

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 27
3: =====
4: Generated: 2025-12-07T06:17:35.012359
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: tests/phases/test_adapter.py
11: =====
12:
13: """Test Phase1â\206\222Phase2 Adapter Contract
14:
15: Tests PAÃ\227DIM metadata preservation, sentence_metadata.extra structure.
16: """
17: import pytest
18: from unittest.mock import MagicMock, patch
19: import hashlib
20:
21: from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract
22:
23:
24: class TestAdapterContract:
25:     """Test adapter contract validation."""
26:
27:     def test_adapter_preserves_pa_dim_metadata(self):
28:         """Test adapter preserves policy_area_id and dimension_id in sentence_metadata.extra."""
29:         from farfan_pipeline.processing.models import (
30:             CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
31:             TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest
32:         )
33:
34:         contract = AdapterContract()
35:
36:         chunk_graph = ChunkGraph()
37:         chunk = Chunk(
38:             id="test_chunk", text="Test chunk text",
39:             text_span=TextSpan(0, 50), resolution=ChunkResolution.MESO,
40:             bytes_hash=hashlib.blake2b(b"test").hexdigest(),
41:             policy_area_id="PA01", dimension_id="DIM01"
42:         )
43:         chunk_graph.add_chunk(chunk)
44:
45:         cpp = CanonPolicyPackage(
46:             schema_version="SPC-2025.1", chunk_graph=chunk_graph,
47:             policy_manifest=PolicyManifest(),
48:             quality_metrics=QualityMetrics(provenance_completeness=0.9, structural_consistency=0.9),
49:             integrity_index=IntegrityIndex(blake2b_root="a"*64),
50:             metadata={"document_id": "test"}
51:         )
52:
53:         result = contract.validate_input(cpp)
54:         assert result.passed
55:
56:     def test_adapter_output_has_chunked_mode(self):
```

```
57:         """Test adapter output has processing_mode='chunked'."""
58:         contract = AdapterContract()
59:
60:         invariant = next(inv for inv in contract.invariants if inv.name == "processing_mode_chunked")
61:         assert invariant is not None
62:         assert "chunked" in invariant.description
63:
64:     def test_adapter_requires_chunk_id_in_extra(self):
65:         """Test adapter requires chunk_id in sentence_metadata.extra."""
66:         contract = AdapterContract()
67:
68:         invariant = next(inv for inv in contract.invariants if inv.name == "chunk_id_preserved")
69:         assert invariant is not None
70:         assert "chunk_id" in invariant.description
71:
72:     def test_adapter_requires_policy_area_id_in_extra(self):
73:         """Test adapter requires policy_area_id in sentence_metadata.extra."""
74:         contract = AdapterContract()
75:
76:         invariant = next(inv for inv in contract.invariants if inv.name == "policy_area_id_preserved")
77:         assert invariant is not None
78:         assert "policy_area_id" in invariant.description
79:
80:     def test_adapter_requires_dimension_id_in_extra(self):
81:         """Test adapter requires dimension_id in sentence_metadata.extra."""
82:         contract = AdapterContract()
83:
84:         invariant = next(inv for inv in contract.invariants if inv.name == "dimension_id_preserved")
85:         assert invariant is not None
86:         assert "dimension_id" in invariant.description
87:
88:
89:
90: =====
91: FILE: tests/phases/test_failure_propagation.py
92: =====
93:
94: """Test Failure Propagation (Phase N Failure â\206\222 ABORT)
95:
96: Tests that phase failures propagate correctly and halt pipeline execution.
97: """
98: import pytest
99: from unittest.mock import AsyncMock, MagicMock, patch
100: from datetime import datetime, timezone
101:
102: from farfan_pipeline.core.phases.phase_protocol import PhaseContract, ContractValidationResult
103:
104:
105: class TestFailurePropagation:
106:     """Test phase failure propagation."""
107:
108:     @pytest.mark.asyncio
109:     async def test_input_validation_failure_aborts(self):
110:         """Test input validation failure aborts phase execution."""
111:
112:         class TestContract(PhaseContract):
```

