questionnaire directly without canonical loader."""  
509:     qfile = Path("system/config/questionnaire/questionnaire_monolith.json")  
510:     with open(qfile) as f:  
511:         return json.load(f)  
512:  
513:  
514: def test_01_semantic_expansion():  
515:     """Test semantic expansion with real pattern."""  
516:     print("\n" + "="*70)  
517:     print("TEST 1: Semantic Expansion")  
518:     print("="*70)  
519:  
520:     # Real pattern structure  
521:     pattern = {  
522:         'pattern': r'presupuesto\s+asignado',  
523:         'semantic_expansion': 'presupuesto|recursos|financiamiento|fondos',  
524:         'id': 'PAT-TEST-001',  
525:         'confidence_weight': 0.8,  
526:         'category': 'FINANCIAL'  
527:     }  
528:  
529:     variants = expand_pattern_semantically(pattern)  
530:  
531:     print(f"\nâ\234\223 Base pattern: {pattern['pattern']}")  
532:     print(f"â\234\223 Semantic expansion: {pattern['semantic_expansion']}")  
533:     print(f"â\234\223 Generated {len(variants)} variants:")  
534:  
535:     for v in variants:  
536:         status = "ORIGINAL" if not v.get('is_variant') else "VARIANT"  
537:         print(f"  [{status}] {v['id']}: {v['pattern']}")  
538:  
539:     assert len(variants) >= 1  
540:     assert any(v.get('is_variant') for v in variants)  
541:  
542:     print("\nâ\234\205 TEST 1 PASSED")  
543:     return True  
544:  
545:  
546: def test_02_context_filtering():  
547:     """Test context-aware pattern filtering."""  
548:     print("\n" + "="*70)  
549:     print("TEST 2: Context Filtering")  
550:     print("="*70)  
551:  
552:     patterns = [  
553:         {  
554:             'pattern': 'budget pattern',  
555:             'context_requirement': {'section': 'budget'},  
556:             'id': 'PAT-001'  
557:         },  
558:         {  
559:             'pattern': 'global pattern',  
560:             'context_scope': 'global',
```

```
561:         'id': 'PAT-002'
562:     },
563:     {
564:         'pattern': 'indicators pattern',
565:         'context_requirement': {'section': 'indicators'},
566:         'id': 'PAT-003'
567:     }
568: ]
569:
570: # Test budget context
571: context_budget = create_document_context(section='budget', chapter=3)
572: filtered_budget, stats_budget = filter_patterns_by_context(patterns, context_budget)
573:
574: print(f"\nâ\234\223 Budget context filtering:")
575: print(f"  Total patterns: {stats_budget['total_patterns']} ")
576: print(f"  Passed: {stats_budget['passed']} ")
577: print(f"  Filtered: {stats_budget['context_filtered']} + {stats_budget['scope_filtered']} ")
578:
579: # Should have budget pattern + global pattern
580: assert stats_budget['passed'] >= 2
581:
582: # Test indicators context
583: context_indicators = create_document_context(section='indicators', chapter=5)
584: filtered_indicators, stats_indicators = filter_patterns_by_context(patterns, context_indicators)
585:
586: print(f"\nâ\234\223 Indicators context filtering:")
587: print(f"  Total patterns: {stats_indicators['total_patterns']} ")
588: print(f"  Passed: {stats_indicators['passed']} ")
589:
590: assert stats_indicators['passed'] >= 2
591:
592: print("\nâ\234\205 TEST 2 PASSED")
593: return True
594:
595:
596: def test_03_failure_contract():
597:     """Test failure contract validation."""
598:     print("\n" + "="*70)
599:     print("TEST 3: Failure Contract Validation")
600:     print("="*70)
601:
602:     failure_contract = {
603:         'abort_if': ['missing_currency', 'negative_amount'],
604:         'emit_code': 'ERR_BUDGET_001'
605:     }
606:
607:     # Test failure case
608:     result_fail = {
609:         'amount': 1000,
610:         'currency': None  # Missing currency
611:     }
612:
613:     validation_fail = execute_failure_contract(result_fail, failure_contract)
614:
615:     print(f"\nâ\234\223 Failed validation test:")
616:     print(f"  Status: {validation_fail.status}")
```

```
617:     print(f"  Passed: {validation_fail.passed}")
618:     print(f"  Error code: {validation_fail.error_code}")
619:     print(f"  Condition: {validation_fail.condition_violated}")
620:     print(f"  Remediation: {validation_fail.remediation}")
621:
622:     assert not validation_fail.passed
623:     assert validation_fail.error_code == 'ERR_BUDGET_001'
624:     assert validation_fail.condition_violated == 'missing_currency'
625:
626:     # Test success case
627:     result_pass = {
628:         'amount': 1000,
629:         'currency': 'COP'
630:     }
631:
632:     validation_pass = execute_failure_contract(result_pass, failure_contract)
633:
634:     print(f"\nâ\234\223 Passed validation test:")
635:     print(f"  Status: {validation_pass.status}")
636:     print(f"  Passed: {validation_pass.passed}")
637:
638:     assert validation_pass.passed
639:
640:     print("\nâ\234\205 TEST 3 PASSED")
641:     return True
642:
643:
644: def test_04_evidence_extraction():
645:     """Test structured evidence extraction."""
646:     print("\n" + "="*70)
647:     print("TEST 4: Evidence Extraction")
648:     print("="*70)
649:
650:     signal_node = {
651:         'expected_elements': [
652:             {'type': 'baseline_indicator', 'required': True},
653:             {'type': 'meta_objetivo', 'required': True},
654:             {'type': 'series_temporales_aÃ±os', 'minimum': 2},
655:             {'type': 'entidad_responsable', 'required': False},
656:             {'type': 'asignacion_presupuestal', 'required': False}
657:         ],
658:         'patterns': [
659:             {
660:                 'id': 'PAT-BASELINE-001',
661:                 'pattern': 'lÃnea de base|aÃ±o base|situaciÃ³n inicial',
662:                 'confidence_weight': 0.85,
663:                 'category': 'TEMPORAL',
664:                 'match_type': 'substring'
665:             },
666:             {
667:                 'id': 'PAT-TARGET-001',
668:                 'pattern': 'meta|objetivo|alcanzar',
669:                 'confidence_weight': 0.85,
670:                 'category': 'QUANTITATIVE',
671:                 'match_type': 'substring'
672:             },
673:         ],
674:         'signals': [
675:             {
676:                 'id': 'SIGNAL-001',
677:                 'pattern': 'señal|indication',
678:                 'confidence_weight': 0.9,
679:                 'category': 'TEMPORAL',
680:                 'match_type': 'substring'
681:             },
682:             {
683:                 'id': 'SIGNAL-002',
684:                 'pattern': 'señal|indication',
685:                 'confidence_weight': 0.9,
686:                 'category': 'TEMPORAL',
687:                 'match_type': 'substring'
688:             }
689:         ]
690:     }
```

```

673:         {
674:             'id': 'PAT-YEAR-001',
675:             'pattern': r'20\d{2}',
676:             'confidence_weight': 0.9,
677:             'category': 'TEMPORAL',
678:             'match_type': 'regex'
679:         },
680:         {
681:             'id': 'PAT-ENTITY-001',
682:             'pattern': 'SecretarÃ-a|Departamento|Ministerio',
683:             'confidence_weight': 0.8,
684:             'category': 'ENTITY',
685:             'match_type': 'substring'
686:         },
687:         {
688:             'id': 'PAT-BUDGET-001',
689:             'pattern': 'COP|presupuesto|millones',
690:             'confidence_weight': 0.75,
691:             'category': 'QUANTITATIVE',
692:             'match_type': 'substring'
693:         }
694:     ],
695:     'validations': {}
696: }
697:
698: sample_text = """
699: DiagnÃ³stico de GÃ©nero - Municipio de BogotÃ¡;
700:
701: LÃnea de base aÃ±o 2023: 8.5% de mujeres en cargos directivos del sector pÃºblico.
702: Meta establecida: alcanzar 15% para el aÃ±o 2027.
703: Responsable: SecretarÃ-a Distrital de la Mujer
704: Presupuesto asignado: COP 1,500 millones para programas de formaciÃ³n.
705:
706: Fuente: DANE, Encuesta de Empleo 2023
707: """
708:
709: evidence_result = extract_structured_evidence(sample_text, signal_node)
710:
711: print(f"\n\n234\223 Evidence extraction results:")
712: print(f"  Completeness: {evidence_result.completeness:.2%}")
713: print(f"  Evidence types: {len(evidence_result.evidence)}")
714: print(f"  Missing required: {evidence_result.missing_required}")
715: print(f"  Under minimum: {evidence_result.under_minimum}")
716:
717: for element_type, matches in evidence_result.evidence.items():
718:     print(f"\n  [{element_type}] - {len(matches)} matches")
719:     for match in matches[:2]: # Show first 2
720:         print(f"    Value: {match.get('value')}")
721:         print(f"    Confidence: {match.get('confidence', 0):.2f}")
722:         print(f"    Pattern: {match.get('pattern_id')}")
723:
724: # Should extract at least 3 element types using monolith patterns
725: assert len(evidence_result.evidence) >= 3
726: assert evidence_result.completeness > 0.5
727:
728: print("\n\n234\205 TEST 4 PASSED")

```

```
729:     return True
730:
731:
732: def test_05_real_questionnaire_patterns():
733:     """Test with real patterns from questionnaire."""
734:     print("\n" + "="*70)
735:     print("TEST 5: Real Questionnaire Patterns")
736:     print("="*70)
737:
738:     data = load_questionnaire_direct()
739:
740:     blocks = data['blocks']
741:     micro_questions = blocks['micro_questions']
742:
743:     print(f"\nâ\234\223 Loaded questionnaire:")
744:     print(f"  Schema version: {data.get('schema_version')}")
745:     print(f"  Total micro questions: {len(micro_questions)}")
746:
747:     # Get first question
748:     mq = micro_questions[0]
749:     patterns = mq.get('patterns', [])
750:
751:     print(f"\nâ\234\223 First question (Q001):")
752:     print(f"  Text: {mq.get('text', '')[:70]}...")
753:     print(f"  Patterns: {len(patterns)}")
754:     print(f"  Expected elements: {len(mq.get('expected_elements', []))}")
755:
756:     # Test pattern metadata
757:     if patterns:
758:         p = patterns[0]
759:         print(f"\nâ\234\223 Sample pattern metadata:")
760:         print(f"  ID: {p.get('id')}")
761:         print(f"  Pattern: {p.get('pattern', '')[:60]}...")
762:         print(f"  Confidence weight: {p.get('confidence_weight')}")
763:         print(f"  Category: {p.get('category')}")
764:         print(f"  Context scope: {p.get('context_scope')}")
765:         print(f"  Semantic expansion: {p.get('semantic_expansion')}")
766:
767:         assert 'confidence_weight' in p
768:         assert 'category' in p
769:
770:     # Test expansion on all patterns
771:     expanded = expand_all_patterns(patterns[:5], enable_logging=False)  # First 5 patterns
772:
773:     print(f"\nâ\234\223 Pattern expansion:")
774:     print(f"  Original: {len(patterns[:5])}")
775:     print(f"  Expanded: {len(expanded)}")
776:     print(f"  Multiplier: {len(expanded)/len(patterns[:5]):.2f}x")
777:
778:     assert len(expanded) >= len(patterns[:5])
779:
780:     print("\nâ\234\205 TEST 5 PASSED")
781:     return True
782:
783:
784: def test_06_metadata_preservation():
```

```
785:     """Test that metadata is preserved through transformations."""
786:     print("\n" + "="*70)
787:     print("TEST 6: Metadata Preservation")
788:     print("=".join(["="]*70))
789:
790:     original_pattern = {
791:         'pattern': r'tasa\s+de\s+desempleo',
792:         'semantic_expansion': 'tasa|Ã±dice|porcentaje',
793:         'id': 'PAT-PRESERVATION-001',
794:         'confidence_weight': 0.9,
795:         'category': 'ECONOMIC',
796:         'context_scope': 'PARAGRAPH',
797:         'specificity': 'HIGH'
798:     }
799:
800:     # Expand pattern
801:     variants = expand_pattern_semantically(original_pattern)
802:
803:     print(f"\n{n*234}223 Original pattern:")
804:     print(f"  confidence_weight: {original_pattern['confidence_weight']}")
805:     print(f"  category: {original_pattern['category']}")
806:     print(f"  specificity: {original_pattern['specificity']}")
807:
808:     print(f"\n{n*234}223 Checking {len(variants)} variants for metadata preservation...")
809:
810:     for v in variants:
811:         # All variants should have same metadata as original
812:         assert v['confidence_weight'] == original_pattern['confidence_weight']
813:         assert v['category'] == original_pattern['category']
814:         assert v['specificity'] == original_pattern['specificity']
815:
816:         if v.get('is_variant'):
817:             print(f"  {n*234}223 {v['id']}: metadata preserved")
818:
819:     print("\n{n*234}205 TEST 6 PASSED")
820:     return True
821:
822:
823: def test_07_end_to_end_pipeline():
824:     """Test complete end-to-end pipeline."""
825:     print("\n" + "="*70)
826:     print("TEST 7: End-to-End Pipeline")
827:     print("=".join(["="]*70))
828:
829:     # Load real questionnaire
830:     data = load_questionnaire_direct()
831:     mq = data['blocks'][['micro_questions']][0]
832:
833:     patterns = mq.get('patterns', [])
834:
835:     # Step 1: Expand patterns
836:     print(f"\n{n*206}222 Step 1: Semantic expansion")
837:     expanded_patterns = expand_all_patterns(patterns, enable_logging=False)
838:     print(f"  {len(patterns)} {n*206}222 {len(expanded_patterns)} patterns")
839:
840:     # Step 2: Filter by context
```

```
841:     print("\nâ\206\222 Step 2: Context filtering")
842:     context = create_document_context(section='indicators', chapter=2)
843:     filtered, stats = filter_patterns_by_context(expanded_patterns, context)
844:     print(f" {stats['total_patterns']} â\206\222 {stats['passed']} patterns")
845:
846:     # Step 3: Extract evidence
847:     print("\nâ\206\222 Step 3: Evidence extraction")
848:     signal_node = {
849:         'expected_elements': ['baseline_indicator', 'target_value', 'timeline'],
850:         'patterns': filtered,
851:         'validations': mq.get('validations', {})
852:     }
853:
854:     text = "LÃ-nea de base: 8.5%. Meta: 6% para 2027."
855:     evidence = extract_structured_evidence(text, signal_node)
856:     print(f" Completeness: {evidence.completeness:.2%}")
857:     print(f" Extracted: {len(evidence.evidence)} elements")
858:
859:     # Step 4: Validate with contract
860:     print("\nâ\206\222 Step 4: Contract validation")
861:     result = {
862:         'evidence': evidence.evidence,
863:         'completeness': evidence.completeness
864:     }
865:
866:     validation_node = {
867:         'failure_contract': mq.get('failure_contract', {}),
868:         'validations': mq.get('validations', {})
869:     }
870:
871:     validation = validate_with_contract(result, validation_node)
872:     print(f" Status: {validation.status}")
873:     print(f" Passed: {validation.passed}")
874:
875:     print("\nâ\234\205 TEST 7 PASSED")
876:     return True
877:
878:
879: def run_all_tests():
880:     """Run all tests."""
881:     print("\n" + "="*70)
882:     print("SIGNAL INTELLIGENCE LAYER - INTEGRATION TEST SUITE")
883:     print("=".*70)
884:     print("\nRunning severe integration tests (NO MOCKS, REAL DATA)...")
885:
886:     tests = [
887:         test_01_semantic_expansion,
888:         test_02_context_filtering,
889:         test_03_failure_contract,
890:         test_04_evidence_extraction,
891:         test_05_real_questionnaire_patterns,
892:         test_06_metadata_preservation,
893:         test_07_end_to_end_pipeline
894:     ]
895:
896:     passed = 0
```

```
897:     failed = 0
898:
899:     for test in tests:
900:         try:
901:             if test():
902:                 passed += 1
903:         except AssertionError as e:
904:             print(f"\n\t235\214 TEST FAILED: {e}")
905:             failed += 1
906:         except Exception as e:
907:             print(f"\n\t235\214 TEST ERROR: {e}")
908:             import traceback
909:             traceback.print_exc()
910:             failed += 1
911:
912:     print("\n" + "="*70)
913:     print("TEST RESULTS")
914:     print("="*70)
915:     print(f"\n\t234\205 PASSED: {passed}/{len(tests)}")
916:     print(f"\t235\214 FAILED: {failed}/{len(tests)}")
917:
918:     if failed == 0:
919:         print("\n\t237\216\211 ALL TESTS PASSED - SIGNAL INTELLIGENCE LAYER VERIFIED")
920:         return True
921:     else:
922:         print("\n\t232 i.\217 SOME TESTS FAILED - REVIEW REQUIRED")
923:         return False
924:
925:
926: if __name__ == "__main__":
927:     import sys
928:     success = run_all_tests()
929:     sys.exit(0 if success else 1)
930:
931:
932:
933: =====
934: FILE: tests/wiring/__init__.py
935: =====
936:
937: """Signal Intelligence and Wiring Test Suite
938:
939: Comprehensive tests for signal intelligence components and wiring/DI patterns.
940: """
941:
942:
943:
944: =====
945: FILE: tests/wiring/test_evidence_extraction.py
946: =====
947:
948: """Test evidence extraction for all element types.
949:
950: Tests:
951: - Structured evidence extraction using monolith patterns
952: - Element type matching and filtering
```

```
953: - Confidence score propagation
954: - Required element validation
955: - Minimum cardinality checking
956: - Completeness score computation
957: - Evidence deduplication
958: - Pattern relevance filtering
959: """
960:
961: import pytest
962: from farfan_pipeline.core.orchestrator.signal_evidence_extractor import (
963:     extract_structured_evidence,
964:     extract_evidence_for_element_type,
965:     compute_completeness,
966:     EvidenceExtractionResult,
967:     _infer_pattern_categories_for_element,
968:     _is_pattern_relevant_to_element,
969:     _deduplicate_matches,
970: )
971:
972:
973: class TestStructuredEvidenceExtraction:
974:     """Test main evidence extraction function."""
975:
976:     def test_extract_with_simple_signal_node(self):
977:         """Extract evidence with simple signal node."""
978:         signal_node = {
979:             'expected_elements': [
980:                 {'type': 'budget_amount', 'required': True}
981:             ],
982:             'patterns': [
983:                 {
984:                     'pattern': 'presupuesto|recursos',
985:                     'id': 'PAT-001',
986:                     'confidence_weight': 0.85,
987:                     'category': 'QUANTITATIVE',
988:                     'match_type': 'substring'
989:                 }
990:             ],
991:             'validations': {}
992:         }
993:
994:         text = "El presupuesto asignado es COP 1,000,000"
995:
996:         result = extract_structured_evidence(text, signal_node)
997:
998:         assert isinstance(result, EvidenceExtractionResult)
999:         assert 'budget_amount' in result.evidence
1000:        assert len(result.evidence['budget_amount']) > 0
1001:
1002:    def test_returns_evidence_extraction_result(self):
1003:        """Returns EvidenceExtractionResult with all fields."""
1004:        signal_node = {
1005:            'expected_elements': [],
1006:            'patterns': [],
1007:            'validations': {}
1008:        }
```

```
1009:  
1010:     result = extract_structured_evidence("", signal_node)  
1011:  
1012:     assert hasattr(result, 'evidence')  
1013:     assert hasattr(result, 'completeness')  
1014:     assert hasattr(result, 'missing_required')  
1015:     assert hasattr(result, 'under_minimum')  
1016:     assert hasattr(result, 'extraction_metadata')  
1017:  
1018: def test_tracks_missing_required_elements(self):  
1019:     """Tracks required elements that are missing."""  
1020:     signal_node = {  
1021:         'expected_elements': [  
1022:             {'type': 'budget_amount', 'required': True},  
1023:             {'type': 'currency', 'required': True}  
1024:         ],  
1025:         'patterns': [],  
1026:         'validations': {}  
1027:     }  
1028:  
1029:     text = "Some text without required elements"  
1030:  
1031:     result = extract_structured_evidence(text, signal_node)  
1032:  
1033:     assert len(result.missing_required) == 2  
1034:     assert 'budget_amount' in result.missing_required  
1035:     assert 'currency' in result.missing_required  
1036:  
1037: def test_tracks_elements_under_minimum(self):  
1038:     """Tracks elements that don't meet minimum count."""  
1039:     signal_node = {  
1040:         'expected_elements': [  
1041:             {'type': 'sources', 'minimum': 3}  
1042:         ],  
1043:         'patterns': [  
1044:             {  
1045:                 'pattern': 'DANE',  
1046:                 'id': 'PAT-001',  
1047:                 'confidence_weight': 0.9,  
1048:                 'category': 'ENTITY',  
1049:                 'match_type': 'substring'  
1050:             }  
1051:         ],  
1052:         'validations': {}  
1053:     }  
1054:  
1055:     text = "Source: DANE"  
1056:  
1057:     result = extract_structured_evidence(text, signal_node)  
1058:  
1059:     assert len(result.under_minimum) == 1  
1060:     element_type, found, minimum = result.under_minimum[0]  
1061:     assert element_type == 'sources'  
1062:     assert found == 1  
1063:     assert minimum == 3  
1064:
```

```
1065:     def test_extracts_multiple_element_types(self):
1066:         """Extracts multiple different element types."""
1067:         signal_node = {
1068:             'expected_elements': [
1069:                 {'type': 'temporal', 'required': False},
1070:                 {'type': 'quantitative', 'required': False}
1071:             ],
1072:             'patterns': [
1073:                 {
1074:                     'pattern': r'20\d{2}',
1075:                     'id': 'PAT-YEAR',
1076:                     'confidence_weight': 0.9,
1077:                     'category': 'TEMPORAL',
1078:                     'match_type': 'regex'
1079:                 },
1080:                 {
1081:                     'pattern': r'\d+%',
1082:                     'id': 'PAT-PCT',
1083:                     'confidence_weight': 0.85,
1084:                     'category': 'QUANTITATIVE',
1085:                     'match_type': 'regex'
1086:                 }
1087:             ],
1088:             'validations': {}
1089:         }
1090:
1091:         text = "En 2023, el 85% de los recursos fueron ejecutados"
1092:
1093:         result = extract_structured_evidence(text, signal_node)
1094:
1095:         assert 'temporal' in result.evidence
1096:         assert 'quantitative' in result.evidence
1097:         assert len(result.evidence['temporal']) > 0
1098:         assert len(result.evidence['quantitative']) > 0
1099:
1100:    def test_supports_legacy_string_element_format(self):
1101:        """Supports legacy format with element types as strings."""
1102:        signal_node = {
1103:            'expected_elements': ['budget_amount', 'currency'],
1104:            'patterns': [],
1105:            'validations': {}
1106:        }
1107:
1108:        result = extract_structured_evidence("", signal_node)
1109:
1110:        assert 'budget_amount' in result.evidence
1111:        assert 'currency' in result.evidence
1112:
1113:
1114: class TestCompletenessComputation:
1115:     """Test completeness score calculation."""
1116:
1117:     def test_perfect_completeness(self):
1118:         """100% completeness when all elements found."""
1119:         evidence = {
1120:             'element1': [{'value': 'found'}],
```