```
113:         def validate_input(self, data):
114:             return ContractValidationResult(
115:                 passed=False, contract_type="input", phase_name="test",
116:                 errors=["Input validation failed"]
117:             )
118:
119:         def validate_output(self, data):
120:             return ContractValidationResult(True, "output", "test")
121:
122:         async def execute(self, data):
123:             return data
124:
125:     contract = TestContract("test")
126:
127:     with pytest.raises(ValueError) as exc_info:
128:         await contract.run("invalid_input")
129:
130:     assert "Input contract validation failed" in str(exc_info.value)
131:
132: @pytest.mark.asyncio
133: async def test_output_validation_failure_aborts(self):
134:     """Test output validation failure aborts phase execution."""
135:
136:     class TestContract(PhaseContract):
137:         def validate_input(self, data):
138:             return ContractValidationResult(True, "input", "test")
139:
140:         def validate_output(self, data):
141:             return ContractValidationResult(
142:                 passed=False, contract_type="output", phase_name="test",
143:                 errors=["Output validation failed"]
144:             )
145:
146:         async def execute(self, data):
147:             return "invalid_output"
148:
149:     contract = TestContract("test")
150:
151:     with pytest.raises(ValueError) as exc_info:
152:         await contract.run("input")
153:
154:     assert "Output contract validation failed" in str(exc_info.value)
155:
156: @pytest.mark.asyncio
157: async def test_invariant_failure_aborts(self):
158:     """Test invariant failure aborts phase execution."""
159:
160:     class TestContract(PhaseContract):
161:         def __init__(self):
162:             super().__init__("test")
163:             self.add_invariant(
164:                 "test_invariant", "Test invariant",
165:                 lambda data: False, "Invariant failed"
166:             )
167:
168:         def validate_input(self, data):
```

```
169:         return ContractValidationResult(True, "input", "test")
170:
171:     def validate_output(self, data):
172:         return ContractValidationResult(True, "output", "test")
173:
174:     async def execute(self, data):
175:         return data
176:
177:     contract = TestContract()
178:
179:     with pytest.raises(RuntimeError) as exc_info:
180:         await contract.run("input")
181:
182:     assert "Phase invariants failed" in str(exc_info.value)
183:
184: @pytest.mark.asyncio
185: async def test_execution_error_captured_in_metadata(self):
186:     """Test execution errors are captured in phase metadata."""
187:
188:     class TestContract(PhaseContract):
189:         def validate_input(self, data):
190:             return ContractValidationResult(True, "input", "test")
191:
192:         def validate_output(self, data):
193:             return ContractValidationResult(True, "output", "test")
194:
195:         async def execute(self, data):
196:             raise RuntimeError("Execution failed")
197:
198:     contract = TestContract("test")
199:
200:     with pytest.raises(RuntimeError):
201:         await contract.run("input")
202:
203:     assert contract.metadata is not None
204:     assert contract.metadata.success is False
205:     assert "Execution failed" in contract.metadata.error
206:
207: @pytest.mark.asyncio
208: async def test_phase_failure_prevents_subsequent_phases(self):
209:     """Test phase failure prevents subsequent phases from executing."""
210:     from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
211:     from pathlib import Path
212:
213:     orchestrator = PhaseOrchestrator()
214:
215:     with patch.object(orchestrator.phase0, 'run', side_effect=ValueError("Phase 0 failed")):
216:         result = await orchestrator.run_pipeline(
217:             pdf_path=Path("nonexistent.pdf"),
218:             run_id="test_run",
219:             questionnaire_path=None
220:         )
221:
222:         assert result.success is False
223:         assert len(result.errors) > 0
224:         assert result.phases_completed == 0
```

```
225:
226:
227: class TestPipelineAbortBehavior:
228:     """Test pipeline abort behavior on failure."""
229:
230:     @pytest.mark.asyncio
231:     async def test_phase0_failure_aborts_pipeline(self):
232:         """Test Phase 0 failure aborts entire pipeline."""
233:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
234:         from pathlib import Path
235:
236:         orchestrator = PhaseOrchestrator()
237:
238:         result = await orchestrator.run_pipeline(
239:             pdf_path=Path("nonexistent.pdf"),
240:             run_id="test_run",
241:             questionnaire_path=None
242:         )
243:
244:         assert result.success is False
245:         assert result.phases_completed == 0
246:         assert result.canonical_input is None
247:
248:     @pytest.mark.asyncio
249:     async def test_manifest_records_failure_point(self):
250:         """Test manifest records which phase failed."""
251:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
252:         from pathlib import Path
253:
254:         orchestrator = PhaseOrchestrator()
255:
256:         result = await orchestrator.run_pipeline(
257:             pdf_path=Path("nonexistent.pdf"),
258:             run_id="test_run",
259:             questionnaire_path=None
260:         )
261:
262:         manifest = result.manifest
263:         assert "phases" in manifest or len(result.errors) > 0
264:
265:
266:
267: =====
268: FILE: tests/phases/test_hash_stability.py
269: =====
270:
271: """Test BLAKE3/BLAKE2b Hash Stability Across Runs
272:
273: Tests deterministic hash generation for chunks and integrity verification.
274: """
275: import hashlib
276: import pytest
277:
278:
279: class TestHashStability:
280:     """Test hash stability and determinism."""
```