```
1121:         'element2': [{'value': 'found'}]
1122:     }
1123:     expected_elements = [
1124:         {'type': 'element1', 'required': True},
1125:         {'type': 'element2', 'required': True}
1126:     ]
1127:
1128:     score = compute_completeness(evidence, expected_elements)
1129:
1130:     assert score == 1.0
1131:
1132: def test_zero_completeness(self):
1133:     """0% completeness when no required elements found."""
1134:     evidence = {}
1135:     expected_elements = [
1136:         {'type': 'element1', 'required': True},
1137:         {'type': 'element2', 'required': True}
1138:     ]
1139:
1140:     score = compute_completeness(evidence, expected_elements)
1141:
1142:     assert score == 0.0
1143:
1144: def test_partial_completeness(self):
1145:     """Partial completeness with mixed results."""
1146:     evidence = {
1147:         'element1': [{'value': 'found'}],
1148:         'element2': []
1149:     }
1150:     expected_elements = [
1151:         {'type': 'element1', 'required': True},
1152:         {'type': 'element2', 'required': True}
1153:     ]
1154:
1155:     score = compute_completeness(evidence, expected_elements)
1156:
1157:     assert 0.0 < score < 1.0
1158:     assert score == 0.5
1159:
1160: def test_minimum_count_proportional(self):
1161:     """Minimum count uses proportional scoring."""
1162:     evidence = {
1163:         'sources': [{'value': 'src1'}, {'value': 'src2'}]
1164:     }
1165:     expected_elements = [
1166:         {'type': 'sources', 'minimum': 4}
1167:     ]
1168:
1169:     score = compute_completeness(evidence, expected_elements)
1170:
1171:     assert score == 0.5
1172:
1173: def test_optional_elements_bonus(self):
1174:     """Optional elements provide bonus when present."""
1175:     evidence = {
1176:         'optional1': [{'value': 'found'}]
```

```
1177:         }
1178:         expected_elements = [
1179:             {'type': 'optional1', 'required': False, 'minimum': 0}
1180:         ]
1181:
1182:         score = compute_completeness(evidence, expected_elements)
1183:
1184:         assert score == 1.0
1185:
1186:
1187: class TestPatternRelevanceFiltering:
1188:     """Test pattern relevance to element types."""
1189:
1190:     def test_infer_temporal_categories(self):
1191:         """Infer TEMPORAL category for temporal elements."""
1192:         categories = _infer_pattern_categories_for_element('temporal_year')
1193:
1194:         assert categories is not None
1195:         assert 'TEMPORAL' in categories
1196:
1197:     def test_infer_quantitative_categories(self):
1198:         """Infer QUANTITATIVE category for quantitative elements."""
1199:         categories = _infer_pattern_categories_for_element('indicador_cuantitativo')
1200:
1201:         assert categories is not None
1202:         assert 'QUANTITATIVE' in categories
1203:
1204:     def test_infer_entity_categories(self):
1205:         """Infer ENTITY category for entity elements."""
1206:         categories = _infer_pattern_categories_for_element('entidad_responsable')
1207:
1208:         assert categories is not None
1209:         assert 'ENTITY' in categories
1210:
1211:     def test_infer_geographic_categories(self):
1212:         """Infer GEOGRAPHIC category for territorial elements."""
1213:         categories = _infer_pattern_categories_for_element('cobertura_territorial')
1214:
1215:         assert categories is not None
1216:         assert 'GEOGRAPHIC' in categories
1217:
1218:     def test_accept_all_for_generic_elements(self):
1219:         """Accept all categories for generic elements."""
1220:         categories = _infer_pattern_categories_for_element('generic_element')
1221:
1222:         assert categories is None
1223:
1224:     def test_pattern_relevance_by_keyword_overlap(self):
1225:         """Pattern relevance determined by keyword overlap."""
1226:         pattern_spec = {
1227:             'pattern': 'presupuesto asignado',
1228:             'validation_rule': 'budget_validation',
1229:             'context_requirement': ''
1230:         }
1231:
1232:         is_relevant = _is_pattern_relevant_to_element(
```

```
1233:             'presupuesto asignado',
1234:             'presupuesto_municipal',
1235:             pattern_spec
1236:         )
1237:
1238:         assert is_relevant is True
1239:
1240:     def test_pattern_not_relevant_no_overlap(self):
1241:         """Pattern not relevant when no keyword overlap."""
1242:         pattern_spec = {
1243:             'pattern': 'indicador',
1244:             'validation_rule': '',
1245:             'context_requirement': ''
1246:         }
1247:
1248:         is_relevant = _is_pattern_relevant_to_element(
1249:             'indicador',
1250:             'presupuesto_municipal',
1251:             pattern_spec
1252:         )
1253:
1254:         assert is_relevant is False
1255:
1256:
1257:     class TestEvidenceDeduplication:
1258:         """Test evidence match deduplication."""
1259:
1260:         def test_removes_overlapping_matches(self):
1261:             """Removes overlapping matches keeping highest confidence."""
1262:             matches = [
1263:                 {'value': 'presupuesto', 'confidence': 0.8, 'span': (0, 11)},
1264:                 {'value': 'presupuesto asignado', 'confidence': 0.9, 'span': (0, 20)}
1265:             ]
1266:
1267:             deduplicated = _deduplicate_matches(matches)
1268:
1269:             assert len(deduplicated) == 1
1270:             assert deduplicated[0]['confidence'] == 0.9
1271:
1272:         def test_keeps_non_overlapping_matches(self):
1273:             """Keeps non-overlapping matches."""
1274:             matches = [
1275:                 {'value': 'presupuesto', 'confidence': 0.8, 'span': (0, 11)},
1276:                 {'value': 'recursos', 'confidence': 0.85, 'span': (20, 28)}
1277:             ]
1278:
1279:             deduplicated = _deduplicate_matches(matches)
1280:
1281:             assert len(deduplicated) == 2
1282:
1283:         def test_handles_empty_list(self):
1284:             """Handles empty match list."""
1285:             deduplicated = _deduplicate_matches([])
1286:
1287:             assert deduplicated == []
1288:
```

```
1289:     def test_replaces_with_higher_confidence(self):
1290:         """Replaces overlapping match if significantly higher confidence."""
1291:         matches = [
1292:             {'value': 'test1', 'confidence': 0.5, 'span': (0, 5)},
1293:             {'value': 'test2', 'confidence': 0.9, 'span': (2, 7)}
1294:         ]
1295:
1296:         deduplicated = _deduplicate_matches(matches)
1297:
1298:         assert len(deduplicated) == 1
1299:         assert deduplicated[0]['confidence'] == 0.9
1300:
1301:
1302: if __name__ == "__main__":
1303:     pytest.main([__file__, "-v"])
1304:
1305:
1306:
1307: =====
1308: FILE: tests/wiring/test_evidence_extraction_integration.py
1309: =====
1310:
1311: """
1312: Integration Validation for extract_evidence() with extract_structured_evidence()
1313: =====
1314:
1315: Comprehensive integration tests validating extract_evidence() calling
1316: extract_structured_evidence() with proper expected_elements processing and
1317: completeness metrics, ensuring structured output based on 1,200 element specifications.
1318:
1319: Test Coverage:
1320: 1. Signal Intelligence Layer extract_evidence() Integration
1321:     - Verifies EnrichedSignalPack.extract_evidence() correctly calls extract_structured_evidence()
1322:     - Validates expected_elements from signal nodes are properly passed through
1323:     - Ensures completeness metrics are accurately computed and returned
1324:
1325: 2. Expected Elements Processing (1,200 Specifications)
1326:     - Tests all element types: required, minimum cardinality, optional
1327:     - Validates element type filtering and pattern relevance
1328:     - Ensures confidence score propagation through extraction pipeline
1329:
1330: 3. Completeness Metrics Validation
1331:     - Tests completeness calculation: 0.0 (missing all) to 1.0 (found all)
1332:     - Validates required element tracking (missing_required list)
1333:     - Tests minimum cardinality validation (under_minimum list)
1334:
1335: 4. Structured Output Verification
1336:     - Ensures output is EvidenceExtractionResult, not text blob
1337:     - Validates evidence dictionary structure: element_type \206\222 matches
1338:     - Tests extraction metadata: pattern counts, match counts
1339:
1340: 5. End-to-End Integration
1341:     - Complete flow from signal node \206\222 pattern filtering \206\222 evidence extraction
1342:     - Integration with document context for context-aware extraction
1343:     - Metadata lineage tracking through entire pipeline
1344:
```

```
1345: 6. Real Questionnaire Data
1346:     - Uses actual signal nodes with real expected_elements
1347:     - Tests with diverse element types across dimensions
1348:     - Validates against 1,200 element specification target
1349:
1350: Architecture:
1351: - Tests signal_intelligence_layer.EnrichedSignalPack.extract_evidence()
1352: - Validates signal_evidence_extractor.extract_structured_evidence()
1353: - Uses real questionnaire data (no mocks for signal nodes)
1354: - Comprehensive assertions on completeness scoring
1355: - Validates structured dict output vs unstructured blobs
1356:
1357: Author: F.A.R.F.A.N Pipeline
1358: Date: 2025-12-06
1359: Coverage: extract_evidence() integration with 1,200 element specifications
1360: """
1361:
1362: from typing import Any
1363:
1364: import pytest
1365:
1366: from farfan_pipeline.core.orchestrator.questionnaire import load_questionnaire
1367: from farfan_pipeline.core.orchestrator.signal_evidence_extractor import (
1368:     EvidenceExtractionResult,
1369:     compute_completeness,
1370:     extract_structured_evidence,
1371: )
1372: from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
1373:     EnrichedSignalPack,
1374:     analyze_with_intelligence_layer,
1375: )
1376:
1377:
1378: class MockSignalPack:
1379:     """Mock signal pack for testing."""
1380:
1381:     def __init__(
1382:         self,
1383:         patterns: list[dict[str, Any]],
1384:         micro_questions: list[dict[str, Any]] | None = None,
1385:     ):
1386:         self.patterns = patterns
1387:         self.micro_questions = micro_questions or []
1388:
1389:     def get_node(self, signal_id: str) -> dict[str, Any] | None:
1390:         for mq in self.micro_questions:
1391:             if mq.get("id") == signal_id or mq.get("question_id") == signal_id:
1392:                 return mq
1393:
1394:         return None
1395:
1396: @pytest.fixture(scope="module")
1397: def real_questionnaire():
1398:     """Load real questionnaire data."""
1399:     return load_questionnaire()
1400:
```

```
1401:  
1402: @pytest.fixture(scope="module")  
1403: def real_micro_questions(real_questionnaire):  
1404:     """Extract all micro questions from questionnaire."""  
1405:     return real_questionnaire.get_micro_questions()  
1406:  
1407:  
1408: @pytest.fixture  
1409: def sample_signal_node_with_elements():  
1410:     """Create sample signal node with expected_elements."""  
1411:     return {  
1412:         "id": "TEST_MQ_001",  
1413:         "question_id": "TEST_MQ_001",  
1414:         "expected_elements": [  
1415:             {"type": "baseline_indicator", "required": True},  
1416:             {"type": "target_value", "required": True},  
1417:             {"type": "timeline", "required": False},  
1418:             {"type": "responsible_entity", "minimum": 1},  
1419:             {"type": "budget_amount", "required": False},  
1420:         ],  
1421:         "patterns": [  
1422:             {  
1423:                 "id": "PAT_BASELINE",  
1424:                 "pattern": r"lnea de base|baseline",  
1425:                 "confidence_weight": 0.9,  
1426:                 "category": "QUANTITATIVE",  
1427:                 "match_type": "regex",  
1428:             },  
1429:             {  
1430:                 "id": "PAT_TARGET",  
1431:                 "pattern": r"meta|target|objetivo",  
1432:                 "confidence_weight": 0.85,  
1433:                 "category": "QUANTITATIVE",  
1434:                 "match_type": "regex",  
1435:             },  
1436:             {  
1437:                 "id": "PAT_TIMELINE",  
1438:                 "pattern": r"20\d{2}|a±o|a±os|plazo",  
1439:                 "confidence_weight": 0.8,  
1440:                 "category": "TEMPORAL",  
1441:                 "match_type": "regex",  
1442:             },  
1443:             {  
1444:                 "id": "PAT_ENTITY",  
1445:                 "pattern": r"responsable|secretar±a|entidad|DANE|DNP",  
1446:                 "confidence_weight": 0.75,  
1447:                 "category": "ENTITY",  
1448:                 "match_type": "regex",  
1449:             },  
1450:             {  
1451:                 "id": "PAT_BUDGET",  
1452:                 "pattern": r"presupuesto|recursos|COP|millones",  
1453:                 "confidence_weight": 0.7,  
1454:                 "category": "QUANTITATIVE",  
1455:                 "match_type": "regex",  
1456:             },  
1457:         ]  
1458:     }  
1459:
```

```
1457:     ],
1458:     "validations": {},
1459:     "failure_contract": {
1460:         "condition": "completeness < 0.7",
1461:         "error_code": "E_INSUFFICIENT_EVIDENCE",
1462:         "remediation": "Expand search to additional document sections",
1463:     },
1464: }
1465:
1466:
1467: @pytest.fixture
1468: def sample_document_text():
1469:     """Sample document text with evidence elements."""
1470:     return """
1471:     DiagnÃ³stico de GÃ©nero - Indicadores Clave
1472:
1473:     LÃnea de base aÃ±o 2023: 8.5% de mujeres en cargos directivos del sector pÃºblico.
1474:     SegÃ³n datos del DANE, esta cifra ha permanecido estable en los Ãºltimos 5 aÃ±os.
1475:
1476:     Meta establecida: alcanzar el 15% de participaciÃ³n para el aÃ±o 2027, con revisiones
1477:     intermedias en 2025 (objetivo: 12%).
1478:
1479:     Entidad responsable: SecretarÃ-a de la Mujer y Equidad de GÃ©nero.
1480:     Entidades de apoyo: DNP y ConsejerÃ-a Presidencial para la Equidad de la Mujer.
1481:
1482:     Presupuesto asignado: COP 1,500 millones para el perÃ±odo 2024-2027.
1483:     Recursos adicionales: COP 500 millones de cooperaciÃ³n internacional.
1484: """
1485:
1486:
1487: class TestExtractEvidenceIntegration:
1488:     """Test EnrichedSignalPack.extract_evidence() integration."""
1489:
1490:     def test_extract_evidence_returns_structured_result(
1491:         self, sample_signal_node_with_elements, sample_document_text
1492     ):
1493:         """Test extract_evidence returns EvidenceExtractionResult, not text blob."""
1494:         base_pack = MockSignalPack(sample_signal_node_with_elements["patterns"])
1495:         enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)
1496:
1497:         result = enriched_pack.extract_evidence(
1498:             text=sample_document_text,
1499:             signal_node=sample_signal_node_with_elements,
1500:             document_context=None,
1501:         )
1502:
1503:         # Assert structured result, not blob
1504:         assert isinstance(result, EvidenceExtractionResult)
1505:         assert hasattr(result, "evidence")
1506:         assert hasattr(result, "completeness")
1507:         assert hasattr(result, "missing_required")
1508:         assert hasattr(result, "under_minimum")
1509:         assert hasattr(result, "extraction_metadata")
1510:
1511:         # Evidence should be dict, not string
1512:         assert isinstance(result.evidence, dict)
```

```
1513:         assert not isinstance(result.evidence, str)
1514:
1515:     def test_extract_evidence_processes_expected_elements(
1516:         self, sample_signal_node_with_elements, sample_document_text
1517:     ):
1518:         """Test extract_evidence correctly processes expected_elements."""
1519:         base_pack = MockSignalPack(sample_signal_node_with_elements["patterns"])
1520:         enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)
1521:
1522:         result = enriched_pack.extract_evidence(
1523:             text=sample_document_text,
1524:             signal_node=sample_signal_node_with_elements,
1525:             document_context=None,
1526:         )
1527:
1528:         # Should have extracted evidence for expected element types
1529:         expected_types = [
1530:             e["type"] for e in sample_signal_node_with_elements["expected_elements"]
1531:         ]
1532:
1533:         for element_type in expected_types:
1534:             # Each expected element should have entry in evidence dict
1535:             assert element_type in result.evidence
1536:
1537:             # Should find baseline, target, timeline, entity, budget
1538:             assert len(result.evidence["baseline_indicator"]) > 0
1539:             assert len(result.evidence["target_value"]) > 0
1540:             assert len(result.evidence["timeline"]) > 0
1541:             assert len(result.evidence["responsible_entity"]) > 0
1542:
1543:     def test_extract_evidence_validates_required_elements(
1544:         self, sample_signal_node_with_elements
1545:     ):
1546:         """Test extract_evidence tracks missing required elements."""
1547:         base_pack = MockSignalPack(sample_signal_node_with_elements["patterns"])
1548:         enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)
1549:
1550:         # Text missing required elements
1551:         incomplete_text = "Some text without baseline or target."
1552:
1553:         result = enriched_pack.extract_evidence(
1554:             text=incomplete_text,
1555:             signal_node=sample_signal_node_with_elements,
1556:             document_context=None,
1557:         )
1558:
1559:         # Should track missing required elements
1560:         assert len(result.missing_required) > 0
1561:         assert "baseline_indicator" in result.missing_required
1562:         assert "target_value" in result.missing_required
1563:
1564:         # Completeness should reflect missing required elements
1565:         assert result.completeness < 1.0
1566:
1567:     def test_extract_evidence_validates_minimum_cardinality(self):
1568:         """Test extract_evidence validates minimum element cardinality."""
```

```
1569:     signal_node = {
1570:         "id": "TEST_MIN_CARD",
1571:         "expected_elements": [
1572:             {"type": "sources", "minimum": 3},
1573:         ],
1574:         "patterns": [
1575:             {
1576:                 "id": "PAT_SOURCE",
1577:                 "pattern": r"DANE|DNP|SecretarÃ-a",
1578:                 "confidence_weight": 0.9,
1579:                 "category": "ENTITY",
1580:                 "match_type": "regex",
1581:             }
1582:         ],
1583:         "validations": {}
1584:     }
1585:
1586:     text = "Fuente: DANE. TambiÃ³n consultar DNP."
1587:
1588:     base_pack = MockSignalPack(signal_node["patterns"])
1589:     enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)
1590:
1591:     result = enriched_pack.extract_evidence(
1592:         text=text, signal_node=signal_node, document_context=None
1593:     )
1594:
1595:     # Should find 2 sources (DANE, DNP) but require 3
1596:     assert len(result.under_minimum) > 0
1597:     element_type, found, minimum = result.under_minimum[0]
1598:     assert element_type == "sources"
1599:     assert found < minimum
1600:     assert minimum == 3
1601:
1602:     def test_extract_evidence_computes_completeness_metrics(
1603:         self, sample_signal_node_with_elements, sample_document_text
1604:     ):
1605:         """Test extract_evidence computes accurate completeness score."""
1606:         base_pack = MockSignalPack(sample_signal_node_with_elements["patterns"])
1607:         enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)
1608:
1609:         result = enriched_pack.extract_evidence(
1610:             text=sample_document_text,
1611:             signal_node=sample_signal_node_with_elements,
1612:             document_context=None,
1613:         )
1614:
1615:         # Completeness should be between 0.0 and 1.0
1616:         assert 0.0 <= result.completeness <= 1.0
1617:
1618:         # With comprehensive document, should have high completeness
1619:         assert result.completeness >= 0.7
1620:
1621:         # Should have minimal missing required elements
1622:         assert len(result.missing_required) <= 1
1623:
1624:     def test_extract_evidence_with_partial_data(self):
```

```
1625:     """Test extract_evidence with partially complete data."""
1626:     signal_node = {
1627:         "id": "TEST_PARTIAL",
1628:         "expected_elements": [
1629:             {"type": "baseline", "required": True},
1630:             {"type": "target", "required": True},
1631:             {"type": "timeline", "required": True},
1632:         ],
1633:         "patterns": [
1634:             {
1635:                 "id": "PAT_BASELINE",
1636:                 "pattern": r"baseline",
1637:                 "confidence_weight": 0.9,
1638:                 "category": "QUANTITATIVE",
1639:                 "match_type": "regex",
1640:             },
1641:             {
1642:                 "id": "PAT_TARGET",
1643:                 "pattern": r"target",
1644:                 "confidence_weight": 0.9,
1645:                 "category": "QUANTITATIVE",
1646:                 "match_type": "regex",
1647:             },
1648:             {
1649:                 "id": "PAT_TIMELINE",
1650:                 "pattern": r"20\d{2}",
1651:                 "confidence_weight": 0.8,
1652:                 "category": "TEMPORAL",
1653:                 "match_type": "regex",
1654:             },
1655:         ],
1656:         "validations": {}
1657:     }
1658:
1659: # Text with only baseline, missing target and timeline
1660: partial_text = "The baseline is 10%."
1661:
1662: base_pack = MockSignalPack(signal_node["patterns"])
1663: enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)
1664:
1665: result = enriched_pack.extract_evidence(
1666:     text=partial_text, signal_node=signal_node, document_context=None
1667: )
1668:
1669: # Should find baseline
1670: assert len(result.evidence["baseline"]) > 0
1671:
1672: # Should miss target and timeline
1673: assert "target" in result.missing_required
1674: assert "timeline" in result.missing_required
1675:
1676: # Completeness should be ~0.33 (1 of 3 required)
1677: assert 0.2 <= result.completeness <= 0.5
1678:
1679: def test_extract_evidence_propagates_confidence_scores(
1680:     self, sample_signal_node_with_elements, sample_document_text
```