```
281:
282:     def test_blake2b_deterministic(self):
283:         """Test BLAKE2b produces same hash for same content."""
284:         content = b"test chunk content for determinism"
285:         hash1 = hashlib.blake2b(content).hexdigest()
286:         hash2 = hashlib.blake2b(content).hexdigest()
287:
288:         assert hash1 == hash2
289:         assert len(hash1) == 128
290:
291:     def test_blake2b_different_content_different_hash(self):
292:         """Test BLAKE2b produces different hashes for different content."""
293:         hash1 = hashlib.blake2b(b"content A").hexdigest()
294:         hash2 = hashlib.blake2b(b"content B").hexdigest()
295:
296:         assert hash1 != hash2
297:
298:     def test_sha256_deterministic(self):
299:         """Test SHA256 produces same hash for same content."""
300:         content = b"test PDF content"
301:         hash1 = hashlib.sha256(content).hexdigest()
302:         hash2 = hashlib.sha256(content).hexdigest()
303:
304:         assert hash1 == hash2
305:         assert len(hash1) == 64
306:
307:     def test_chunk_bytes_hash_stability(self):
308:         """Test Chunk bytes_hash is stable across runs."""
309:         from farfan_pipeline.processing.models import Chunk, ChunkResolution, TextSpan
310:
311:         chunk_text = "Policy text for testing hash stability"
312:         chunk_bytes = chunk_text.encode('utf-8')
313:
314:         hash1 = hashlib.blake2b(chunk_bytes).hexdigest()
315:         hash2 = hashlib.blake2b(chunk_bytes).hexdigest()
316:
317:         chunk1 = Chunk(
318:             id="test1", text=chunk_text,
319:             text_span=TextSpan(0, len(chunk_text)),
320:             resolution=ChunkResolution.MESO,
321:             bytes_hash=hash1
322:         )
323:
324:         chunk2 = Chunk(
325:             id="test2", text=chunk_text,
326:             text_span=TextSpan(0, len(chunk_text)),
327:             resolution=ChunkResolution.MESO,
328:             bytes_hash=hash2
329:         )
330:
331:         assert chunk1.bytes_hash == chunk2.bytes_hash
332:
333:     def test_integrity_index_blake2b_root(self):
334:         """Test IntegrityIndex uses BLAKE2b for root hash."""
335:         from farfan_pipeline.processing.models import IntegrityIndex
336:
```

```
337:         chunk_hashes = {
338:             "chunk1": "a" * 128,
339:             "chunk2": "b" * 128,
340:             "chunk3": "c" * 128
341:         }
342:
343:         combined = "".join(sorted(chunk_hashes.values()))
344:         root_hash = hashlib.blake2b(combined.encode()).hexdigest()
345:
346:         integrity = IntegrityIndex(
347:             blake2b_root=root_hash,
348:             chunk_hashes=chunk_hashes
349:         )
350:
351:         assert len(integrity.blake2b_root) == 128
352:         assert integrity.blake2b_root == root_hash
353:
354:     def test_hash_stability_across_multiple_runs(self):
355:         """Test hash remains stable across multiple computations."""
356:         content = b"Stable content for multiple hash runs"
357:
358:         hashes = [hashlib.blake2b(content).hexdigest() for _ in range(10)]
359:
360:         assert len(set(hashes)) == 1
361:         assert all(h == hashes[0] for h in hashes)
362:
363:     def test_empty_content_has_valid_hash(self):
364:         """Test empty content produces valid hash."""
365:         hash_empty = hashlib.blake2b(b"").hexdigest()
366:
367:         assert len(hash_empty) == 128
368:         assert isinstance(hash_empty, str)
369:
370:     def test_unicode_content_hash_stability(self):
371:         """Test hash stability with unicode content."""
372:         content = "Política de desarrollo con Æ± y tildes: Æ;Ã@Ã-Ã³Ã°"
373:         bytes1 = content.encode('utf-8')
374:         bytes2 = content.encode('utf-8')
375:
376:         hash1 = hashlib.blake2b(bytes1).hexdigest()
377:         hash2 = hashlib.blake2b(bytes2).hexdigest()
378:
379:         assert hash1 == hash2
380:
381:
382:
383: =====
384: FILE: tests/phases/test_manifest_builder.py
385: =====
386:
387: """Test Phase Manifest Builder
388:
389: Tests manifest completeness, phase recording, and artifact tracking.
390: """
391: import pytest
392: from pathlib import Path
```