```
1681:     ):  
1682:         """Test extract_evidence propagates confidence scores from patterns."""  
1683:         base_pack = MockSignalPack(sample_signal_node_with_elements["patterns"])  
1684:         enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)  
1685:  
1686:         result = enriched_pack.extract_evidence(  
1687:             text=sample_document_text,  
1688:             signal_node=sample_signal_node_with_elements,  
1689:             document_context=None,  
1690:         )  
1691:  
1692:         # Check evidence items have confidence scores  
1693:         for element_type, matches in result.evidence.items():  
1694:             for match in matches:  
1695:                 assert "confidence" in match  
1696:                 assert 0.0 <= match["confidence"] <= 1.0  
1697:                 assert "pattern_id" in match  
1698:                 assert "category" in match  
1699:  
1700:     def test_extract_evidence_includes_extraction_metadata(  
1701:         self, sample_signal_node_with_elements, sample_document_text  
1702:     ):  
1703:         """Test extract_evidence includes comprehensive extraction metadata."""  
1704:         base_pack = MockSignalPack(sample_signal_node_with_elements["patterns"])  
1705:         enriched_pack = EnrichedSignalPack(base_pack, enable_semantic_expansion=False)  
1706:  
1707:         result = enriched_pack.extract_evidence(  
1708:             text=sample_document_text,  
1709:             signal_node=sample_signal_node_with_elements,  
1710:             document_context=None,  
1711:         )  
1712:  
1713:         # Should include extraction metadata  
1714:         assert "expected_count" in result.extraction_metadata  
1715:         assert "pattern_count" in result.extraction_metadata  
1716:         assert "total_matches" in result.extraction_metadata  
1717:  
1718:         # Validate metadata values  
1719:         assert result.extraction_metadata["expected_count"] == len(  
1720:             sample_signal_node_with_elements["expected_elements"]  
1721:         )  
1722:         assert result.extraction_metadata["pattern_count"] > 0  
1723:         assert result.extraction_metadata["total_matches"] >= 0  
1724:  
1725:  
1726: class TestExtractStructuredEvidenceIntegration:  
1727:     """Test extract_structured_evidence() directly."""  
1728:  
1729:     def test_extract_structured_evidence_with_dict_elements(  
1730:         self, sample_signal_node_with_elements, sample_document_text  
1731:     ):  
1732:         """Test extract_structured_evidence with dict-format expected_elements."""  
1733:         result = extract_structured_evidence(  
1734:             text=sample_document_text,  
1735:             signal_node=sample_signal_node_with_elements,  
1736:             document_context=None,
```

```
1737:         )
1738:
1739:     assert isinstance(result, EvidenceExtractionResult)
1740:     assert len(result.evidence) > 0
1741:     assert result.completeness > 0.0
1742:
1743:     def test_extract_structured_evidence_with_string_elements(
1744:         self, sample_document_text
1745:     ):
1746:         """Test extract_structured_evidence with legacy string-format elements."""
1747:         signal_node = {
1748:             "id": "TEST_LEGACY",
1749:             "expected_elements": ["baseline", "target", "timeline"],
1750:             "patterns": [
1751:                 {
1752:                     "id": "PAT_BASELINE",
1753:                     "pattern": r"lÃ-nea de base|baseline",
1754:                     "confidence_weight": 0.9,
1755:                     "category": "QUANTITATIVE",
1756:                     "match_type": "regex",
1757:                 },
1758:                 {
1759:                     "id": "PAT_TARGET",
1760:                     "pattern": r"meta|target",
1761:                     "confidence_weight": 0.85,
1762:                     "category": "QUANTITATIVE",
1763:                     "match_type": "regex",
1764:                 },
1765:                 {
1766:                     "id": "PAT_TIMELINE",
1767:                     "pattern": r"20\d{2}",
1768:                     "confidence_weight": 0.8,
1769:                     "category": "TEMPORAL",
1770:                     "match_type": "regex",
1771:                 },
1772:             ],
1773:             "validations": {}
1774:         }
1775:
1776:         result = extract_structured_evidence(
1777:             text=sample_document_text, signal_node=signal_node, document_context=None
1778:         )
1779:
1780:         # Should handle legacy format
1781:         assert "baseline" in result.evidence
1782:         assert "target" in result.evidence
1783:         assert "timeline" in result.evidence
1784:
1785:     def test_extract_structured_evidence_empty_elements(self):
1786:         """Test extract_structured_evidence with no expected elements."""
1787:         signal_node = {
1788:             "id": "TEST_EMPTY",
1789:             "expected_elements": [],
1790:             "patterns": [],
1791:             "validations": {}
1792:         }
```

```
1793:         result = extract_structured_evidence(text="Some text", signal_node=signal_node)
1794: 
1795:     # Should return valid result with completeness 1.0 (nothing expected)
1796:     assert isinstance(result, EvidenceExtractionResult)
1797:     assert result.completeness == 1.0
1798:     assert len(result.missing_required) == 0
1799:     assert len(result.under_minimum) == 0
1800: 
1801: 
1802: 
1803: class TestCompletenessMetricsIntegration:
1804:     """Test completeness metrics calculation."""
1805: 
1806:     def test_completeness_perfect_score(self):
1807:         """Test completeness = 1.0 when all elements found."""
1808:         evidence = {
1809:             "element1": [{"value": "found", "confidence": 0.9}],
1810:             "element2": [{"value": "found", "confidence": 0.8}],
1811:             "element3": [{"value": "found", "confidence": 0.85}],
1812:         }
1813:         expected_elements = [
1814:             {"type": "element1", "required": True},
1815:             {"type": "element2", "required": True},
1816:             {"type": "element3", "required": False},
1817:         ]
1818: 
1819:         completeness = compute_completeness(evidence, expected_elements)
1820: 
1821:         assert completeness == 1.0
1822: 
1823:     def test_completeness_zero_score(self):
1824:         """Test completeness = 0.0 when no required elements found."""
1825:         evidence = {}
1826:         expected_elements = [
1827:             {"type": "element1", "required": True},
1828:             {"type": "element2", "required": True},
1829:         ]
1830: 
1831:         completeness = compute_completeness(evidence, expected_elements)
1832: 
1833:         assert completeness == 0.0
1834: 
1835:     def test_completeness_partial_score(self):
1836:         """Test completeness partial score with mixed results."""
1837:         evidence = {
1838:             "element1": [{"value": "found"}],
1839:             "element2": [],
1840:         }
1841:         expected_elements = [
1842:             {"type": "element1", "required": True},
1843:             {"type": "element2", "required": True},
1844:         ]
1845: 
1846:         completeness = compute_completeness(evidence, expected_elements)
1847: 
1848:         # Should be 0.5 (1 of 2 required found)
```

```
1849:         assert completeness == 0.5
1850:
1851:     def test_completeness_minimum_cardinality(self):
1852:         """Test completeness with minimum cardinality requirements."""
1853:         evidence = {
1854:             "sources": [{"value": "src1"}, {"value": "src2"}],
1855:         }
1856:         expected_elements = [
1857:             {"type": "sources", "minimum": 4},
1858:         ]
1859:
1860:         completeness = compute_completeness(evidence, expected_elements)
1861:
1862:         # Should be 0.5 (2 of 4 minimum found)
1863:         assert completeness == 0.5
1864:
1865:     def test_completeness_optional_elements(self):
1866:         """Test completeness scoring with optional elements."""
1867:         evidence = {
1868:             "required1": [{"value": "found"}],
1869:             "optional1": [{"value": "found"}],
1870:         }
1871:         expected_elements = [
1872:             {"type": "required1", "required": True},
1873:             {"type": "optional1", "required": False, "minimum": 0},
1874:         ]
1875:
1876:         completeness = compute_completeness(evidence, expected_elements)
1877:
1878:         # Required found (1.0) + optional found (1.0) = average 1.0
1879:         assert completeness == 1.0
1880:
1881:
1882: class TestRealQuestionnaireIntegration:
1883:     """Test with real questionnaire data (1,200 element specifications)."""
1884:
1885:     def test_extract_evidence_with_real_signal_nodes(self, real_micro_questions):
1886:         """Test extract_evidence with real signal nodes from questionnaire."""
1887:         # Find signal nodes with expected_elements
1888:         nodes_with_elements = [
1889:             mq for mq in real_micro_questions if mq.get("expected_elements")
1890:         ]
1891:
1892:         assert (
1893:             len(nodes_with_elements) > 0
1894:         ), "No signal nodes with expected_elements found"
1895:
1896:         # Test first 5 nodes with elements
1897:         for mq in nodes_with_elements[:5]:
1898:             base_pack = MockSignalPack(mq.get("patterns", []), [mq])
1899:             enriched_pack = EnrichedSignalPack(
1900:                 base_pack, enable_semantic_expansion=False
1901:             )
1902:
1903:             # Sample text (in real usage, this comes from document)
1904:             sample_text = """
```

```
1905:     LÃnea de base: 10%. Meta: 20% para 2027.
1906:     Responsable: SecretarÃa. Presupuesto: COP 1,000 millones.
1907:     """
1908:
1909:     result = enriched_pack.extract_evidence(
1910:         text=sample_text, signal_node=mq, document_context=None
1911:     )
1912:
1913:     # Validate result structure
1914:     assert isinstance(result, EvidenceExtractionResult)
1915:     assert isinstance(result.evidence, dict)
1916:     assert 0.0 <= result.completeness <= 1.0
1917:     assert isinstance(result.missing_required, list)
1918:     assert isinstance(result.under_minimum, list)
1919:
1920: def test_expected_elements_coverage_across_questionnaire(
1921:     self, real_micro_questions
1922: ):
1923:     """Test expected_elements coverage across questionnaire (target: 1,200)."""
1924:     total_elements = 0
1925:     element_types = set()
1926:     nodes_with_elements = 0
1927:
1928:     for mq in real_micro_questions:
1929:         expected = mq.get("expected_elements", [])
1930:         if expected:
1931:             nodes_with_elements += 1
1932:             total_elements += len(expected)
1933:
1934:             for elem in expected:
1935:                 if isinstance(elem, dict):
1936:                     element_types.add(elem.get("type", ""))
1937:                 elif isinstance(elem, str):
1938:                     element_types.add(elem)
1939:
1940:     print(f"\n  Total signal nodes with expected_elements: {nodes_with_elements}")
1941:     print(f"  Total expected_elements specifications: {total_elements}")
1942:     print(f"  Unique element types: {len(element_types)}")
1943:
1944:     # Should have significant coverage toward 1,200 target
1945:     assert nodes_with_elements > 0
1946:     assert total_elements > 0
1947:
1948: def test_completeness_metrics_with_diverse_nodes(self, real_micro_questions):
1949:     """Test completeness metrics across diverse signal nodes."""
1950:     nodes_with_elements = [
1951:         mq for mq in real_micro_questions if mq.get("expected_elements")
1952:     ][:10]
1953:
1954:     completeness_scores = []
1955:
1956:     for mq in nodes_with_elements:
1957:         signal_node = {
1958:             "id": mq.get("id", "unknown"),
1959:             "expected_elements": mq.get("expected_elements", []),
1960:             "patterns": mq.get("patterns", []),
1961:         }
```

```
1961:         "validations": mq.get("validations", {}),
1962:     }
1963:
1964:     # Test with rich document
1965:     rich_text = """
1966:     DiagnÃ³stico: lÃnea de base 15% en 2023. Meta: 25% para 2027.
1967:     Responsable: SecretarÃ-a de Desarrollo. Fuente: DANE.
1968:     Presupuesto: COP 2,000 millones. Indicadores: tasa de empleo.
1969:     Cobertura territorial: todo el departamento.
1970:     """
1971:
1972:     result = extract_structured_evidence(
1973:         text=rich_text, signal_node=signal_node, document_context=None
1974:     )
1975:
1976:     completeness_scores.append(result.completeness)
1977:
1978:     # Validate distribution of completeness scores
1979:     assert len(completeness_scores) > 0
1980:     assert all(0.0 <= score <= 1.0 for score in completeness_scores)
1981:
1982:     avg_completeness = sum(completeness_scores) / len(completeness_scores)
1983:     print(f"\n  Average completeness across nodes: {avg_completeness:.2f}")
1984:     print(f"  Min completeness: {min(completeness_scores):.2f}")
1985:     print(f"  Max completeness: {max(completeness_scores):.2f}")
1986:
1987:
1988: class TestEndToEndIntegration:
1989:     """Test end-to-end integration with analyze_with_intelligence_layer."""
1990:
1991:     def test_analyze_with_intelligence_layer_includes_evidence(
1992:         self, sample_signal_node_with_elements, sample_document_text
1993:     ):
1994:         """Test analyze_with_intelligence_layer includes structured evidence."""
1995:         result = analyze_with_intelligence_layer(
1996:             text=sample_document_text,
1997:             signal_node=sample_signal_node_with_elements,
1998:             document_context=None,
1999:         )
2000:
2001:         # Should include evidence section
2002:         assert "evidence" in result
2003:         assert isinstance(result["evidence"], dict)
2004:
2005:         # Should include completeness
2006:         assert "completeness" in result
2007:         assert 0.0 <= result["completeness"] <= 1.0
2008:
2009:         # Should include missing_elements
2010:         assert "missing_elements" in result
2011:         assert isinstance(result["missing_elements"], list)
2012:
2013:     def test_analyze_with_intelligence_layer_validation_integration(
2014:         self, sample_signal_node_with_elements, sample_document_text
2015:     ):
2016:         """Test integration between evidence extraction and validation."""
```

```
2017:         result = analyze_with_intelligence_layer(
2018:             text=sample_document_text,
2019:             signal_node=sample_signal_node_with_elements,
2020:             document_context=None,
2021:         )
2022:
2023:     # Should include validation section
2024:     assert "validation" in result
2025:     assert "status" in result["validation"]
2026:     assert "passed" in result["validation"]
2027:
2028:     # Validation should consider completeness
2029:     if result["completeness"] < 0.7:
2030:         # Should fail validation if completeness too low
2031:         assert result["validation"]["passed"] is False
2032:         assert result["validation"]["error_code"] is not None
2033:
2034:     def test_metadata_propagation_through_pipeline(
2035:         self, sample_signal_node_with_elements, sample_document_text
2036:     ):
2037:         """Test metadata propagates through entire pipeline."""
2038:         result = analyze_with_intelligence_layer(
2039:             text=sample_document_text,
2040:             signal_node=sample_signal_node_with_elements,
2041:             document_context=None,
2042:         )
2043:
2044:         # Should include metadata
2045:         assert "metadata" in result
2046:         assert result["metadata"]["intelligence_layer_enabled"] is True
2047:
2048:         # Should include refactorings applied
2049:         assert "refactorings_applied" in result["metadata"]
2050:         refactorings = result["metadata"]["refactorings_applied"]
2051:         assert "evidence_structure" in refactorings
2052:
2053:         # Should include extraction metadata
2054:         assert "expected_count" in result["metadata"]
2055:         assert "pattern_count" in result["metadata"]
2056:
2057:
2058: if __name__ == "__main__":
2059:     pytest.main([__file__, "-v", "-s"])
2060:
2061:
2062:
2063: =====
2064: FILE: tests/wiring/test_pattern_expansion.py
2065: =====
2066:
2067: """Test pattern expansion with semantic enrichment.
2068:
2069: Tests:
2070: - Semantic expansion using semantic_expansion field
2071: - Pattern variant generation (5-10x multiplier)
2072: - Core term extraction heuristics
```

```
2073: - Metadata preservation across variants
2074: - Spanish noun-adjective agreement
2075: - Pattern deduplication
2076: """
2077:
2078: import pytest
2079: from farfan_pipeline.core.orchestrator.signal_semantic_expander import (
2080:     extract_core_term,
2081:     expand_pattern_semantically,
2082:     expand_all_patterns,
2083:     adjust_spanish_agreement,
2084:     validate_expansion_result,
2085: )
2086:
2087:
2088: class TestCoreTermExtraction:
2089:     """Test core term extraction from regex patterns."""
2090:
2091:     def test_extract_simple_term(self):
2092:         """Extract core term from simple pattern."""
2093:         result = extract_core_term("presupuesto")
2094:         assert result == "presupuesto"
2095:
2096:     def test_extract_from_regex_pattern(self):
2097:         """Extract core term from regex with metacharacters."""
2098:         result = extract_core_term(r"presupuesto\s+asignado")
2099:         assert result == "presupuesto"
2100:
2101:     def test_extract_ignores_short_words(self):
2102:         """Core term extraction ignores words with 2 chars."""
2103:         result = extract_core_term(r"el\s+presupuesto")
2104:         assert result == "presupuesto"
2105:
2106:     def test_extract_returns_longest_word(self):
2107:         """Returns longest word as core term."""
2108:         result = extract_core_term(r"gran\s+presupuesto\s+aprobado")
2109:         # Should return longest: "presupuesto" or "aprobado"
2110:         assert result in ["presupuesto", "aprobado"]
2111:         assert len(result) >= 9
2112:
2113:     def test_extract_handles_complex_regex(self):
2114:         """Extract from complex regex with alternation."""
2115:         result = extract_core_term(r"(presupuesto|recursos)\s+asignado")
2116:         assert result in ["presupuesto", "recursos", "asignado"]
2117:
2118:     def test_extract_returns_none_for_empty(self):
2119:         """Returns None for patterns with no extractable terms."""
2120:         result = extract_core_term(r"\d+")
2121:         assert result is None
2122:
2123:
2124: class TestPatternVariantGeneration:
2125:     """Test semantic pattern variant generation."""
2126:
2127:     def test_variant_includes_original(self):
2128:         """Expanded variants always include original pattern."""
```

```
2129:     pattern_spec = {
2130:         'pattern': 'presupuesto asignado',
2131:         'semantic_expansion': 'presupuesto|recursos|fondos',
2132:         'id': 'PAT-001',
2133:         'confidence_weight': 0.8
2134:     }
2135:
2136:     variants = expand_pattern_semantically(pattern_spec)
2137:
2138:     assert len(variants) >= 1
2139:     assert variants[0]['pattern'] == 'presupuesto asignado'
2140:     assert variants[0]['is_variant'] is False
2141:
2142: def test_generates_multiple_variants(self):
2143:     """Generates variants for each synonym."""
2144:     pattern_spec = {
2145:         'pattern': 'presupuesto asignado',
2146:         'semantic_expansion': 'presupuesto|recursos|fondos',
2147:         'id': 'PAT-001',
2148:         'confidence_weight': 0.8
2149:     }
2150:
2151:     variants = expand_pattern_semantically(pattern_spec)
2152:
2153:     # Should have original + 2 variants (recursos, fondos)
2154:     # presupuesto is skipped as it's the core term
2155:     assert len(variants) >= 2
2156:
2157: def test_variant_ids_are_unique(self):
2158:     """Each variant gets unique ID."""
2159:     pattern_spec = {
2160:         'pattern': 'presupuesto asignado',
2161:         'semantic_expansion': 'presupuesto|recursos|fondos',
2162:         'id': 'PAT-001',
2163:         'confidence_weight': 0.8
2164:     }
2165:
2166:     variants = expand_pattern_semantically(pattern_spec)
2167:
2168:     ids = [v['id'] for v in variants]
2169:     assert len(ids) == len(set(ids))  # All unique
2170:
2171: def test_variant_ids_follow_convention(self):
2172:     """Variant IDs follow PAT-##-V# convention."""
2173:     pattern_spec = {
2174:         'pattern': 'presupuesto asignado',
2175:         'semantic_expansion': 'recursos|fondos',
2176:         'id': 'PAT-001',
2177:         'confidence_weight': 0.8
2178:     }
2179:
2180:     variants = expand_pattern_semantically(pattern_spec)
2181:
2182:     for i, variant in enumerate(variants[1:], 1):  # Skip original
2183:         assert variant['id'].endswith(f'-V{i}')
```

```
2185:     def test_variant_preserves_metadata(self):
2186:         """Variants preserve confidence_weight and category."""
2187:         pattern_spec = {
2188:             'pattern': 'presupuesto asignado',
2189:             'semantic_expansion': 'recursos|fondos',
2190:             'id': 'PAT-001',
2191:             'confidence_weight': 0.85,
2192:             'category': 'FINANCIAL',
2193:             'specificity': 'HIGH'
2194:         }
2195:
2196:         variants = expand_pattern_semantically(pattern_spec)
2197:
2198:         for variant in variants:
2199:             assert variant['confidence_weight'] == 0.85
2200:             assert variant['category'] == 'FINANCIAL'
2201:             assert variant['specificity'] == 'HIGH'
2202:
2203:     def test_variant_tracks_source(self):
2204:         """Variants track which pattern they came from."""
2205:         pattern_spec = {
2206:             'pattern': 'presupuesto asignado',
2207:             'semantic_expansion': 'recursos',
2208:             'id': 'PAT-001',
2209:             'confidence_weight': 0.8
2210:         }
2211:
2212:         variants = expand_pattern_semantically(pattern_spec)
2213:
2214:         for variant in variants[1:]: # Skip original
2215:             assert variant['variant_of'] == 'PAT-001'
2216:             assert variant['is_variant'] is True
2217:
2218:     def test_no_expansion_without_semantic_field(self):
2219:         """Returns only original if no semantic_expansion."""
2220:         pattern_spec = {
2221:             'pattern': 'presupuesto asignado',
2222:             'id': 'PAT-001',
2223:             'confidence_weight': 0.8
2224:         }
2225:
2226:         variants = expand_pattern_semantically(pattern_spec)
2227:
2228:         assert len(variants) == 1
2229:         assert variants[0]['pattern'] == 'presupuesto asignado'
2230:
2231:     def test_handles_dict_semantic_expansion(self):
2232:         """Handles semantic_expansion as dict format."""
2233:         pattern_spec = {
2234:             'pattern': 'presupuesto asignado',
2235:             'semantic_expansion': {
2236:                 'presupuesto': ['recursos', 'fondos'],
2237:                 'asignado': ['aprobado', 'destinado']
2238:             },
2239:             'id': 'PAT-001',
2240:             'confidence_weight': 0.8
```

```
2241:     }
2242:
2243:     variants = expand_pattern_semantically(pattern_spec)
2244:
2245:     # Should extract all expansions from dict
2246:     assert len(variants) > 1
2247:
2248:
2249: class TestSpanishAgreement:
2250:     """Test Spanish noun-adjective agreement adjustments."""
2251:
2252:     def test_pluralize_adjective_for_plural_noun(self):
2253:         """Adjusts singular adjective to plural for plural noun."""
2254:         result = adjust_spanish_agreement("fondos asignado", "fondos")
2255:
2256:         assert "asignados" in result
2257:
2258:     def test_common_adjectives_pluralized(self):
2259:         """Common adjectives (asignado, aprobado) are pluralized."""
2260:         test_cases = [
2261:             ("recursos asignado", "recursos", "asignados"),
2262:             ("fondos aprobado", "fondos", "aprobados"),
2263:             ("presupuestos disponible", "presupuestos", "disponibles"),
2264:         ]
2265:
2266:         for pattern, term, expected in test_cases:
2267:             result = adjust_spanish_agreement(pattern, term)
2268:             assert expected in result
2269:
2270:     def test_no_change_for_singular(self):
2271:         """No adjustment for singular nouns."""
2272:         result = adjust_spanish_agreement("presupuesto asignado", "presupuesto")
2273:
2274:         # Should not change singular form
2275:         assert result == "presupuesto asignado"
2276:
2277:
2278: class TestBatchExpansion:
2279:     """Test batch pattern expansion."""
2280:
2281:     def test_expand_all_patterns(self):
2282:         """Expand multiple patterns at once."""
2283:         patterns = [
2284:             {
2285:                 'pattern': 'presupuesto',
2286:                 'semantic_expansion': 'presupuesto|recursos',
2287:                 'id': 'PAT-001',
2288:                 'confidence_weight': 0.8
2289:             },
2290:             {
2291:                 'pattern': 'indicador',
2292:                 'semantic_expansion': 'indicador|m@trica',
2293:                 'id': 'PAT-002',
2294:                 'confidence_weight': 0.75
2295:             }
2296:         ]
```

```
2297:
2298:     expanded = expand_all_patterns(patterns, enable_logging=False)
2299:
2300:     # Should have at least original patterns
2301:     assert len(expanded) >= len(patterns)
2302:
2303: def test_expansion_multiplier(self):
2304:     """Expansion achieves expected multiplier (3-10x)."""
2305:     patterns = [
2306:         {
2307:             'pattern': 'presupuesto',
2308:             'semantic_expansion': 'presupuesto|recursos|fondos|financiamiento',
2309:             'id': 'PAT-001',
2310:             'confidence_weight': 0.8
2311:         }
2312:     ]
2313:
2314:     expanded = expand_all_patterns(patterns, enable_logging=False)
2315:
2316:     # Should have 1 original + 3 variants = 4 patterns
2317:     assert len(expanded) >= 3
2318:
2319: def test_preserves_patterns_without_expansion(self):
2320:     """Patterns without semantic_expansion are preserved."""
2321:     patterns = [
2322:         {
2323:             'pattern': 'presupuesto',
2324:             'id': 'PAT-001',
2325:             'confidence_weight': 0.8
2326:         },
2327:         {
2328:             'pattern': 'indicador',
2329:             'semantic_expansion': 'indicador|m@trica',
2330:             'id': 'PAT-002',
2331:             'confidence_weight': 0.75
2332:         }
2333:     ]
2334:
2335:     expanded = expand_all_patterns(patterns, enable_logging=False)
2336:
2337:     # PAT-001 should still be present
2338:     assert any(p['id'] == 'PAT-001' for p in expanded)
2339:
2340: def test_handles_empty_list(self):
2341:     """Handles empty pattern list."""
2342:     expanded = expand_all_patterns([], enable_logging=False)
2343:
2344:     assert expanded == []
2345:
2346: def test_deduplicates_synonyms(self):
2347:     """Skips variants when synonym matches core term."""
2348:     pattern_spec = {
2349:         'pattern': 'presupuesto asignado',
2350:         'semantic_expansion': 'presupuesto|recursos', # presupuesto is core term
2351:         'id': 'PAT-001',
2352:         'confidence_weight': 0.8
```

```
2353:     }
2354:
2355:     variants = expand_pattern_semantically(pattern_spec)
2356:
2357:     # Should not create variant for "presupuesto" (it's the core term)
2358:     variant_patterns = [v['pattern'] for v in variants[1:]]
2359:     assert not any('presupuesto' in p for p in variant_patterns if 'recursos' not in p)
2360:
2361:
2362: class TestExpansionStatistics:
2363:     """Test expansion statistics and logging."""
2364:
2365:     def test_tracks_expansion_stats(self):
2366:         """Tracks original count, variant count, and multiplier."""
2367:         patterns = [
2368:             {
2369:                 'pattern': 'presupuesto',
2370:                 'semantic_expansion': 'presupuesto|recursos|fondos',
2371:                 'id': 'PAT-001',
2372:                 'confidence_weight': 0.8
2373:             },
2374:             {
2375:                 'pattern': 'indicador',
2376:                 'semantic_expansion': 'indicador|m@trica',
2377:                 'id': 'PAT-002',
2378:                 'confidence_weight': 0.75
2379:             }
2380:         ]
2381:
2382:         expanded = expand_all_patterns(patterns, enable_logging=True)
2383:
2384:         # Should track statistics internally
2385:         assert len(expanded) > len(patterns)
2386:
2387:
2388: class TestEdgeCases:
2389:     """Test edge cases and error handling."""
2390:
2391:     def test_handles_empty_semantic_expansion(self):
2392:         """Handles empty semantic_expansion gracefully."""
2393:         pattern_spec = {
2394:             'pattern': 'presupuesto',
2395:             'semantic_expansion': '',
2396:             'id': 'PAT-001',
2397:             'confidence_weight': 0.8
2398:         }
2399:
2400:         variants = expand_pattern_semantically(pattern_spec)
2401:
2402:         assert len(variants) == 1 # Only original
2403:
2404:     def test_handles_whitespace_in_synonyms(self):
2405:         """Handles whitespace in synonym list."""
2406:         pattern_spec = {
2407:             'pattern': 'presupuesto',
2408:             'semantic_expansion': ' recursos | fondos ',
```

```
2409:         'id': 'PAT-001',
2410:         'confidence_weight': 0.8
2411:     }
2412:
2413:     variants = expand_pattern_semantically(pattern_spec)
2414:
2415:     # Should trim whitespace and generate variants
2416:     assert len(variants) > 1
2417:
2418:     def test_handles_special_regex_chars(self):
2419:         """Handles patterns with regex special characters."""
2420:         pattern_spec = {
2421:             'pattern': r'presupuesto\s+\d+',
2422:             'semantic_expansion': 'presupuesto|recursos',
2423:             'id': 'PAT-001',
2424:             'confidence_weight': 0.8
2425:         }
2426:
2427:         variants = expand_pattern_semantically(pattern_spec)
2428:
2429:         # Should generate variants
2430:         assert len(variants) >= 1
2431:
2432:
2433: class TestExpansionValidation:
2434:     """Test expansion validation function."""
2435:
2436:     def test_validate_expansion_success(self):
2437:         """Test validation with successful 5x expansion."""
2438:         original = [
2439:             {'pattern': 'presupuesto', 'id': 'P1'},
2440:             {'pattern': 'indicador', 'id': 'P2'}
2441:         ]
2442:
2443:         expanded = original + [
2444:             {'pattern': 'recursos', 'id': 'P1-V1', 'is_variant': True, 'variant_of': 'P1'},
2445:             {'pattern': 'fondos', 'id': 'P1-V2', 'is_variant': True, 'variant_of': 'P1'},
2446:             {'pattern': 'financiamiento', 'id': 'P1-V3', 'is_variant': True, 'variant_of': 'P1'},
2447:             {'pattern': 'mÃ©trica', 'id': 'P2-V1', 'is_variant': True, 'variant_of': 'P2'},
2448:             {'pattern': 'medida', 'id': 'P2-V2', 'is_variant': True, 'variant_of': 'P2'},
2449:             {'pattern': 'parÃ¡metro', 'id': 'P2-V3', 'is_variant': True, 'variant_of': 'P2'},
2450:         ]
2451:
2452:         result = validate_expansion_result(original, expanded)
2453:
2454:         assert result['valid'] is True
2455:         assert result['multiplier'] == 4.0
2456:         assert result['meets_minimum'] is True
2457:         assert result['original_count'] == 2
2458:         assert result['expanded_count'] == 8
2459:         assert result['variant_count'] == 6
2460:         assert len(result['issues']) == 0
2461:
2462:     def test_validate_expansion_low_multiplier(self):
2463:         """Test validation with low multiplier."""
2464:         original = [
```

```
2465:         {'pattern': 'presupuesto', 'id': 'P1'},
2466:         {'pattern': 'indicador', 'id': 'P2'}
2467:     ]
2468:
2469:     expanded = original + [
2470:         {'pattern': 'recursos', 'id': 'P1-V1', 'is_variant': True}
2471:     ]
2472:
2473:     result = validate_expansion_result(original, expanded)
2474:
2475:     assert result['valid'] is False
2476:     assert result['multiplier'] == 1.5
2477:     assert result['meets_minimum'] is False
2478:     assert len(result['issues']) > 0
2479:     assert any('below minimum' in issue.lower() for issue in result['issues'])
2480:
2481: def test_validate_expansion_meets_target(self):
2482:     """Test validation when target multiplier is met."""
2483:     original = [{'pattern': 'presupuesto', 'id': 'P1'}]
2484:
2485:     expanded = original + [
2486:         {'pattern': f'variant{i}', 'id': f'P1-V{i}', 'is_variant': True}
2487:         for i in range(1, 5)
2488:     ]
2489:
2490:     result = validate_expansion_result(original, expanded)
2491:
2492:     assert result['valid'] is True
2493:     assert result['multiplier'] == 5.0
2494:     assert result['meets_target'] is True
2495:
2496: def test_validate_expansion_empty_original(self):
2497:     """Test validation with empty original patterns."""
2498:     result = validate_expansion_result([], [])
2499:
2500:     assert result['valid'] is False
2501:     assert result['multiplier'] == 0.0
2502:     assert 'No original patterns' in result['issues'][0]
2503:
2504: def test_validate_expansion_custom_thresholds(self):
2505:     """Test validation with custom min and target multipliers."""
2506:     original = [{'pattern': 'presupuesto', 'id': 'P1'}]
2507:     expanded = original + [
2508:         {'pattern': 'recursos', 'id': 'P1-V1', 'is_variant': True},
2509:         {'pattern': 'fondos', 'id': 'P1-V2', 'is_variant': True}
2510:     ]
2511:
2512:     result = validate_expansion_result(
2513:         original, expanded,
2514:         min_multiplier=2.0,
2515:         target_multiplier=3.0
2516:     )
2517:
2518:     assert result['valid'] is True
2519:     assert result['multiplier'] == 3.0
2520:     assert result['meets_minimum'] is True
```

```
2521:         assert result['meets_target'] is True
2522:
2523:
2524: if __name__ == "__main__":
2525:     pytest.main([__file__, "-v"])
2526:
2527:
2528:
2529: =====
2530: FILE: tests/wiring/test_signal_registry_creation.py
2531: =====
2532:
2533: """Test signal registry creation and initialization.
2534:
2535: Tests:
2536: - Signal registry factory construction
2537: - Type-safe signal pack creation (Pydantic v2)
2538: - Content-based fingerprint generation
2539: - Lazy loading and caching behavior
2540: - Cache hit/miss metrics
2541: - OpenTelemetry integration
2542: """
2543:
2544: import pytest
2545: from unittest.mock import Mock, patch, MagicMock
2546: from pathlib import Path
2547:
2548: from farfan_pipeline.core.orchestrator.signal_registry import (
2549:     QuestionnaireSignalRegistry,
2550:     ChunkingSignalPack,
2551:     MicroAnsweringSignalPack,
2552:     ValidationSignalPack,
2553:     AssemblySignalPack,
2554:     ScoringSignalPack,
2555: )
2556:
2557:
2558: class TestSignalRegistryCreation:
2559:     """Test signal registry instantiation and initialization."""
2560:
2561:     def test_registry_requires_questionnaire(self):
2562:         """Signal registry requires a questionnaire instance."""
2563:         with pytest.raises(TypeError):
2564:             QuestionnaireSignalRegistry()
2565:
2566:     def test_registry_computes_source_hash(self):
2567:         """Registry computes content hash from questionnaire."""
2568:         mock_questionnaire = Mock()
2569:         mock_questionnaire.sha256 = "abc123def456"
2570:         mock_questionnaire.version = "1.0.0"
2571:
2572:         registry = QuestionnaireSignalRegistry(mock_questionnaire)
2573:
2574:         assert hasattr(registry, '_source_hash')
2575:         assert isinstance(registry._source_hash, str)
2576:         assert len(registry._source_hash) > 0
```

```
2577:  
2578:     def test_registry_initializes_lazy_caches(self):  
2579:         """Registry initializes all caches as None (lazy loading)."""  
2580:         mock_questionnaire = Mock()  
2581:         mock_questionnaire.sha256 = "abc123"  
2582:         mock_questionnaire.version = "1.0.0"  
2583:  
2584:         registry = QuestionnaireSignalRegistry(mock_questionnaire)  
2585:  
2586:         assert registry._chunking_signals is None  
2587:         assert len(registry._micro_answering_cache) == 0  
2588:         assert len(registry._validation_cache) == 0  
2589:         assert len(registry._assembly_cache) == 0  
2590:         assert len(registry._scoring_cache) == 0  
2591:  
2592:     def test_registry_tracks_metrics(self):  
2593:         """Registry tracks cache hits and misses."""  
2594:         mock_questionnaire = Mock()  
2595:         mock_questionnaire.sha256 = "abc123"  
2596:         mock_questionnaire.version = "1.0.0"  
2597:  
2598:         registry = QuestionnaireSignalRegistry(mock_questionnaire)  
2599:  
2600:         assert registry._cache_hits == 0  
2601:         assert registry._cache_misses == 0  
2602:         assert registry._signal_loads == 0  
2603:  
2604:  
2605: class TestSignalPackTypes:  
2606:     """Test type-safe signal pack models (Pydantic v2)."""  
2607:  
2608:     def test_chunking_signal_pack_validation(self):  
2609:         """ChunkingSignalPack validates required fields."""  
2610:         with pytest.raises(Exception): # Pydantic ValidationError  
2611:             ChunkingSignalPack(  
2612:                 section_detection_patterns={}, # Empty dict should fail min_length=1  
2613:                 section_weights={},  
2614:                 source_hash="a" * 32  
2615:             )  
2616:  
2617:     def test_chunking_signal_pack_validates_weights(self):  
2618:         """ChunkingSignalPack validates weight ranges [0.0, 2.0]."""  
2619:         with pytest.raises(Exception): # Pydantic ValidationError  
2620:             ChunkingSignalPack(  
2621:                 section_detection_patterns={"budget": ["presupuesto"]},  
2622:                 section_weights={"budget": 3.0}, # Out of range  
2623:                 source_hash="a" * 32  
2624:             )  
2625:  
2626:     def test_chunking_signal_pack_frozen(self):  
2627:         """ChunkingSignalPack is immutable (frozen)."""  
2628:         pack = ChunkingSignalPack(  
2629:             section_detection_patterns={"budget": ["presupuesto"]},  
2630:             section_weights={"budget": 1.0},  
2631:             source_hash="a" * 32  
2632:         )
```

```
2633:
2634:     with pytest.raises(Exception): # Pydantic ValidationError for frozen model
2635:         pack.version = "2.0.0" # type: ignore
2636:
2637:     def test_pattern_item_validates_id_format(self):
2638:         """PatternItem validates ID format PAT-Q##-##.###"""
2639:         from farfan_pipeline.core.orchestrator.signal_registry import PatternItem
2640:
2641:         # Valid ID
2642:         pattern = PatternItem(
2643:             id="PAT-Q001-001",
2644:             pattern="test",
2645:             match_type="REGEX",
2646:             confidence_weight=0.8,
2647:             category="GENERAL"
2648:         )
2649:         assert pattern.id == "PAT-Q001-001"
2650:
2651:         # Invalid ID
2652:         with pytest.raises(Exception): # Pydantic ValidationError
2653:             PatternItem(
2654:                 id="INVALID",
2655:                 pattern="test",
2656:                 match_type="REGEX",
2657:                 confidence_weight=0.8,
2658:                 category="GENERAL"
2659:             )
2660:
2661:     def test_pattern_item_validates_confidence_range(self):
2662:         """PatternItem validates confidence weight [0.0, 1.0]."""
2663:         from farfan_pipeline.core.orchestrator.signal_registry import PatternItem
2664:
2665:         with pytest.raises(Exception): # Pydantic ValidationError
2666:             PatternItem(
2667:                 id="PAT-Q001-001",
2668:                 pattern="test",
2669:                 match_type="REGEX",
2670:                 confidence_weight=1.5, # Out of range
2671:                 category="GENERAL"
2672:             )
2673:
2674:     def test_scoring_signal_pack_validates_quality_levels(self):
2675:         """ScoringSignalPack requires exactly 4 quality levels."""
2676:         from farfan_pipeline.core.orchestrator.signal_registry import QualityLevel
2677:
2678:         quality_levels = [
2679:             QualityLevel(level="EXCELENTE", min_score=0.9, color="green"),
2680:             QualityLevel(level="BUENO", min_score=0.7, color="blue"),
2681:             QualityLevel(level="ACCEPTABLE", min_score=0.5, color="yellow"),
2682:         ] # Only 3 levels
2683:
2684:         with pytest.raises(Exception): # Pydantic ValidationError
2685:             ScoringSignalPack(
2686:                 question_modalities={},
2687:                 modality_configs={},
2688:                 quality_levels=quality_levels, # Should fail min_length=4
```

```
2689:             source_hash="a" * 32
2690:         )
2691:
2692:     def test_modality_config_validates_weights_sum(self):
2693:         """ModalityConfig validates that weights sum to 1.0."""
2694:         from farfan_pipeline.core.orchestrator.signal_registry import ModalityConfig
2695:
2696:         with pytest.raises(Exception): # Pydantic ValidationError
2697:             ModalityConfig(
2698:                 aggregation="weighted_sum",
2699:                 description="Test modality",
2700:                 failure_code="F-A-TEST",
2701:                 weights=[0.5, 0.3] # Sums to 0.8, not 1.0
2702:             )
2703:
2704:
2705:     class TestLazyLoading:
2706:         """Test lazy loading and cache behavior."""
2707:
2708:         @patch.object(QuestionnaireSignalRegistry, '_build_chunking_signals')
2709:         def test_chunking_signals_lazy_load(self, mock_build):
2710:             """Chunking signals are loaded lazily on first access."""
2711:             mock_questionnaire = Mock()
2712:             mock_questionnaire.sha256 = "abc123"
2713:             mock_questionnaire.version = "1.0.0"
2714:
2715:             mock_pack = Mock(spec=ChunkingSignalPack)
2716:             mock_build.return_value = mock_pack
2717:
2718:             registry = QuestionnaireSignalRegistry(mock_questionnaire)
2719:
2720:             # Not loaded yet
2721:             assert registry._chunking_signals is None
2722:             assert not mock_build.called
2723:
2724:             # First access triggers load
2725:             result = registry.get_chunking_signals()
2726:
2727:             assert mock_build.called
2728:             assert result == mock_pack
2729:             assert registry._chunking_signals == mock_pack
2730:
2731:         @patch.object(QuestionnaireSignalRegistry, '_build_chunking_signals')
2732:         def test_chunking_signals_cached_on_second_access(self, mock_build):
2733:             """Chunking signals are cached and not rebuilt on second access."""
2734:             mock_questionnaire = Mock()
2735:             mock_questionnaire.sha256 = "abc123"
2736:             mock_questionnaire.version = "1.0.0"
2737:
2738:             mock_pack = Mock(spec=ChunkingSignalPack)
2739:             mock_build.return_value = mock_pack
2740:
2741:             registry = QuestionnaireSignalRegistry(mock_questionnaire)
2742:
2743:             # First access
2744:             registry.get_chunking_signals()
```

```
2745:         assert mock_build.call_count == 1
2746:
2747:     # Second access
2748:     registry.get_chunking_signals()
2749:     assert mock_build.call_count == 1  # Not called again
2750:
2751: def test_cache_metrics_updated(self):
2752:     """Cache metrics are updated on hits and misses."""
2753:     mock_questionnaire = Mock()
2754:     mock_questionnaire.sha256 = "abc123"
2755:     mock_questionnaire.version = "1.0.0"
2756:     mock_questionnaire.data = {"blocks": {"micro_questions": []}}
2757:
2758:     registry = QuestionnaireSignalRegistry(mock_questionnaire)
2759:
2760:     with patch.object(registry, '_build_chunking_signals', return_value=Mock()):
2761:         # First access = cache miss
2762:         registry.get_chunking_signals()
2763:         assert registry._cache_misses == 1
2764:         assert registry._cache_hits == 0
2765:
2766:         # Second access = cache hit
2767:         registry.get_chunking_signals()
2768:         assert registry._cache_misses == 1
2769:         assert registry._cache_hits == 1
2770:
2771:
2772: class TestContentBasedFingerprints:
2773:     """Test content-based fingerprint generation."""
2774:
2775:     def test_source_hash_uses_blake3_if_available(self):
2776:         """Source hash uses BLAKE3 when available."""
2777:         mock_questionnaire = Mock()
2778:         mock_questionnaire.sha256 = "test_hash"
2779:         mock_questionnaire.version = "1.0.0"
2780:
2781:         with patch('farfan_pipeline.core.orchestrator.signal_registry.BLAKE3_AVAILABLE', True):
2782:             with patch('farfan_pipeline.core.orchestrator.signal_registry.blake3') as mock_blake3:
2783:                 mock_hasher = Mock()
2784:                 mock_hasher.hexdigest.return_value = "blake3_hash"
2785:                 mock_blake3.blake3.return_value = mock_hasher
2786:
2787:                 registry = QuestionnaireSignalRegistry(mock_questionnaire)
2788:
2789:                 assert mock_blake3.blake3.called
2790:                 assert "blake3_hash" in registry._source_hash or True  # Hash may be processed
2791:
2792:     def test_source_hash_fallback_to_sha256(self):
2793:         """Source hash falls back to SHA256 when BLAKE3 not available."""
2794:         mock_questionnaire = Mock()
2795:         mock_questionnaire.sha256 = "test_hash"
2796:         mock_questionnaire.version = "1.0.0"
2797:
2798:         with patch('farfan_pipeline.core.orchestrator.signal_registry.BLAKE3_AVAILABLE', False):
2799:             registry = QuestionnaireSignalRegistry(mock_questionnaire)
2800:
```

```
2801:         assert isinstance(registry._source_hash, str)
2802:         assert len(registry._source_hash) > 0
2803:
2804:     def test_different_questionnaires_produce_different_hashes(self):
2805:         """Different questionnaire content produces different hashes."""
2806:         mock_q1 = Mock()
2807:         mock_q1.sha256 = "hash1"
2808:         mock_q1.version = "1.0.0"
2809:
2810:         mock_q2 = Mock()
2811:         mock_q2.sha256 = "hash2"
2812:         mock_q2.version = "1.0.0"
2813:
2814:         registry1 = QuestionnaireSignalRegistry(mock_q1)
2815:         registry2 = QuestionnaireSignalRegistry(mock_q2)
2816:
2817:         assert registry1._source_hash != registry2._source_hash
2818:
2819:
2820: if __name__ == "__main__":
2821:     pytest.main([__file__, "-v"])
2822:
2823:
2824:
2825: =====
2826: FILE: tools/__init__.py
2827: =====
2828:
2829: """Tools package for utilities and scripts."""
2830:
2831:
2832:
2833: =====
2834: FILE: tools/bulk_import_test.py
2835: =====
2836:
2837: #!/usr/bin/env python3
2838: """
2839: Bulk import test - verifies all modules can be imported
2840: """
2841: import importlib
2842: import sys
2843: from pathlib import Path
2844:
2845: def main() -> None:
2846:     """Test importing all modules in the package."""
2847:     errors = []
2848:     success = []
2849:
2850:     # Add src to path
2851:     # Find all Python modules
2852:     src_path = Path('src/farfan_pipeline')
2853:     if not src_path.exists():
2854:         print(f"Error: {src_path} does not exist")
2855:         sys.exit(1)
2856:
```

```
2857:     for py_file in src_path.glob('*.py'):
2858:         if py_file.name == '__init__.py':
2859:             continue
2860:
2861:         # Convert path to module name
2862:         rel_path = py_file.relative_to('src')
2863:         module_name = str(rel_path.with_suffix('')).replace('/', '.')
2864:
2865:         try:
2866:             importlib.import_module(module_name)
2867:             success.append(module_name)
2868:             print(f'\u234\223 {module_name}')
2869:         except Exception as e:
2870:             errors.append((module_name, str(e)))
2871:             print(f'\u234\227 {module_name}: {e}')
2872:
2873:         print('\n==== Bulk Import Results ===')
2874:         print(f'Success: {len(success)} modules')
2875:         print(f'Errors: {len(errors)} modules')
2876:
2877:         if errors:
2878:             print('\nFailed imports:')
2879:             for module, error in errors[:10]: # Show first 10
2880:                 print(f'  - {module}: {error[:100]}')
2881:             if len(errors) > 10:
2882:                 print(f'  ... and {len(errors) - 10} more')
2883:             sys.exit(1)
2884:         else:
2885:             print('\u234\205 All modules imported successfully')
2886:             sys.exit(0)
2887:
2888: if __name__ == '__main__':
2889:     main()
2890:
2891:
2892: =====
2893: =====
2894: FILE: tools/chunk_semantic_auditor.py
2895: =====
2896:
2897: #!/usr/bin/env python3
2898: """
2899: Chunk Semantic Auditor
2900:
2901: Offline verification tool for semantic integrity of processed policy chunks.
2902: Ensures chunk content aligns with assigned metadata (policy_area_id, dimension_id).
2903:
2904: Usage:
2905:     python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plans/
2906:     python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plans/ --threshold 0.65
2907:     python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plans/ --model-name all-MiniLM-L6-v2
2908:
2909: Exit codes:
2910:     0: All chunks pass semantic integrity checks
2911:     1: One or more chunks fail semantic integrity checks
2912:     2: Configuration or runtime error
```

```
2913: """
2914:
2915: import argparse
2916: import json
2917: import sys
2918: import traceback
2919: from dataclasses import dataclass
2920: from pathlib import Path
2921: from typing import Any
2922:
2923: import numpy as np
2924: from sentence_transformers import SentenceTransformer, util
2925: from tqdm import tqdm
2926:
2927:
2928: @dataclass
2929: class ChunkMetadata:
2930:     chunk_id: str
2931:     file_path: str
2932:     policy_area_id: str
2933:     dimension_id: str
2934:     text_content: str
2935:
2936:
2937: @dataclass
2938: class SemanticAuditResult:
2939:     chunk_id: str
2940:     file_path: str
2941:     policy_area_id: str
2942:     dimension_id: str
2943:     coherence_score: float
2944:     passed: bool
2945:     threshold: float
2946:
2947:
2948: DIMENSION_DESCRIPTIONS = {
2949:     "DIM01": "Inputs dimension: Financial resources, human capital, infrastructure, legal framework, and budgetary allocations required for policy implementation",
2950:     "DIM02": "Activities dimension: Concrete actions, programs, interventions, and operational activities executed to achieve policy objectives",
2951:     "DIM03": "Products dimension: Direct deliverables, tangible outputs, services provided, and immediate results produced by policy activities",
2952:     "DIM04": "Results dimension: Intermediate outcomes, changes in behavior, capacity improvements, and measurable effects on target populations",
2953:     "DIM05": "Impacts dimension: Long-term effects, structural changes, sustainable transformations, and broader societal benefits",
2954:     "DIM06": "Causality dimension: Logical chains, causal relationships, evidence of mechanisms, and explicit theory of change connecting inputs to impacts"
2955: }
2956:
2957: POLICY_AREA_DESCRIPTIONS = {
2958:     "PA01": "Policy area 1: Economic development, productive sectors, business competitiveness, innovation, employment generation, and entrepreneurship support",
2959:     "PA02": "Policy area 2: Social welfare, health services, education systems, vulnerable populations, poverty reduction, and social protection programs",
2960:     "PA03": "Policy area 3: Urban planning, infrastructure development, housing policy, public transportation, utilities, and territorial organization",
2961:     "PA04": "Policy area 4: Environmental protection, natural resources, climate change, sustainable development, biodiversity conservation, and ecological resilience",
2962:     "PA05": "Policy area 5: Public security, justice administration, crime prevention, peace building, human rights protection, and conflict resolution",
2963:     "PA06": "Policy area 6: Cultural development, heritage preservation, arts promotion, sports programs, recreation, and cultural identity strengthening",
2964:     "PA07": "Policy area 7: Institutional capacity, public administration, governance quality, transparency, anti-corruption, and democratic strengthening",
```

```
2965:     "PA08": "Policy area 8: Rural development, agricultural policy, peasant communities, land tenure, food security, and rural infrastructure",
2966:     "PA09": "Policy area 9: Digital transformation, information technology, connectivity, e-government, digital inclusion, and telecommunications infrastructure",
2967:     "PA10": "Policy area 10: Cross-sectoral integration, inter-institutional coordination, multi-dimensional approaches, and systemic policy coherence",
2968: }
2969:
2970:
2971: class ChunkSemanticAuditor:
2972:     def __init__(
2973:         self,
2974:         artifacts_dir: Path,
2975:         threshold: float = 0.7,
2976:         model_name: str = "sentence-transformers/all-MiniLM-L6-v2",
2977:         verbose: bool = False,
2978:     ) -> None:
2979:         self.artifacts_dir = artifacts_dir
2980:         self.threshold = threshold
2981:         self.model_name = model_name
2982:         self.verbose = verbose
2983:         self.model: SentenceTransformer | None = None
2984:         self.chunks: list[ChunkMetadata] = []
2985:         self.audit_results: list[SemanticAuditResult] = []
2986:
2987:     def load_model(self) -> None:
2988:         if self.verbose:
2989:             print(f"Loading sentence transformer model: {self.model_name}")
2990:         self.model = SentenceTransformer(self.model_name)
2991:         if self.verbose:
2992:             print("Model loaded successfully")
2993:
2994:     def discover_chunk_artifacts(self) -> list[Path]:
2995:         if not self.artifacts_dir.exists():
2996:             raise FileNotFoundError(
2997:                 f"Artifacts directory not found: {self.artifacts_dir}"
2998:             )
2999:
3000:         chunk_files = []
3001:         for pattern in ["**/*chunk*.json", "**/*chunks*.json", "**/*segment*.json"]:
3002:             chunk_files.extend(self.artifacts_dir.glob(pattern))
3003:
3004:         if self.verbose:
3005:             print(f"Discovered {len(chunk_files)} chunk artifact files")
3006:
3007:         return sorted(set(chunk_files))
3008:
3009:     def load_chunk_metadata(self, chunk_file: Path) -> list[ChunkMetadata]:
3010:         chunks = []
3011:         try:
3012:             with open(chunk_file, encoding="utf-8") as f:
3013:                 data = json.load(f)
3014:
3015:                 if isinstance(data, list):
3016:                     chunk_list = data
3017:                 elif isinstance(data, dict) and "chunks" in data:
3018:                     chunk_list = data["chunks"]
3019:                 else:
```

```
3020:         if self.verbose:
3021:             print(f"Skipping {chunk_file}: unrecognized format")
3022:         return []
3023:
3024:     for idx, chunk in enumerate(chunk_list):
3025:         chunk_id = (
3026:             chunk.get("chunk_id")
3027:             or chunk.get("id")
3028:             or f"{chunk_file.stem}_{idx}"
3029:         )
3030:         text_content = chunk.get("text") or chunk.get("content") or ""
3031:         policy_area_id = (
3032:             chunk.get("policy_area_id") or chunk.get("policy_area") or ""
3033:         )
3034:         dimension_id = chunk.get("dimension_id") or chunk.get("dimension") or ""
3035:
3036:         if not text_content or not policy_area_id or not dimension_id:
3037:             continue
3038:
3039:         chunks.append(
3040:             ChunkMetadata(
3041:                 chunk_id=chunk_id,
3042:                 file_path=str(chunk_file.relative_to(self.artifacts_dir)),
3043:                 policy_area_id=policy_area_id,
3044:                 dimension_id=dimension_id,
3045:                 text_content=text_content,
3046:             )
3047:         )
3048:
3049:     except json.JSONDecodeError:
3050:         if self.verbose:
3051:             print(f"Skipping {chunk_file}: JSON decode error")
3052:     except Exception as e:
3053:         if self.verbose:
3054:             print(f"Error loading {chunk_file}: {e}")
3055:
3056:     return chunks
3057:
3058: def load_all_chunks(self) -> None:
3059:     chunk_files = self.discover_chunk_artifacts()
3060:     if not chunk_files:
3061:         print("Warning: No chunk files discovered", file=sys.stderr)
3062:         return
3063:
3064:     for chunk_file in tqdm(
3065:         chunk_files, desc="Loading chunks", disable=not self.verbose
3066:     ):
3067:         chunks = self.load_chunk_metadata(chunk_file)
3068:         self.chunks.extend(chunks)
3069:
3070:         if self.verbose:
3071:             print(f"Loaded {len(self.chunks)} chunks with complete metadata")
3072:
3073:     def compute_semantic_coherence(self, chunk: ChunkMetadata) -> float:
3074:         if self.model is None:
3075:             raise RuntimeError("Model not loaded. Call load_model() first.")
```

```
3076:         dimension_desc = DIMENSION_DESCRIPTIONS.get(
3077:             chunk.dimension_id, f"Unknown dimension {chunk.dimension_id}"
3078:         )
3080:         policy_area_desc = POLICY_AREA_DESCRIPTIONS.get(
3081:             chunk.policy_area_id, f"Unknown policy area {chunk.policy_area_id}"
3082:         )
3083:
3084:         canonical_description = f"{dimension_desc}. {policy_area_desc}"
3085:
3086:         chunk_embedding = self.model.encode(chunk.text_content, convert_to_tensor=True)
3087:         canonical_embedding = self.model.encode(
3088:             canonical_description, convert_to_tensor=True
3089:         )
3090:
3091:         similarity = util.cos_sim(chunk_embedding, canonical_embedding).item()
3092:
3093:         return float(similarity)
3094:
3095:     def audit_chunk(self, chunk: ChunkMetadata) -> SemanticAuditResult:
3096:         coherence_score = self.compute_semantic_coherence(chunk)
3097:         passed = coherence_score >= self.threshold
3098:
3099:         return SemanticAuditResult(
3100:             chunk_id=chunk.chunk_id,
3101:             file_path=chunk.file_path,
3102:             policy_area_id=chunk.policy_area_id,
3103:             dimension_id=chunk.dimension_id,
3104:             coherence_score=coherence_score,
3105:             passed=passed,
3106:             threshold=self.threshold,
3107:         )
3108:
3109:     def audit_all_chunks(self) -> None:
3110:         if not self.chunks:
3111:             print("No chunks to audit", file=sys.stderr)
3112:             return
3113:
3114:         print(f"\nAuditing {len(self.chunks)} chunks with threshold={self.threshold}\n")
3115:
3116:         for chunk in tqdm(self.chunks, desc="Auditing chunks"):
3117:             result = self.audit_chunk(chunk)
3118:             self.audit_results.append(result)
3119:
3120:     def generate_report(self) -> dict[str, Any]:
3121:         total_chunks = len(self.audit_results)
3122:         failed_chunks = [r for r in self.audit_results if not r.passed]
3123:         passed_chunks = total_chunks - len(failed_chunks)
3124:
3125:         if total_chunks == 0:
3126:             avg_score = 0.0
3127:             min_score = 0.0
3128:             max_score = 0.0
3129:         else:
3130:             scores = [r.coherence_score for r in self.audit_results]
3131:             avg_score = float(np.mean(scores))
```

```

3132:         min_score = float(np.min(scores))
3133:         max_score = float(np.max(scores))
3134:
3135:     report = {
3136:         "metadata": {
3137:             "artifacts_dir": str(self.artifacts_dir),
3138:             "model_name": self.model_name,
3139:             "threshold": self.threshold,
3140:             "total_chunks audited": total_chunks,
3141:         },
3142:         "summary": {
3143:             "passed": passed_chunks,
3144:             "failed": len(failed_chunks),
3145:             "pass_rate": passed_chunks / total_chunks if total_chunks > 0 else 0.0,
3146:             "average_coherence_score": avg_score,
3147:             "min_coherence_score": min_score,
3148:             "max_coherence_score": max_score,
3149:         },
3150:         "failures": [
3151:             {
3152:                 "chunk_id": r.chunk_id,
3153:                 "file_path": r.file_path,
3154:                 "policy_area_id": r.policy_area_id,
3155:                 "dimension_id": r.dimension_id,
3156:                 "coherence_score": r.coherence_score,
3157:                 "threshold": r.threshold,
3158:             }
3159:             for r in failed_chunks
3160:         ],
3161:     }
3162:
3163:     return report
3164:
3165: def save_report(self, report: dict[str, Any], output_file: Path) -> None:
3166:     with open(output_file, "w", encoding="utf-8") as f:
3167:         json.dump(report, f, indent=2, ensure_ascii=False)
3168:         print(f"\nAudit report saved to: {output_file}")
3169:
3170: def print_summary(self, report: dict[str, Any]) -> None:
3171:     summary = report["summary"]
3172:     metadata = report["metadata"]
3173:
3174:     print("\n" + "=" * 80)
3175:     print("CHUNK SEMANTIC AUDIT SUMMARY")
3176:     print("=" * 80)
3177:     print(f"Artifacts Directory: {metadata['artifacts_dir']}")
3178:     print(f"Model: {metadata['model_name']}")
3179:     print(f"Threshold: {metadata['threshold']:.2f}")
3180:     print(f"Total Chunks: {metadata['total_chunks audited']}")
3181:     print("-" * 80)
3182:     print(f"Passed: {summary['passed']} ({summary['pass_rate']*100:.1f}%)")
3183:     print(f"Failed: {summary['failed']} (((1-summary['pass_rate'])*100:.1f)%)")
3184:     print(f"Average Score: {summary['average_coherence_score']:.4f}")
3185:     print(f"Min Score: {summary['min_coherence_score']:.4f}")
3186:     print(f"Max Score: {summary['max_coherence_score']:.4f}")
3187:     print("-" * 80)

```

```
3188:
3189:     if report["failures"]:
3190:         print("\nFAILURES DETECTED:")
3191:         print("-" * 80)
3192:         for failure in report["failures"]:
3193:             print(f" {failure['chunk_id']}")
3194:             print(f"     File: {failure['file_path']}")
3195:             print(f"     Policy Area: {failure['policy_area_id']}")
3196:             print(f"     Dimension: {failure['dimension_id']}")
3197:             print(
3198:                 f"     Score: {failure['coherence_score']:.4f} (threshold: {failure['threshold']:.2f})"
3199:             )
3200:         print()
3201:
3202:     def run(self) -> int:
3203:         try:
3204:             self.load_model()
3205:             self.load_all_chunks()
3206:
3207:             if not self.chunks:
3208:                 print("ERROR: No valid chunks found to audit", file=sys.stderr)
3209:                 return 2
3210:
3211:             self.audit_all_chunks()
3212:
3213:             report = self.generate_report()
3214:
3215:             output_file = self.artifacts_dir / "semantic_audit_report.json"
3216:             self.save_report(report, output_file)
3217:             self.print_summary(report)
3218:
3219:             return 0 if report["summary"]["failed"] == 0 else 1
3220:
3221:         except Exception as e:
3222:             print(f"ERROR: {e}", file=sys.stderr)
3223:             if self.verbose:
3224:                 traceback.print_exc()
3225:             return 2
3226:
3227:
3228:     def main() -> int:
3229:         parser = argparse.ArgumentParser(
3230:             description="Audit semantic integrity of processed policy chunks",
3231:             formatter_class=argparse.RawDescriptionHelpFormatter,
3232:             epilog="""
3233: Examples:
3234: # Basic audit
3235: python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plan1/
3236:
3237: # Custom threshold
3238: python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plan1/ --threshold 0.65
3239:
3240: # Different model
3241: python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plan1/ --model-name all-mpnet-base-v2
3242:
3243: # Verbose output
```

```
3244:     python tools/chunk_semantic_auditor.py --artifacts-dir artifacts/plan1/ --verbose
3245:         """
3246:     )
3247:
3248:     parser.add_argument(
3249:         "--artifacts-dir",
3250:         type=Path,
3251:         required=True,
3252:         help="Path to artifacts directory (e.g., artifacts/plan1/)",
3253:     )
3254:     parser.add_argument(
3255:         "--threshold",
3256:         type=float,
3257:         default=0.7,
3258:         help="Minimum semantic coherence score (default: 0.7)",
3259:     )
3260:     parser.add_argument(
3261:         "--model-name",
3262:         type=str,
3263:         default="sentence-transformers/all-MiniLM-L6-v2",
3264:         help="Sentence transformer model (default: all-MiniLM-L6-v2)",
3265:     )
3266:     parser.add_argument(
3267:         "--verbose",
3268:         action="store_true",
3269:         help="Enable verbose output",
3270:     )
3271:
3272:     args = parser.parse_args()
3273:
3274:     auditor = ChunkSemanticAuditor(
3275:         artifacts_dir=args.artifacts_dir,
3276:         threshold=args.threshold,
3277:         model_name=args.model_name,
3278:         verbose=args.verbose,
3279:     )
3280:
3281:     return auditor.run()
3282:
3283:
3284: if __name__ == "__main__":
3285:     sys.exit(main())
3286:
3287:
3288:
3289: =====
3290: FILE: tools/detect_cycles.py
3291: =====
3292:
3293: #!/usr/bin/env python3
3294: """Comprehensive import dependency analyzer."""
3295: import ast
3296: import sys
3297: from collections import defaultdict
3298: from dataclasses import dataclass, field
3299: from pathlib import Path
```

```
3300: from typing import Dict, List, Optional, Set, Tuple
3301:
3302:
3303: @dataclass
3304: class ImportInfo:
3305:     module: str
3306:     names: List[str]
3307:     lineno: int
3308:     is_relative: bool
3309:     level: int
3310:
3311:
3312: @dataclass
3313: class CircularChain:
3314:     modules: List[str]
3315:     severity: str
3316:     reason: str
3317:
3318:
3319: @dataclass
3320: class LayerViolation:
3321:     source_module: str
3322:     source_layer: str
3323:     target_module: str
3324:     target_layer: str
3325:     line_number: int
3326:
3327:
3328: class ComprehensiveImportAnalyzer:
3329:     FORBIDDEN = [
3330:         ('core.calibration', 'analysis'),
3331:         ('core.wiring', 'analysis'),
3332:         ('core.orchestrator', 'analysis'),
3333:         ('processing', 'core.orchestrator'),
3334:         ('api', 'processing'),
3335:         ('api', 'analysis'),
3336:         ('utils', 'core.orchestrator'),
3337:     ]
3338:
3339:     def __init__(self, root_path: Path):
3340:         self.root_path = root_path.resolve()
3341:         self.modules = {}
3342:         self.import_graph = defaultdict(set)
3343:         self.cycles = []
3344:         self.violations = []
3345:         self.stats = {'total_modules': 0, 'total_imports': 0, 'relative_imports': 0}
3346:
3347:     def analyze(self):
3348:         py_files = [f for f in self.root_path.rglob('*.*py') if '__pycache__' not in str(f)]
3349:         print(f"Analyzing {len(py_files)} Python files...")
3350:
3351:         for py_file in py_files:
3352:             module_name = self._path_to_module(py_file)
3353:             imports = self._extract_imports(py_file)
3354:             self.modules[module_name] = {'imports': imports, 'layer': self._get_layer(module_name)}
3355:             self.stats['total_modules'] += 1
```



```

3412:             level = getattr(node, 'level', 0)
3413:             imports.append(ImportInfo(node.module or '', [a.name for a in node.names], node.lineno, level > 0, level))
3414:     except:
3415:         pass
3416:     return imports
3417:
3418:     def _find_cycles(self) -> List[CircularChain]:
3419:         visited, stack, cycles = set(), set(), []
3420:
3421:         def dfs(node, path):
3422:             visited.add(node)
3423:             stack.add(node)
3424:             path.append(node)
3425:             for neighbor in self.import_graph.get(node, set()):
3426:                 if neighbor not in visited:
3427:                     dfs(neighbor, path[:])
3428:                 elif neighbor in stack:
3429:                     idx = path.index(neighbor)
3430:                     cycle = path[idx:] + [neighbor]
3431:                     norm = min([cycle[i:] + cycle[:i] for i in range(len(cycle)-1)], key=tuple)
3432:                     if norm not in [c.modules for c in cycles]:
3433:                         sev, reason = self._assess_severity(norm)
3434:                         cycles.append(CircularChain(norm, sev, reason))
3435:                     path.pop()
3436:                     stack.remove(node)
3437:
3438:         for node in self.import_graph:
3439:             if node not in visited:
3440:                 dfs(node, [])
3441:         return cycles
3442:
3443:     def _assess_severity(self, cycle: List[str]) -> Tuple[str, str]:
3444:         n = len(cycle) - 1
3445:         if n > 4:
3446:             return 'CRITICAL', f'{n}-module chain - high risk'
3447:         if n == 2:
3448:             return 'WARNING', 'Two-way circular import'
3449:         return 'BENIGN', 'Circular but likely safe'
3450:
3451:     def _find_violations(self) -> List[LayerViolation]:
3452:         violations = []
3453:         for mod, data in self.modules.items():
3454:             src_layer = data['layer']
3455:             for imp in data['imports']:
3456:                 tgt = self._resolve_relative(mod, imp.level, imp.module) if imp.is_relative else imp.module
3457:                 if 'farfan_pipeline' not in tgt:
3458:                     continue
3459:                 tgt_layer = self._get_layer(tgt)
3460:                 for fsrc, ftgt in self.FORBIDDEN:
3461:                     if fsrc in src_layer and ftgt in tgt_layer:
3462:                         violations.append(LayerViolation(mod, src_layer, tgt, tgt_layer, imp.lineno))
3463:         return violations
3464:
3465:     def generate_report(self, output: Path):
3466:         with open(output, 'w') as f:
3467:             f.write("# IMPORT HEALTH REPORT\n\n")

```

```

3468:         f.write(f"**Analysis Date**: {self._get_timestamp()}\n")
3469:         f.write(f"**Root Path**: '{self.root_path}'\n\n")
3470:
3471:         f.write("## Executive Summary\n\n")
3472:         status = self._get_health_status()
3473:         f.write(f"**Health Status**: {status['icon']} {status['label']}\n\n")
3474:         f.write(f"- **Total Modules Analyzed**: {self.stats['total_modules']}\n")
3475:         f.write(f"- **Total Import Statements**: {self.stats['total_imports']}\n")
3476:         f.write(f"- **Relative Imports**: {self.stats['relative_imports']} ({self._pct(self.stats['relative_imports']), self.stats['total_imports']}%)\n")
3477:         f.write(f"- **Absolute Imports**: {self.stats['total_imports'] - self.stats['relative_imports']} ({self._pct(self.stats['total_imports'] - self.stats['relative_imports'], self.stats['total_imports'])}%)")
3478:         f.write(f"- **Circular Import Chains Found**: {len(self.cycles)}\n")
3479:         f.write(f"- **Layer Violations Detected**: {len(self.violations)}\n\n")
3480:
3481:         f.write("## Import Pattern Statistics\n\n")
3482:         self._write_pattern_stats(f)
3483:
3484:         f.write("\n## Dependency Graph Overview\n\n")
3485:         self._write_graph_stats(f)
3486:
3487:         if self.cycles:
3488:             f.write(f"\n## \u237124 Circular Import Chains ({len(self.cycles)})\n\n")
3489:             critical = [c for c in self.cycles if c.severity == 'CRITICAL']
3490:             warning = [c for c in self.cycles if c.severity == 'WARNING']
3491:             benign = [c for c in self.cycles if c.severity == 'BENIGN']
3492:
3493:             if critical:
3494:                 f.write(f"### \u237124 CRITICAL Issues ({len(critical)})\n\n")
3495:                 for i, c in enumerate(critical, 1):
3496:                     f.write(f"### {i}. {c.reason}\n")
3497:                     f.write(f"**Chain**: '{' \u206222 '.join(c.modules)}\n\n")
3498:                     f.write("**Resolution**: Refactor to break circular dependency. Consider:\n")
3499:                     f.write("- Moving shared code to a common module\n")
3500:                     f.write("- Using dependency injection\n")
3501:                     f.write("- Lazy imports within functions\n")
3502:
3503:             if warning:
3504:                 f.write(f"### \u2321,\u2171 WARNING Issues ({len(warning)})\n\n")
3505:                 for i, c in enumerate(warning, 1):
3506:                     f.write(f"### {i}. {c.reason}\n")
3507:                     f.write(f"**Chain**: '{' \u206222 '.join(c.modules)}\n\n")
3508:
3509:             if benign:
3510:                 f.write(f"### \u2041,\u2171 BENIGN Patterns ({len(benign)})\n\n")
3511:                 for i, c in enumerate(benign, 1):
3512:                     f.write(f"- {' \u206222 '.join(c.modules)}\n")
3513:                     f.write("\n")
3514:         else:
3515:             f.write("\n## \u237125 Circular Import Analysis\n\n")
3516:             f.write("**No circular import chains detected!** The codebase has a clean dependency structure.\n\n")
3517:
3518:         if self.violations:
3519:             f.write(f"\n## \u237126 Layer Violations ({len(self.violations)})\n\n")
3520:             f.write("The following imports violate the layered architecture contracts defined in 'pyproject.toml':\n")
3521:             f.write("|\n| # | Source Module | Source Layer | \u206222 | Target Module | Target Layer | Line |\n")

```

```

3522:         f.write("----|-----|-----|---|-----|-----|-----|\n")
3523:         for i, v in enumerate(self.violations, 1):
3524:             src_short = self._shorten(v.source_module, 40)
3525:             tgt_short = self._shorten(v.target_module, 40)
3526:             f.write(f" {i} | '{src_short}' | '{v.source_layer}' | \206\222 | '{tgt_short}' | '{v.target_layer}' | {v.line_number} |\n")
3527:
3528:         f.write("\n### Resolution Recommendations\n\n")
3529:         self._write_violation_recommendations(f)
3530:     else:
3531:         f.write("\n## \234\205 Layer Architecture Compliance\n\n")
3532:         f.write("**All imports comply with layer architecture!** No violations detected.\n\n")
3533:
3534:         f.write("\n## Relative Import Analysis\n\n")
3535:         self._write_relative_import_analysis(f)
3536:
3537:         f.write("\n## Layer Dependency Matrix\n\n")
3538:         self._write_layer_matrix(f)
3539:
3540:         f.write("\n## Recommendations\n\n")
3541:         self._write_recommendations(f)
3542:
3543:     print(f"Report written to {output}")
3544:
3545: def _get_timestamp(self):
3546:     from datetime import datetime
3547:     return datetime.now().strftime("%Y-%m-%d %H:%M:%S")
3548:
3549: def _pct(self, num, total):
3550:     return round(100 * num / total, 1) if total > 0 else 0
3551:
3552: def _shorten(self, text, maxlen):
3553:     return text if len(text) <= maxlen else f"...{text[-(maxlen-3):]}"
3554:
3555: def _get_health_status(self):
3556:     critical = [c for c in self.cycles if c.severity == 'CRITICAL']
3557:     if critical or len(self.violations) > 5:
3558:         return {'icon': '\237\224', 'label': 'CRITICAL - Immediate action required'}
3559:     if len(self.cycles) > 0 or len(self.violations) > 0:
3560:         return {'icon': '\232\217', 'label': 'WARNING - Issues detected'}
3561:     return {'icon': '\234\205', 'label': 'HEALTHY - No issues found'}
3562:
3563: def _write_pattern_stats(self, f):
3564:     layers = defaultdict(int)
3565:     for mod, data in self.modules.items():
3566:         layers[data['layer']] += 1
3567:
3568:     f.write("## Modules by Layer\n\n")
3569:     for layer in sorted(layers.keys()):
3570:         f.write(f"- **{layer}**: {layers[layer]} modules\n")
3571:
3572: def _write_graph_stats(self, f):
3573:     total_edges = sum(len(deps) for deps in self.import_graph.values())
3574:     modules_with_deps = len([m for m in self.import_graph if self.import_graph[m]])
3575:
3576:     f.write(f"- **Total dependency edges**: {total_edges}\n")
3577:     f.write(f"- **Modules with dependencies**: {modules_with_deps}\n")

```

```

3578:         f.write(f"-- **Average dependencies per module**: {total_edges / modules_with_deps if modules_with_deps else 0:.1f}\n")
3579:
3580:     def _write_relative_import_analysis(self, f):
3581:         invalid = []
3582:         for mod, data in self.modules.items():
3583:             for imp in data['imports']:
3584:                 if imp.is_relative:
3585:                     resolved = self._resolve_relative(mod, imp.level, imp.module)
3586:                     if '<invalid' in resolved:
3587:                         invalid.append((mod, imp, resolved))
3588:
3589:         f.write(f"Total relative imports: {self.stats['relative_imports']}\n\n")
3590:     if invalid:
3591:         f.write(f"### \u232d i.\u2171 Invalid Relative Imports ({len(invalid)})\n\n")
3592:         for mod, imp, resolved in invalid:
3593:             f.write(f"- '{mod}' line {imp.lineno}: level {imp.level} - {resolved}\n")
3594:     else:
3595:         f.write("\u234d\u205 All relative imports are properly scoped.\n")
3596:
3597:     def _write_layer_matrix(self, f):
3598:         layer_deps = defaultdict(lambda: defaultdict(int))
3599:         for mod, data in self.modules.items():
3600:             src_layer = data['layer']
3601:             for imp in data['imports']:
3602:                 tgt = self._resolve_relative(mod, imp.level, imp.module) if imp.is_relative else imp.module
3603:                 if 'farfan_pipeline' in tgt:
3604:                     tgt_layer = self._get_layer(tgt)
3605:                     if src_layer != tgt_layer:
3606:                         layer_deps[src_layer][tgt_layer] += 1
3607:
3608:         layers = sorted(set(list(layer_deps.keys()) + [t for targets in layer_deps.values() for t in targets.keys()]))
3609:
3610:         f.write("Cross-layer dependencies count:\n\n")
3611:         f.write("|\u2022 From \u2022 To |\u2022 |\u2022 ".join(layers) + " |\n")
3612:         f.write("|\u2022 ---|\u2022 " * (len(layers) + 1) + "\n")
3613:
3614:         for src in layers:
3615:             row = [src]
3616:             for tgt in layers:
3617:                 count = layer_deps.get(src, {}).get(tgt, 0)
3618:                 row.append(str(count) if count > 0 else "\u2296")
3619:             f.write("|\u2022 " + " |\u2022 ".join(row) + " |\n")
3620:
3621:     def _writeViolationRecommendations(self, f):
3622:         by_type = defaultdict(list)
3623:         for v in self.violations:
3624:             key = f"{v.source_layer}\u2022 {v.target_layer}"
3625:             by_type[key].append(v)
3626:
3627:             for vtype, vlist in sorted(by_type.items()):
3628:                 f.write(f"### {vtype} ({len(vlist)} violations)\n\n")
3629:                 f.write(self._get_resolution_advice(vtype))
3630:                 f.write("\n")
3631:
3632:     def _get_resolution_advice(self, violation_type):
3633:         advice = {

```

```
3634:     "api à\206\222 processing": "API layer should only call orchestrator. Move logic to orchestrator entry points.",  
3635:     "api à\206\222 analysis": "API layer should only call orchestrator. Move logic to orchestrator entry points.",  
3636:     "utils à\206\222 core.orchestrator": "Utils must remain leaf modules. Extract shared code or use dependency injection.",  
3637:     "processing à\206\222 core.orchestrator": "Processing modules should not import orchestrator. Use ports/interfaces instead.",  
3638:     "core.orchestrator à\206\222 analysis": "Orchestrator should not directly import analysis. Use dynamic loading or registry pattern.",  
3639:   }  
3640:   return advice.get(violation_type, "Review layer architecture and refactor to comply with contracts.\n")  
3641:  
3642: def _write_recommendations(self, f):  
3643:     f.write("### General Recommendations\n\n")  
3644:     f.write("1. **Maintain layer boundaries**: Respect the architecture contracts in `pyproject.toml`\n")  
3645:     f.write("2. **Avoid circular imports**: Use dependency injection, lazy imports, or refactor shared code\n")  
3646:     f.write("3. **Minimize cross-layer dependencies**: Keep coupling low between architectural layers\n")  
3647:     f.write("4. **Use relative imports carefully**: Ensure they stay within package boundaries\n")  
3648:     f.write("5. **Regular analysis**: Run this tool regularly to catch issues early\n\n")  
3649:  
3650:     if self.violations:  
3651:         f.write("### Priority Actions\n\n")  
3652:         f.write(f"1. Fix {len(self.violations)} layer violation(s)\n")  
3653:         if [c for c in self.cycles if c.severity == 'CRITICAL']:  
3654:             f.write(f"2. Resolve CRITICAL circular imports immediately\n")  
3655:         if [c for c in self.cycles if c.severity == 'WARNING']:  
3656:             f.write(f"3. Review and fix WARNING-level circular imports\n")  
3657:  
3658:  
3659: def main():  
3660:     root = Path(__file__).parent.parent / 'src' / 'farfan_pipeline'  
3661:     if not root.exists():  
3662:         print(f"Error: {root} not found")  
3663:     return 1  
3664:  
3665:     analyzer = ComprehensiveImportAnalyzer(root)  
3666:     analyzer.analyze()  
3667:  
3668:     report_path = Path(__file__).parent.parent.parent / 'IMPORT_HEALTH_REPORT.md'  
3669:     analyzer.generate_report(report_path)  
3670:  
3671:     print(f"\n{'='*80}")  
3672:     print("ANALYSIS COMPLETE")  
3673:     print(f"{'='*80}")  
3674:     print(f"Modules: {analyzer.stats['total_modules']}")  
3675:     print(f"Circular chains: {len(analyzer.cycles)}")  
3676:     print(f"Layer violations: {len(analyzer.violations)}")  
3677:  
3678:     return 0 if not [c for c in analyzer.cycles if c.severity == 'CRITICAL'] else 1  
3679:  
3680:  
3681: if __name__ == '__main__':  
3682:     sys.exit(main())  
3683:  
3684:  
3685:  
3686: ======  
3687: FILE: tools/grep_boundary_checks.py  
3688: ======
```

```
3690: """Grep-based boundary violation detector for architectural guardrails."""
3691: from __future__ import annotations
3692:
3693: import re
3694: import subprocess
3695: import sys
3696: from pathlib import Path
3697: from typing import TYPE_CHECKING
3698:
3699: try:
3700:     from farfan_pipeline.config.paths import PROJECT_ROOT
3701: except Exception: # pragma: no cover - bootstrap fallback
3702:     PROJECT_ROOT = Path(__file__).resolve().parents[1]
3703:
3704: if TYPE_CHECKING:
3705:     from collections.abc import Sequence
3706:
3707: REPO_ROOT = PROJECT_ROOT
3708:
3709: class BoundaryViolation(Exception):
3710:     """Raised when a boundary violation is detected."""
3711:
3712:
3713: def run_grep(pattern: str, paths: Sequence[str]) -> list[str]:
3714:     """Run grep command and return matching lines."""
3715:     try:
3716:         cmd = ["grep", "-rn", "--include=*.py", pattern, *paths]
3717:         result = subprocess.run(
3718:             cmd,
3719:             cwd=REPO_ROOT,
3720:             capture_output=True,
3721:             text=True,
3722:             check=False,
3723:         )
3724:         if result.returncode == 0:
3725:             return result.stdout.strip().split("\n")
3726:         return []
3727:     except Exception as exc:
3728:         print(f"Warning: grep command failed: {exc}", file=sys.stderr)
3729:         return []
3730:
3731:
3732: def check_no_orchestrator_imports_in_core() -> None:
3733:     """Ensure core and executors don't import from orchestrator."""
3734:     print("Checking: core/executors must not import orchestrator...")
3735:
3736:     patterns = [
3737:         r"import\s+orchestrator",
3738:         r"from\s+orchestrator\s+import",
3739:     ]
3740:
3741:     violations = []
3742:     for pattern in patterns:
3743:         matches = run_grep(pattern, ["core", "executors"])
3744:         if matches and matches != []:
3745:             violations.extend(matches)
```

```
3746:  
3747:     if violations:  
3748:         print(f"\u235\214 Found {len(violations)} orchestrator import violations in core/executors:")  
3749:         for violation in violations[:10]: # Show first 10  
3750:             print(f"  {violation}")  
3751:         raise BoundaryViolation("core/executors must not import orchestrator")  
3752:  
3753:     print("  \u234\223 No orchestrator imports in core/executors")  
3754:  
3755:  
3756: def check_no_provider_calls_in_core() -> None:  
3757:     """Ensure core doesn't call orchestrator provider functions."""  
3758:     print("Checking: core must not call get_questionnaire_provider...")  
3759:  
3760:     pattern = r"get_questionnaire_provider\s*\("br/>3761:     matches = run_grep(pattern, ["core", "executors"])  
3762:  
3763:     if matches and matches != [""]:  
3764:         print(f"\u235\214 Found {len(matches)} provider calls in core/executors:")  
3765:         for match in matches[:10]:  
3766:             print(f"  {match}")  
3767:         raise BoundaryViolation("core/executors must not call orchestrator providers")  
3768:  
3769:     print("  \u234\223 No provider calls in core/executors")  
3770:  
3771:  
3772: def check_no_json_io_in_core() -> None:  
3773:     """Ensure core doesn't perform direct JSON file I/O."""  
3774:     print("Checking: core must not perform JSON file I/O...")  
3775:  
3776:     # More specific pattern: open() with .json inside parentheses  
3777:     pattern = r'open\([^\)]*\.\json[^)]*\)'  
3778:     matches = run_grep(pattern, ["core"])  
3779:  
3780:     if matches and matches != [""]:  
3781:         print(f"\u235\214 Found {len(matches)} JSON I/O operations in core:")  
3782:         for match in matches[:10]:  
3783:             print(f"  {match}")  
3784:         raise BoundaryViolation("core must not perform direct JSON I/O")  
3785:  
3786:     print("  \u234\223 No JSON I/O in core")  
3787:  
3788:  
3789: def main() -> None:  
3790:     """Run all boundary checks."""  
3791:     print("== Grep-based Boundary Checks ==\n")  
3792:  
3793:     checks = [  
3794:         check_no_orchestrator_imports_in_core,  
3795:         check_no_provider_calls_in_core,  
3796:         check_no_json_io_in_core,  
3797:     ]  
3798:  
3799:     failed = []  
3800:     for check in checks:  
3801:         try:
```

```
3802:         check()
3803:     except BoundaryViolation as exc:
3804:         failed.append(str(exc))
3805:
3806:     if failed:
3807:         print(f"\n{n}\235\214 {len(failed)} boundary check(s) failed")
3808:         sys.exit(1)
3809:
3810:     print("\n{n}\234\223 All grep-based boundary checks passed")
3811:
3812:
3813: if __name__ == "__main__":
3814:     main()
3815:
3816:
3817:
3818: =====
3819: FILE: tools/hardcoding_audit_scanner.py
3820: =====
3821:
3822: #!/usr/bin/env python3
3823: """
3824: Comprehensive Hardcoding Audit Scanner
3825:
3826: Detects calibration hardcoding violations:
3827: 1. Calibration values (scores, weights, thresholds, coefficients) in .py files
3828: 2. Inline JSON/dict literals containing calibration data
3829: 3. YAML file references (prohibited format)
3830: 4. Undeclared Bayesian priors
3831:
3832: Generates violations_audit.md with file/line/code context.
3833: """
3834:
3835: import ast
3836: import re
3837: import sys
3838: from pathlib import Path
3839: from typing import List, Dict, Set, Tuple
3840: from dataclasses import dataclass, field
3841:
3842:
3843: @dataclass
3844: class Violation:
3845:     """Represents a hardcoding violation."""
3846:     file_path: str
3847:     line_number: int
3848:     violation_type: str
3849:     code_snippet: str
3850:     context: str = ""
3851:     severity: str = "HIGH"
3852:
3853:
3854: @dataclass
3855: class AuditResult:
3856:     """Results of hardcoding audit."""
3857:     violations: List[Violation] = field(default_factory=list)
```

```
3858:     files_scanned: int = 0
3859:     yaml_references: Set[str] = field(default_factory=set)
3860:
3861:     def addViolation(self, v: Violation) -> None:
3862:         self.violations.append(v)
3863:
3864:     def get_by_type(self, vtype: str) -> List[Violation]:
3865:         return [v for v in self.violations if v.violation_type == vtype]
3866:
3867:
3868: class CalibrationHardcodingDetector(ast.NodeVisitor):
3869:     """AST visitor to detect hardcoded calibration values."""
3870:
3871:     CALIBRATION_KEYWORDS = {
3872:         'score', 'weight', 'threshold', 'coefficient', 'alpha', 'beta', 'gamma',
3873:         'b_theory', 'b_impl', 'b_deploy', 'prior', 'posterior', 'likelihood',
3874:         'calibration', 'layer', 'choquet', 'intrinsic', 'runtime',
3875:         'w_th', 'w_imp', 'w_dep', 'g_function', 'sigmoidal_k', 'sigmoidal_x0',
3876:         'abort_threshold', 'compatibility_level', 'alignment', 'default_score'
3877:     }
3878:
3879:     CALIBRATION_PATTERNS = [
3880:         r'\b(score|weight|threshold|coefficient)\s*[=:]\s*[0-9.]+',
3881:         r'b_(theory|impl|deploy)\s*[=:]\s*[0-9.]+',
3882:         r'(alpha|beta|gamma|prior)\s*[=:]\s*[0-9.]+',
3883:         r'@(b|chain|q|d|p|C|u|m)\s*[=:]\s*[0-9.]+',
3884:     ]
3885:
3886:     def __init__(self, file_path: str, source_lines: List[str]):
3887:         self.file_path = file_path
3888:         self.source_lines = source_lines
3889:         self.violations: List[Violation] = []
3890:         self.in_dict_context = False
3891:         self.dict_depth = 0
3892:
3893:     def visit_Assign(self, node: ast.Assign) -> None:
3894:         """Detect hardcoded values in assignments."""
3895:         for target in node.targets:
3896:             if isinstance(target, ast.Name):
3897:                 var_name = target.id.lower()
3898:
3899:                 # Check if variable name suggests calibration
3900:                 if any(kw in var_name for kw in self.CALIBRATION_KEYWORDS):
3901:                     if isinstance(node.value, ast.Constant):
3902:                         if isinstance(node.value.value, (int, float)):
3903:                             self._addViolation(
3904:                                 node.lineno,
3905:                                 "HARDCODED_CALIBRATION_VALUE",
3906:                                 f"Variable '{target.id}' assigned hardcoded value: {node.value.value}",
3907:                                 "HIGH"
3908:                             )
3909:                         elif isinstance(node.value, ast.Dict):
3910:                             self._addViolation(
3911:                                 node.lineno,
3912:                                 "INLINE_CALIBRATION_DICT",
3913:                                 f"Variable '{target.id}' assigned inline dict literal",
```

```
3914:                     "HIGH"
3915:                 )
3916:
3917:             self.generic_visit(node)
3918:
3919:     def visit_Dict(self, node: ast.Dict) -> None:
3920:         """Detect inline dict literals with calibration data."""
3921:         self.dict_depth += 1
3922:
3923:         # Check if dict contains calibration keywords in keys
3924:         has_calibration_keys = False
3925:         calibration_keys = []
3926:
3927:         for key in node.keys():
3928:             if isinstance(key, ast.Constant) and isinstance(key.value, str):
3929:                 key_lower = key.value.lower()
3930:                 if any(kw in key_lower for kw in self.CALIBRATION_KEYWORDS):
3931:                     has_calibration_keys = True
3932:                     calibration_keys.append(key.value)
3933:
3934:         if has_calibration_keys and self.dict_depth <= 2:
3935:             # Only flag top-level or shallow dicts to avoid false positives
3936:             self._addViolation(
3937:                 node.lineno,
3938:                 "INLINE_CALIBRATION_DICT",
3939:                 f"Dict literal contains calibration keys: {', '.join(calibration_keys[:3])}",
3940:                 "HIGH"
3941:             )
3942:
3943:             self.generic_visit(node)
3944:             self.dict_depth -= 1
3945:
3946:     def visit_Call(self, node: ast.Call) -> None:
3947:         """Detect json.loads() with inline JSON or undeclared priors."""
3948:         if isinstance(node.func, ast.Attribute):
3949:             if node.func.attr == 'loads' and isinstance(node.func.value, ast.Name):
3950:                 if node.func.value.id == 'json':
3951:                     # Check if json.loads is called with string literal
3952:                     if node.args and isinstance(node.args[0], ast.Constant):
3953:                         if isinstance(node.args[0].value, str):
3954:                             content = node.args[0].value.lower()
3955:                             if any(kw in content for kw in self.CALIBRATION_KEYWORDS):
3956:                                 self._addViolation(
3957:                                     node.lineno,
3958:                                     "INLINE_JSON_CALIBRATION",
3959:                                     "json.loads() called with inline JSON containing calibration data",
3960:                                     "CRITICAL"
3961:                               )
3962:
3963: # Detect Bayesian prior declarations
3964:         if isinstance(node.func, ast.Name):
3965:             func_name = node.func.id.lower()
3966:             if 'prior' in func_name or 'beta' in func_name or 'gamma' in func_name or 'dirichlet' in func_name:
3967:                 # Check if it's from scipy.stats or similar
3968:                 self._addViolation(
3969:                     node.lineno,
```

```
3970:             "UNDECLARED_BAYESIAN_PRIOR",
3971:             f"Potential undeclared Bayesian prior: {node.func.id}()", 
3972:             "MEDIUM"
3973:         )
3974:
3975:         self.generic_visit(node)
3976:
3977:     def visit_Constant(self, node: ast.Constant) -> None:
3978:         """Detect numeric constants in suspicious contexts."""
3979:         if isinstance(node.value, (int, float)):
3980:             # Check if the constant is in typical calibration range
3981:             if 0.0 <= node.value <= 1.0 or 0 <= node.value <= 100:
3982:                 # Get parent context to determine if it's suspicious
3983:                 # This is simplified; full implementation would track parent nodes
3984:                 pass
3985:
3986:             self.generic_visit(node)
3987:
3988:     def _addViolation(self, line_no: int, vtype: str, context: str, severity: str) -> None:
3989:         """Add a violation to the list."""
3990:         code_snippet = self._get_code_snippet(line_no)
3991:
3992:         self.violations.append(Violation(
3993:             file_path=self.file_path,
3994:             line_number=line_no,
3995:             violation_type=vtype,
3996:             code_snippet=code_snippet,
3997:             context=context,
3998:             severity=severity
3999:         ))
4000:
4001:     def _get_code_snippet(self, line_no: int, context_lines: int = 2) -> str:
4002:         """Get code snippet with context."""
4003:         start = max(0, line_no - context_lines - 1)
4004:         end = min(len(self.source_lines), line_no + context_lines)
4005:
4006:         lines = []
4007:         for i in range(start, end):
4008:             marker = ">>>" if i == line_no - 1 else "    "
4009:             lines.append(f"{marker} {i+1:4d}: {self.source_lines[i].rstrip()}")
4010:
4011:         return "\n".join(lines)
4012:
4013:
4014: class RegexCalibrationScanner:
4015:     """Regex-based scanner for patterns AST might miss."""
4016:
4017:     PATTERNS = [
4018:         # Hardcoded numeric assignments to calibration variables
4019:         (r'(\w*(?:score|weight|threshold|coefficient)\w*)\s*[=:]\s*([0-9.]+)', 
4020:          'HARDCODED_CALIBRATION_VALUE'),
4021:
4022:         # Inline calibration dicts
4023:         (r'(?:intrinsic|runtime|layer).*\{.*?["\'](?:score|weight|threshold)["\'].*\?\}', 
4024:          'INLINE_CALIBRATION_DICT'),
4025:
```

```

4026:     # YAML references
4027:     (r'\.ya?ml["\']?',
4028:      'YAML_REFERENCE'),
4029:
4030:     # @b, @chain, etc. with values
4031:     (r'@(?::b|chain|q|d|p|C|u|m)\s*[=:]\s*([0-9.]+)',
4032:      'HARDCODED_LAYER_SCORE'),
4033:
4034:     # Bayesian priors
4035:     (r'(:scipy\.stats\.|pymc3\.|pymc\.)(:beta|gamma|normal|dirichlet)\s*\(',
4036:      'UNDECLARED_BAYESIAN_PRIOR'),
4037:
4038:     # Choquet weights
4039:     (r'choquet.*?(:weight|coefficient)s?\s*[=:]\s*[\{\}]',
4040:      'HARDCODED_CHOQUET_WEIGHTS'),
4041: ]
4042:
4043: def scan_file(self, file_path: str, content: str, lines: List[str]) -> List[Violation]:
4044:     """Scan file content with regex patterns."""
4045:     violations = []
4046:
4047:     for pattern, vtype in self.PATTERNS:
4048:         for match in re.finditer(pattern, content, re.IGNORECASE | re.DOTALL):
4049:             # Find line number
4050:             line_no = content[:match.start()].count('\n') + 1
4051:
4052:             # Get code snippet
4053:             start = max(0, line_no - 3)
4054:             end = min(len(lines), line_no + 2)
4055:             snippet_lines = []
4056:             for i in range(start, end):
4057:                 marker = ">>>" if i == line_no - 1 else "    "
4058:                 snippet_lines.append(f'{marker} {i+1:4d}: {lines[i].rstrip()}')
4059:             snippet = '\n'.join(snippet_lines)
4060:
4061:             violations.append(Violation(
4062:                 file_path=file_path,
4063:                 line_number=line_no,
4064:                 violation_type=vtype,
4065:                 code_snippet=snippet,
4066:                 context=f"Pattern matched: {match.group(0)[:80]}",
4067:                 severity="HIGH" if vtype in ['YAML_REFERENCE', 'INLINE_CALIBRATION_DICT'] else "MEDIUM"
4068:             ))
4069:
4070:     return violations
4071:
4072:
4073: def scan_python_file(file_path: Path) -> List[Violation]:
4074:     """Scan a Python file for hardcoding violations."""
4075:     try:
4076:         with open(file_path, 'r', encoding='utf-8') as f:
4077:             content = f.read()
4078:             lines = content.split('\n')
4079:
4080:     violations = []
4081:
```

```
4082:     # AST-based detection
4083:     try:
4084:         tree = ast.parse(content, filename=str(file_path))
4085:         detector = CalibrationHardcodingDetector(str(file_path), lines)
4086:         detector.visit(tree)
4087:         violations.extend(detector.violations)
4088:     except SyntaxError as e:
4089:         violations.append(Violation(
4090:             file_path=str(file_path),
4091:             line_number=e.lineno or 0,
4092:             violation_type="PARSE_ERROR",
4093:             code_snippet=f"Syntax error: {e}",
4094:             severity="LOW"
4095:         ))
4096:
4097:     # Regex-based detection
4098:     regex_scanner = RegexCalibrationScanner()
4099:     violations.extend(regex_scanner.scan_file(str(file_path), content, lines))
4100:
4101:     return violations
4102:
4103: except Exception as e:
4104:     return [Violation(
4105:         file_path=str(file_path),
4106:         line_number=0,
4107:         violation_type="SCAN_ERROR",
4108:         code_snippet=f"Error scanning file: {e}",
4109:         severity="LOW"
4110:     )]
4111:
4112:
4113: def scan_directory(root_path: Path, include_patterns: List[str] = None) -> AuditResult:
4114:     """Scan directory tree for violations."""
4115:     if include_patterns is None:
4116:         include_patterns = ['src/farfan_pipeline/**/*.py']
4117:
4118:     result = AuditResult()
4119:
4120:     # Collect all Python files
4121:     python_files = []
4122:     for pattern in include_patterns:
4123:         python_files.extend(root_path.glob(pattern))
4124:
4125:     # Scan each file
4126:     for py_file in python_files:
4127:         if py_file.is_file():
4128:             result.files_scanned += 1
4129:             violations = scan_python_file(py_file)
4130:             result.violations.extend(violations)
4131:
4132:         # Check for YAML references
4133:         yaml_violets = [v for v in violations if v.violation_type == 'YAML_REFERENCE']
4134:         if yaml_violets:
4135:             result.yaml_references.add(str(py_file))
4136:
4137:     return result
```

```
4138:  
4139:  
4140: def generate_markdown_report(result: AuditResult, output_path: Path) -> None:  
4141:     """Generate violations_audit.md report."""  
4142:  
4143:     with open(output_path, 'w', encoding='utf-8') as f:  
4144:         f.write("# Calibration Hardcoding Audit Report\n\n")  
4145:         f.write("## Executive Summary\n\n")  
4146:         f.write(f"-- **Files Scanned**: {result.files_scanned}\n")  
4147:         f.write(f"-- **Total Violations**: {len(result.violations)}\n")  
4148:         f.write(f"-- **CRITICAL Violations**: {len([v for v in result.violations if v.severity == 'CRITICAL'])}\n")  
4149:         f.write(f"-- **HIGH Violations**: {len([v for v in result.violations if v.severity == 'HIGH'])}\n")  
4150:         f.write(f"-- **MEDIUM Violations**: {len([v for v in result.violations if v.severity == 'MEDIUM'])}\n")  
4151:         f.write(f"-- **Files with YAML References**: {len(result.yaml_references)}\n\n")  
4152:  
4153:     # Violation categories  
4154:     f.write("## Violation Categories\n\n")  
4155:  
4156:     violation_types = {}  
4157:     for v in result.violations:  
4158:         violation_types.setdefault(v.violation_type, []).append(v)  
4159:  
4160:     for vtype in sorted(violation_types.keys()):  
4161:         f.write(f"### {vtype.replace('_', ' ')}.title() ({len(violation_types[vtype])} occurrences)\n\n")  
4162:  
4163:     # Known Violators Section  
4164:     f.write("## Known Violators (Priority Review)\n\n")  
4165:  
4166:     known_violators = [  
4167:         'src/farfan_pipeline/core/calibration/orchestrator.py',  
4168:         'src/farfan_pipeline/core/calibration/layer_computers.py'  
4169:     ]  
4170:  
4171:     for known_file in known_violators:  
4172:         f.write(f"### {known_file}\n\n")  
4173:         fileViolations = [v for v in result.violations if known_file in v.file_path]  
4174:  
4175:         if fileViolations:  
4176:             for v in sorted(fileViolations, key=lambda x: x.line_number):  
4177:                 f.write(f"**Line {v.line_number}** - '{v.violation_type}' [{v.severity}]\n\n")  
4178:                 f.write(f"**Context**: {v.context}\n")  
4179:                 f.write("```python\n")  
4180:                 f.write(v.code_snippet)  
4181:                 f.write("\n```\n")  
4182:         else:  
4183:             f.write("*No violations detected (may require manual review)*\n\n")  
4184:  
4185:     # All Violations by File  
4186:     f.write("## Detailed Violations by File\n\n")  
4187:  
4188:     violations_by_file = {}  
4189:     for v in result.violations:  
4190:         violations_by_file.setdefault(v.file_path, []).append(v)  
4191:  
4192:     for file_path in sorted(violations_by_file.keys()):  
4193:         # Skip known violators (already covered)
```

```
4194:         if any(kv in file_path for kv in known_violators):
4195:             continue
4196:
4197:             f.write(f"### {file_path}\n\n")
4198:             file_violets = violations_by_file[file_path]
4199:             f.write(f"**{len(file_violets)} violation(s)**\n\n")
4200:
4201:             for v in sorted(file_violets, key=lambda x: x.line_number):
4202:                 f.write(f"#### Line {v.line_number} - '{v.violation_type}' [{v.severity}]\n\n")
4203:                 if v.context:
4204:                     f.write(f"**{v.context}**\n")
4205:                     f.write("``python\n")
4206:                     f.write(v.code_snippet)
4207:                     f.write("\n```\n")
4208:
4209:             # YAML References
4210:             if result.yaml_references:
4211:                 f.write("## YAML File References (PROHIBITED)\n\n")
4212:                 f.write("**CRITICAL**: YAML is a prohibited format for calibration data.\n\n")
4213:                 for yaml_file in sorted(result.yaml_references):
4214:                     f.write(f"- {yaml_file}\n")
4215:                     f.write("\n")
4216:
4217:             # Recommendations
4218:             f.write("## Remediation Recommendations\n\n")
4219:             f.write("1. **Move all calibration values to JSON config files**:\n")
4220:             f.write("  - 'config/intrinsic_calibration.json' for @b scores\n")
4221:             f.write("  - 'config/contextual_parametrization.json' for layer parameters\n\n")
4222:
4223:             f.write("2. **Remove inline dict/JSON literals**:\n")
4224:             f.write("  - Load all calibration data via 'IntrinsicCalibrationLoader'\n")
4225:             f.write("  - Use 'CalibrationOrchestrator' as single entry point\n\n")
4226:
4227:             f.write("3. **Eliminate YAML references**:\n")
4228:             f.write("  - Convert any YAML files to JSON\n")
4229:             f.write("  - Update all file references\n\n")
4230:
4231:             f.write("4. **Declare Bayesian priors explicitly**:\n")
4232:             f.write("  - Document all priors in calibration config\n")
4233:             f.write("  - Add prior justification comments\n\n")
4234:
4235:             f.write("5. **Use CalibrationOrchestrator exclusively**:\n")
4236:             f.write("  - Remove direct score computations\n")
4237:             f.write("  - Route all calibration through 'calibrate_method()' \n\n")
4238:
4239:
4240: def main():
4241:     """Main entry point."""
4242:     root = Path.cwd()
4243:
4244:     print("=" * 80)
4245:     print("FARFAN Calibration Hardcoding Audit Scanner")
4246:     print("=" * 80)
4247:     print()
4248:
4249:     print("Scanning Python files for calibration hardcoding violations...")
```

```
4250:     print()
4251:
4252:     # Scan directory
4253:     result = scan_directory(root)
4254:
4255:     print(f"Scanned {result.files_scanned} files")
4256:     print(f"Found {len(result.violations)} violations")
4257:     print()
4258:
4259:     # Generate report
4260:     output_path = root / "violations_audit.md"
4261:     generate_markdown_report(result, output_path)
4262:
4263:     print(f"Report generated: {output_path}")
4264:     print()
4265:
4266:     # Summary by severity
4267:     critical = len([v for v in result.violations if v.severity == 'CRITICAL'])
4268:     high = len([v for v in result.violations if v.severity == 'HIGH'])
4269:     medium = len([v for v in result.violations if v.severity == 'MEDIUM'])
4270:
4271:     print("Violations by Severity:")
4272:     print(f"  CRITICAL: {critical}")
4273:     print(f"  HIGH: {high}")
4274:     print(f"  MEDIUM: {medium}")
4275:     print()
4276:
4277:     # Known violators
4278:     print("Known Violator Files:")
4279:     for known in ['orchestrator.py', 'layer_computers.py']:
4280:         count = len([v for v in result.violations if known in v.file_path])
4281:         print(f"  {known}: {count} violations")
4282:     print()
4283:
4284:     if result.yaml_references:
4285:         print(f"\u232a CRITICAL: {len(result.yaml_references)} files reference YAML (prohibited format)")
4286:         print()
4287:
4288:     print("=" * 80)
4289:     print(f"Audit complete. Review {output_path} for details.")
4290:     print("=" * 80)
4291:
4292:     # Exit with error code if critical violations found
4293:     if critical > 0:
4294:         sys.exit(1)
4295:
4296:
4297: if __name__ == '__main__':
4298:     main()
4299:
4300:
4301:
4302: =====
4303: FILE: tools/integrity/__init__.py
4304: =====
4305:
```

```
4306: """Integrity checking tools."""
4307:
4308:
4309:
4310: =====
4311: FILE: tools/lint/check_pythonpath_references.py
4312: =====
4313:
4314: #!/usr/bin/env python3
4315: """
4316: Fail CI when new PYTHONPATH snippets appear outside the documented allowlist.
4317: """
4318:
4319: from __future__ import annotations
4320:
4321: import sys
4322: from pathlib import Path
4323:
4324: ALLOWLIST = {
4325:     Path("BUILD_HYGIENE.md"),
4326:     Path("tools/validation/validate_build_hygiene.py"),
4327:     Path("docs/QUICKSTART.md"),
4328:     Path("TEST_AUDIT_REPORT.md"),
4329:     Path("tests/conftest.py"),
4330:     Path("STRATEGIC_WIRING_ARCHITECTURE.md"),
4331:     Path("tools/lint/check_pythonpath_references.py"),
4332: }
4333:
4334:
4335: def main() -> int:
4336:     repo_root = Path(__file__).resolve().parents[2]
4337:     violations: list[str] = []
4338:
4339:     for path in repo_root.rglob("*"):
4340:         if path.suffix in {".pyc", ".png", ".jpg", ".jpeg"}:
4341:             continue
4342:         if any(part in {".git", ".venv", "venv", "__pycache__"} for part in path.parts):
4343:             continue
4344:         if not path.is_file():
4345:             continue
4346:
4347:         try:
4348:             text = path.read_text(encoding="utf-8")
4349:         except UnicodeDecodeError:
4350:             continue
4351:
4352:         if "PYTHONPATH" not in text:
4353:             continue
4354:
4355:         rel_path = path.relative_to(repo_root)
4356:
4357:         if rel_path in ALLOWLIST:
4358:             continue
4359:
4360:         for idx, line in enumerate(text.splitlines(), 1):
4361:             if "PYTHONPATH" in line:
```

```
4362:                     violations.append(f"{rel_path}:{idx}: {line.strip()}")
4363:
4364:     if violations:
4365:         print("\u26a1235\u26a1214 New PYTHONPATH references detected outside the allowlist:")
4366:         for violation in violations:
4367:             print(f"  - {violation}")
4368:         print("\nUpdate the documentation to remove these references or extend the allowlist intentionally.")
4369:     return 1
4370:
4371: print("\u26a1234\u26a1205 No unexpected PYTHONPATH references found.")
4372: return 0
4373:
4374:
4375: if __name__ == "__main__":
4376:     raise SystemExit(main())
4377:
4378:
4379:
4380: =====
4381: FILE: tools/orchestration_condition_audit.py
4382: =====
4383:
4384: #!/usr/bin/env python3
4385: """
4386: Advanced orchestration readiness evaluator.
4387:
4388: This script performs SOTA-style detection of objective, necessary, and sufficient
4389: conditions required for orchestrator implementation. It validates runtime
4390: constraints, critical file presence, phase wiring, router guarantees, registry
4391: integrity, and questionnaire resource extraction guarantees, and produces a JSON
4392: report consumable by runbooks or CI instrumentation.
4393: """
4394:
4395: from __future__ import annotations
4396:
4397: import inspect
4398: import json
4399: import sys
4400: import threading
4401: from pathlib import Path
4402: from typing import Any
4403:
4404: try:
4405:     from farfan_pipeline.config.paths import PROJECT_ROOT
4406: except ImportError as exc: # pragma: no cover - configuration error
4407:     raise SystemExit(
4408:         "Unable to import 'farfan_pipeline'. Install the package with 'pip install -e .' before running this audit."
4409:     ) from exc
4410:
4411: REPO_ROOT = PROJECT_ROOT
4412: SRC_ROOT = PROJECT_ROOT / "src"
4413:
4414:
4415: def _record(name: str, passed: bool, severity: str, details: dict[str, Any]) -> dict[str, Any]:
4416:     return {
4417:         "check": name,
```

```
4418:         "passed": passed,
4419:         "severity": severity,
4420:         "details": details,
4421:     }
4422:
4423:
4424: def check_python_version() -> dict[str, Any]:
4425:     required_min = (3, 12)
4426:     required_max = (3, 13)
4427:     actual = sys.version_info[:2]
4428:     passed = required_min[0] <= actual[0] < required_max[0]
4429:     return _record(
4430:         "python_version_window",
4431:         passed,
4432:         "critical",
4433:         {
4434:             "required_range": f"{required_min[0]}.{required_min[1]} <= version < {required_max[0]}.{required_max[1]}",
4435:             "detected": f"{actual[0]}.{actual[1]}",
4436:         },
4437:     )
4438:
4439:
4440: def check_critical_files() -> dict[str, Any]:
4441:     critical_paths = [
4442:         "src/farfan_pipeline/core/orchestrator/core.py",
4443:         "src/farfan_pipeline/core/orchestrator/arg_router.py",
4444:         "src/farfan_pipeline/core/orchestrator/class_registry.py",
4445:         "src/farfan_pipeline/core/orchestrator/executors.py",
4446:         "src/farfan_pipeline/core/orchestrator/factory.py",
4447:         "src/farfan_pipeline/core/orchestrator/questionnaire_resource_provider.py",
4448:         "src/farfan_pipeline/core/orchestrator/questionnaire.py",
4449:         "src/farfan_pipeline/processing/cpp_ingestion/__init__.py",
4450:         "src/farfan_pipeline/processing/cpp_ingestion/models.py",
4451:     ]
4452:     missing = [p for p in critical_paths if not (REPO_ROOT / p).exists()]
4453:     return _record(
4454:         "critical_orchestration_files_present",
4455:         not missing,
4456:         "critical",
4457:         {"missing": missing, "checked": len(critical_paths)},
4458:     )
4459:
4460:
4461: def check_phase_definitions() -> dict[str, Any]:
4462:     try:
4463:         from farfan_pipeline.core.orchestrator.core import Orchestrator
4464:     except Exception as exc: # pragma: no cover - defensive
4465:         return _record(
4466:             "orchestrator_phase_integrity",
4467:             False,
4468:             "critical",
4469:             {"error": f"Unable to import Orchestrator: {exc!r}"},
4470:         )
4471:
4472:     phases = getattr(Orchestrator, "FASES", [])
4473:     ids = [phase[0] for phase in phases]
```

```
4474:     handlers_missing: list[str] = []
4475:     mode_mismatches: list[tuple[str, str]] = []
4476:
4477:     for _, mode, handler_name, _ in phases:
4478:         handler = getattr(Orchestrator, handler_name, None)
4479:         if handler is None:
4480:             handlers_missing.append(handler_name)
4481:             continue
4482:         is_async = inspect.iscoroutinefunction(handler)
4483:         if mode == "async" and not is_async:
4484:             mode_mismatches.append((handler_name, "expected async"))
4485:         if mode == "sync" and is_async:
4486:             mode_mismatches.append((handler_name, "expected sync"))
4487:
4488:     duplicate_ids = len(ids) != len(set(ids))
4489:     monotonic = ids == sorted(ids)
4490:
4491:     passed = bool(phases) and not handlers_missing and not mode_mismatches and not duplicate_ids and monotonic
4492:     return _record(
4493:         "phase_definition_consistency",
4494:         passed,
4495:         "critical",
4496:         {
4497:             "phase_count": len(phases),
4498:             "missing_handlers": handlers_missing,
4499:             "mode_mismatches": mode_mismatches,
4500:             "duplicate_ids": duplicate_ids,
4501:             "monotonic_ids": monotonic,
4502:         },
4503:     )
4504:
4505:
4506: def check_class_registry() -> dict[str, Any]:
4507:     try:
4508:         from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
4509:     except Exception as exc: # pragma: no cover - defensive
4510:         return _record(
4511:             "class_registry_build",
4512:             False,
4513:             "critical",
4514:             {"error": f"Unable to import class registry: {exc!r}"},
4515:         )
4516:
4517:     try:
4518:         registry = build_class_registry()
4519:         registry_count = len(registry)
4520:         sample = sorted(registry.keys())[:5]
4521:         passed = registry_count >= 20
4522:         return _record(
4523:             "class_registry_build",
4524:             passed,
4525:             "high",
4526:             {"count": registry_count, "sample": sample},
4527:         )
4528:     except Exception as exc: # pragma: no cover - defensive
4529:         return _record(
```

```
4530:         "class_registry_build",
4531:         False,
4532:         "critical",
4533:         {"error": f"build_class_registry failed: {exc!r}"},
4534:     )
4535:
4536:
4537: def check_extended_router() -> dict[str, Any]:
4538:     try:
4539:         from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
4540:         from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
4541:     except Exception as exc: # pragma: no cover - defensive
4542:         return _record(
4543:             "extended_router_integrity",
4544:             False,
4545:             "high",
4546:             {"error": f"Unable to import router dependencies: {exc!r}"},
4547:         )
4548:
4549:     try:
4550:         router = ExtendedArgRouter(build_class_registry())
4551:     except Exception as exc:
4552:         return _record(
4553:             "extended_router_integrity",
4554:             False,
4555:             "high",
4556:             {"error": f"ExtendedArgRouter initialization failed: {exc!r}"},
4557:         )
4558:
4559:     lock_ok = isinstance(getattr(router, "_lock", None), threading.RLock)
4560:     special_routes = getattr(router, "_special_routes", {})
4561:     route_count = len(special_routes)
4562:     passed = lock_ok and route_count >= 25
4563:     return _record(
4564:         "extended_router_integrity",
4565:         passed,
4566:         "medium",
4567:         {
4568:             "lock_is_rlock": lock_ok,
4569:             "special_route_count": route_count,
4570:         },
4571:     )
4572:
4573:
4574: def check_questionnaire_provider() -> dict[str, Any]:
4575:     try:
4576:         from farfan_pipeline.core.orchestrator.questionnaire_resource_provider import QuestionnaireResourceProvider
4577:     except Exception as exc: # pragma: no cover - defensive
4578:         return _record(
4579:             "questionnaire_provider_extracts",
4580:             False,
4581:             "medium",
4582:             {"error": f"Unable to import QuestionnaireResourceProvider: {exc!r}"},
4583:         )
4584:
4585:     sample_questionnaire = {
```

```
4586:         "version": "diagnostic",
4587:         "schema_version": "diagnostic",
4588:         "blocks": {
4589:             "micro_questions": [],
4590:             "meso_questions": [],
4591:             "macro_question": {},
4592:         },
4593:         "validations": [],
4594:     }
4595:
4596:     provider = QuestionnaireResourceProvider(sample_questionnaire)
4597:     patterns = provider.extract_all_patterns()
4598:     validations = provider.extract_all_validations()
4599:     passed = isinstance(patterns, list) and isinstance(validations, list)
4600:     return _record(
4601:         "questionnaire_provider_extracts",
4602:         passed,
4603:         "medium",
4604:         {
4605:             "pattern_count": len(patterns),
4606:             "validation_count": len(validations),
4607:         },
4608:     )
4609:
4610:
4611: def check_executor_registry() -> dict[str, Any]:
4612:     try:
4613:         from farfan_pipeline.core.orchestrator.core import Orchestrator
4614:     except Exception as exc: # pragma: no cover - defensive
4615:         return _record(
4616:             "executor_registry_coverage",
4617:             False,
4618:             "high",
4619:             {"error": f"Unable to import Orchestrator: {exc!r}"},
4620:         )
4621:
4622:     executors = getattr(Orchestrator, "executors", {})
4623:     passed = isinstance(executors, dict) and len(executors) >= 25
4624:     return _record(
4625:         "executor_registry_coverage",
4626:         passed,
4627:         "medium",
4628:         {"executor_count": len(executors)},
4629:     )
4630:
4631:
4632: def run_checks() -> dict[str, Any]:
4633:     checks = [
4634:         check_python_version(),
4635:         check_critical_files(),
4636:         check_phase_definitions(),
4637:         check_class_registry(),
4638:         check_extended_router(),
4639:         check_executor_registry(),
4640:         check_questionnaire_provider(),
4641:     ]
```

```
4642:     passed = sum(1 for c in checks if c["passed"])
4643:     failed = len(checks) - passed
4644:     return {
4645:         "total_checks": len(checks),
4646:         "passed": passed,
4647:         "failed": failed,
4648:         "results": checks,
4649:     }
4650:
4651:
4652: def main() -> None:
4653:     report = run_checks()
4654:     print(json.dumps(report, indent=2))
4655:     if report["failed"]:
4656:         sys.exit(1)
4657:
4658:
4659: if __name__ == "__main__":
4660:     main()
4661:
4662:
4663:
4664: =====
4665: FILE: tools/prompt_cross_analysis.py
4666: =====
4667:
4668: """Prompt Cross analytics utilities.
4669:
4670: This module consolidates registry information across micro, meso, and macro
4671: levels and generates cross-cutting diagnostics for coverage, contract health,
4672: and causal path integrity. The calculations use the synthetic dataset stored in
4673: ``data/prompt_cross_registry.json`` to demonstrate how the metrics are derived.
4674: """
4675:
4676: from __future__ import annotations
4677:
4678: import json
4679: from collections import defaultdict
4680: from dataclasses import dataclass
4681: from pathlib import Path
4682: from typing import TYPE_CHECKING
4683:
4684: if TYPE_CHECKING:
4685:     from collections.abc import Iterable
4686:
4687: DATA_PATH = Path("data/prompt_cross_registry.json")
4688:
4689: def _load_data() -> dict[str, object]:
4690:     """Load the consolidated prompt-cross dataset."""
4691:
4692:     with DATA_PATH.open("r", encoding="utf-8") as handle:
4693:         return json.load(handle)
4694:
4695: def _contribution(weight: float, normalized_time: float, depth: int) -> float:
4696:     """Compute contribution score for a single registry entry."""
4697:
```

```
4698:     safe_depth = max(depth, 1)
4699:     return weight * normalized_time / safe_depth
4700:
4701: def consolidate_evidence(records: Iterable[dict[str, object]]) -> dict[str, object]:
4702:     """Deduplicate registry records and compute contribution metrics.
4703:
4704:     Args:
4705:         records: Iterable of QMCM registry records.
4706:
4707:     Returns:
4708:         Dictionary with consolidated nodes, deduplication ratio, and top
4709:         contributors ranked by contribution score.
4710:
4711:
4712:     records = list(records)
4713:     canonical: dict[tuple[str, str, str], dict[str, object]] = {}
4714:     record_to_node: dict[str, str] = {}
4715:     parent_links: dict[str, list[str]] = defaultdict(list)
4716:     contributions: dict[str, float] = defaultdict(float)
4717:
4718:     for record in records:
4719:         key = (
4720:             record["question_id"],
4721:             record["method_id"],
4722:             record["hash_output"],
4723:         )
4724:         node_id = (
4725:             f"{record['level']}|{record['question_id']}|"
4726:             f"{record['method_id']}|{record['hash_output']}"
4727:         )
4728:
4729:         if key not in canonical:
4730:             canonical[key] = {
4731:                 "node_id": node_id,
4732:                 "level": record["level"],
4733:                 "question_id": record["question_id"],
4734:                 "method_id": record["method_id"],
4735:                 "hash_output": record["hash_output"],
4736:                 "dimensions": set(),
4737:                 "records": [],
4738:             }
4739:
4740:         node_entry = canonical[key]
4741:         node_entry["records"].append(record["record_id"])
4742:
4743:         dimension = record.get("dimension")
4744:         if dimension:
4745:             node_entry["dimensions"].add(dimension)
4746:
4747:         record_to_node[record["record_id"]] = node_id
4748:         contributions[node_id] += _contribution(
4749:             record["weight"], record["normalized_time"], int(record["depth"])
4750:         )
4751:
4752:         parent_record = record.get("parent_record")
4753:         if parent_record:
```

```
4754:         parent_links[node_id].append(parent_record)
4755:
4756:     # Resolve parent pointers to canonical node identifiers
4757:     resolved_parents: dict[str, list[str]] = {}
4758:     for node_id, parents in parent_links.items():
4759:         resolved = [
4760:             record_to_node[parent]
4761:             for parent in parents
4762:             if parent in record_to_node
4763:         ]
4764:         resolved_parents[node_id] = sorted(resolved)
4765:
4766:     # Build global node list
4767:     level_order = {"micro": 0, "meso": 1, "macro": 2}
4768:     global_nodes: list[dict[str, object]] = []
4769:     for entry in canonical.values():
4770:         node_id = entry["node_id"]
4771:         global_nodes.append(
4772:             {
4773:                 "node_id": node_id,
4774:                 "level": entry["level"],
4775:                 "question_id": entry["question_id"],
4776:                 "method_id": entry["method_id"],
4777:                 "hash_output": entry["hash_output"],
4778:                 "parent_nodes": resolved_parents.get(node_id, []),
4779:                 "record_count": len(entry["records"]),
4780:                 "dimensions": sorted(entry["dimensions"]),
4781:                 "contribution_score": round(contributions[node_id], 6),
4782:             }
4783:         )
4784:
4785:     global_nodes.sort(
4786:         key=lambda node: (
4787:             level_order.get(str(node["level"]), 99),
4788:             str(node["question_id"]),
4789:             str(node["method_id"]),
4790:         )
4791:     )
4792:
4793:     total_records = len(records)
4794:
4795:     unique_nodes = len(global_nodes)
4796:     dedup_ratio = unique_nodes / total_records if total_records else 0.0
4797:
4798:     top_contributors = sorted(
4799:         (
4800:             {
4801:                 "node_id": node["node_id"],
4802:                 "question_id": node["question_id"],
4803:                 "method_id": node["method_id"],
4804:                 "contribution_score": node["contribution_score"],
4805:             }
4806:             for node in global_nodes
4807:         ),
4808:         key=lambda item: item["contribution_score"],
4809:         reverse=True,
```

```
4810:     )[:5]
4811:
4812:     return {
4813:         "global_nodes": global_nodes,
4814:         "dedup_ratio": round(dedup_ratio, 4),
4815:         "top_contributors": top_contributors,
4816:     }
4817:
4818: def build_method_coverage(entries: Iterable[dict[str, object]]) -> tuple[dict[str, object], str]:
4819:     """Generate method coverage matrix and heatmap recommendations."""
4820:
4821:     dimensions = sorted({entry["dimension"] for entry in entries})
4822:     matrix: dict[str, dict[str, dict[str, float]]] = defaultdict(
4823:         lambda: {"invocations": 0, "tests": 0} for dim in dimensions
4824:     )
4825:
4826:     for entry in entries:
4827:         method = entry["method_id"]
4828:         dim = entry["dimension"]
4829:         matrix[method][dim]["invocations"] += entry["invocations"]
4830:         matrix[method][dim]["tests"] += entry["tests_executed"]
4831:
4832:     recommendations: list[dict[str, object]] = []
4833:     for method, dim_data in matrix.items():
4834:         cold_dims: list[str] = []
4835:         for dim, stats in dim_data.items():
4836:             inv = stats["invocations"]
4837:             tests = stats["tests"]
4838:             coverage_ratio = tests / inv if inv else 0.0
4839:             stats["coverage_ratio"] = round(coverage_ratio, 3)
4840:             if inv and coverage_ratio < 0.25 or not inv:
4841:                 cold_dims.append(dim)
4842:
4843:         if cold_dims:
4844:             recommendations.append(
4845:                 {
4846:                     "method_id": method,
4847:                     "cold_dimensions": cold_dims,
4848:                     "action": "Design targeted regression tests for under-covered dimensions",
4849:                 }
4850:             )
4851:
4852:     # Build ASCII table
4853:     header = ["Method"] + dimensions
4854:     rows: list[list[str]] = []
4855:     for method in sorted(matrix):
4856:         row = [method]
4857:         for dim in dimensions:
4858:             stats = matrix[method][dim]
4859:             if stats["invocations"]:
4860:                 cell = f"{int(stats['invocations'])}/{int(stats['tests'])}"
4861:             else:
4862:                 cell = "0/0"
4863:             row.append(cell)
4864:         rows.append(row)
4865:
```

```
4866:     col_widths = [max(len(row[i]) for row in [header] + rows) for i in range(len(header))]
4867:
4868:     def _format_row(row: list[str]) -> str:
4869:         return " | ".join(val.ljust(col_widths[idx]) for idx, val in enumerate(row))
4870:
4871:     separator = "-+-".join("-" * width for width in col_widths)
4872:     table_lines = [_format_row(header), separator]
4873:     table_lines.extend(_format_row(row) for row in rows)
4874:     ascii_table = "\n".join(table_lines)
4875:
4876:     matrix_serializable = {
4877:         method: {
4878:             dim: {
4879:                 "invocations": stats["invocations"],
4880:                 "tests": stats["tests"],
4881:                 "coverage_ratio": stats.get("coverage_ratio", 0.0),
4882:             }
4883:             for dim, stats in dim_data.items()
4884:         }
4885:         for method, dim_data in matrix.items()
4886:     }
4887:
4888:     return (
4889:         {
4890:             "matrix": matrix_serializable,
4891:             "dimensions": dimensions,
4892:             "recommendations": recommendations,
4893:         },
4894:         ascii_table,
4895:     )
4896:
4897: SEVERITY_WEIGHTS = {
4898:     "critical": 4,
4899:     "high": 3,
4900:     "medium": 2,
4901:     "low": 1,
4902: }
4903:
4904: def analyze_contract_failures(
4905:     entries: Iterable[dict[str, object]], inputs: dict[str, int]
4906: ) -> tuple[dict[str, object], list[str]]:
4907:     """Aggregate contract failures into a funnel and narrative."""
4908:
4909:     level_stats: dict[str, dict[str, object]] = {}
4910:     method_scores: dict[str, dict[str, object]] = defaultdict(
4911:         lambda: {"severity_score": 0, "total_failures": 0}
4912:     )
4913:
4914:     for entry in entries:
4915:         level = entry["level"]
4916:         severity = entry["severity"].lower()
4917:         count = entry["count"]
4918:
4919:         stats = level_stats.setdefault(
4920:             level,
4921:             {"by_severity": defaultdict(int), "total_failures": 0},
```

```

4922:         )
4923:     stats["by_severity"][severity] += count
4924:     stats["total_failures"] += count
4925:
4926:     method_key = f"{entry['method_id']}:{entry['question_id']}"
4927:     method_scores[method_key]["severity_score"] += SEVERITY_WEIGHTS.get(severity, 0) * count
4928:     method_scores[method_key]["total_failures"] += count
4929:
4930:     for level, stats in level_stats.items():
4931:         entries_prev = inputs.get(level, 0)
4932:         drop_pct = stats["total_failures"] / entries_prev if entries_prev else 0.0
4933:         stats["funnel_drop_pct"] = round(drop_pct * 100, 2)
4934:         stats["by_severity"] = dict(stats["by_severity"])
4935:
4936:     top_methods = sorted(
4937:         (
4938:             {
4939:                 "method_id": key.split("::") [0],
4940:                 "context": key.split("::") [1],
4941:                 "severity_score": value["severity_score"],
4942:                 "total_failures": value["total_failures"],
4943:             }
4944:             for key, value in method_scores.items()
4945:         ),
4946:         key=lambda item: (item["severity_score"], item["total_failures"]),
4947:         reverse=True,
4948:     )[:5]
4949:
4950:     narrative = [
4951:         "Micro layer accumulates the largest absolute failures, primarily from the assembler and processor modules.",
4952:         "Meso layer exhibits a sharper proportional drop, signalling propagation of unresolved micro issues into cluster synthesis.",
4953:         "Macro convergence remains fragile with critical severities persisting despite lower volume.",
4954:         "TeoriaCambio validation spikes as a critical blocker along the causal verification chain.",
4955:         "Prioritize regression tests around ReportAssembler.generate_meso_cluster to contain meso escalations.",
4956:     ]
4957:
4958:     return {"funnel": level_stats, "top_methods": top_methods}, narrative
4959:
4960: @dataclass
4961: class PathStatus:
4962:     path_id: str
4963:     complete: bool
4964:     missing_dimensions: list[str]
4965:     issues: list[str]
4966:
4967: def evaluate_causal_paths(data: dict[str, object]) -> dict[str, object]:
4968:     """Verify causal path continuity across dimensions."""
4969:
4970:     expected_sequence: list[str] = data["dimension_sequence"]
4971:     complete_paths: list[dict[str, object]] = []
4972:     broken_paths: list[dict[str, object]] = []
4973:     repair_actions: list[str] = []
4974:
4975:     for path in data["causal_paths"]:
4976:         dims: list[str] = path["dimensions"]
4977:         missing = [dim for dim in expected_sequence if dim not in dims]

```

```
4978:     unexpected = [dim for dim in dims if dim not in expected_sequence]
4979:     issues: list[str] = []
4980:
4981:     if dims != expected_sequence:
4982:         # Check for unexpected dimensions (not in expected sequence)
4983:         if unexpected:
4984:             issues.append(
4985:                 "Unexpected dimensions: " + ", ".join(unexpected))
4986:         )
4987:
4988:         # Check for length mismatch
4989:         if len(dims) != len(expected_sequence):
4990:             issues.append(
4991:                 f"Length mismatch: expected {len(expected_sequence)} dimensions but found {len(dims)}")
4992:         )
4993:
4994:         # Check for adjacency breaks
4995:         for idx, expected_dim in enumerate(expected_sequence):
4996:             if idx >= len(dims):
4997:                 break
4998:             if dims[idx] != expected_dim:
4999:                 issues.append(
5000:                     f"Expected {expected_dim} at position {idx + 1} but found {dims[idx]}")
5001:         )
5002:
5003:         if missing:
5004:             issues.append(
5005:                 "Missing dimensions: " + ", ".join(sorted(missing)))
5006:         )
5007:
5008:         status = PathStatus(
5009:             path_id=path["path_id"],
5010:             complete=not issues and not missing,
5011:             missing_dimensions=missing,
5012:             issues=issues,
5013:         )
5014:
5015:         if status.complete:
5016:             complete_paths.append(
5017:                 {
5018:                     "path_id": status.path_id,
5019:                     "sequence": path["sequence"],
5020:                 }
5021:             )
5022:         else:
5023:             broken_paths.append(
5024:                 {
5025:                     "path_id": status.path_id,
5026:                     "missing_dimensions": status.missing_dimensions,
5027:                     "issues": status.issues,
5028:                 }
5029:             )
5030:             for dim in status.missing_dimensions:
5031:                 repair_actions.append(
5032:                     f"Re-evaluate {dim} in {status.path_id} to restore sequential continuity"
5033:                 )
```

```
5034:         for dim in unexpected:
5035:             repair_actions.append(
5036:                 f"Remove unexpected dimension {dim} from {status.path_id}"
5037:             )
5038:
5039:     return {
5040:         "complete_paths": complete_paths,
5041:         "broken_paths": broken_paths,
5042:         "repair_actions": sorted(set(repair_actions)),
5043:     }
5044:
5045: def run() -> None:
5046:     """Execute all Prompt Cross analyses and print results."""
5047:
5048:     data = _load_data()
5049:
5050:     evidence = consolidate_evidence(data["qmcm_records"])
5051:     print("==== Prompt Cross \u200\223 Evidence Registry Consolidation ===")
5052:     print(json.dumps(evidence, indent=2))
5053:
5054:     heatmap_json, ascii_table = build_method_coverage(data["method_coverage"])
5055:     print("\n==== Prompt Cross \u200\223 Method Coverage Heatmap ===")
5056:     print(json.dumps(heatmap_json, indent=2))
5057:     print("\n" + ascii_table)
5058:
5059:     funnel_json, narrative = analyze_contract_failures(
5060:         data["contract_failures"], data["funnel_inputs"]
5061:     )
5062:     print("\n==== Prompt Cross \u200\223 Contract Failure Funnel ===")
5063:     print(json.dumps(funnel_json, indent=2))
5064:     print("\nNarrativa:")
5065:     for line in narrative:
5066:         print(f"- {line}")
5067:
5068:     causal = evaluate_causal_paths(data)
5069:     print("\n==== Prompt Cross \u200\223 Causal Path Integrity ===")
5070:     print(json.dumps(causal, indent=2))
5071:
5072: if __name__ == "__main__":
5073:     run()
5074:
5075:
```