```
393: from datetime import datetime, timezone
394:
395: from farfan_pipeline.core.phases.phase_protocol import (
396:     PhaseManifestBuilder, PhaseMetadata, ContractValidationResult,
397:     PhaseArtifact
398: )
399:
400:
401: class TestManifestBuilder:
402:     """Test PhaseManifestBuilder functionality."""
403:
404:     def test_manifest_builder_initialization(self):
405:         """Test manifest builder initializes empty."""
406:         builder = PhaseManifestBuilder()
407:         assert len(builder.phases) == 0
408:
409:     def test_record_phase_success(self):
410:         """Test recording successful phase execution."""
411:         builder = PhaseManifestBuilder()
412:
413:         metadata = PhaseMetadata(
414:             phase_name="test_phase",
415:             started_at="2025-01-01T00:00:00Z",
416:             finished_at="2025-01-01T00:00:01Z",
417:             duration_ms=1000.0,
418:             success=True
419:         )
420:
421:         input_validation = ContractValidationResult(
422:             passed=True, contract_type="input", phase_name="test_phase"
423:         )
424:
425:         output_validation = ContractValidationResult(
426:             passed=True, contract_type="output", phase_name="test_phase"
427:         )
428:
429:         builder.record_phase(
430:             phase_name="test_phase",
431:             metadata=metadata,
432:             input_validation=input_validation,
433:             output_validation=output_validation,
434:             invariants_checked=["inv1", "inv2"],
435:             artifacts=[]
436:         )
437:
438:         assert "test_phase" in builder.phases
439:         assert builder.phases["test_phase"]["status"] == "success"
440:         assert builder.phases["test_phase"]["duration_ms"] == 1000.0
441:
442:     def test_record_phase_failure(self):
443:         """Test recording failed phase execution."""
444:         builder = PhaseManifestBuilder()
445:
446:         metadata = PhaseMetadata(
447:             phase_name="test_phase",
448:             started_at="2025-01-01T00:00:00Z",
```



```
449:         finished_at="2025-01-01T00:00:01Z",
450:         duration_ms=500.0,
451:         success=False,
452:         error="Test error"
453:     )
454:
455:     input_validation = ContractValidationResult(
456:         passed=False, contract_type="input", phase_name="test_phase",
457:         errors=["Input validation failed"]
458:     )
459:
460:     output_validation = ContractValidationResult(
461:         passed=False, contract_type="output", phase_name="test_phase"
462:     )
463:
464:     builder.record_phase(
465:         phase_name="test_phase",
466:         metadata=metadata,
467:         input_validation=input_validation,
468:         output_validation=output_validation,
469:         invariants_checked=[],
470:         artifacts=[]
471:     )
472:
473:     assert builder.phases["test_phase"]["status"] == "failed"
474:     assert builder.phases["test_phase"]["error"] == "Test error"
475:
476: def test_manifest_to_dict(self):
477:     """Test manifest conversion to dictionary."""
478:     builder = PhaseManifestBuilder()
479:
480:     metadata = PhaseMetadata(
481:         phase_name="phase1", started_at="2025-01-01T00:00:00Z",
482:         finished_at="2025-01-01T00:00:01Z", success=True
483:     )
484:
485:     builder.record_phase(
486:         "phase1", metadata,
487:         ContractValidationResult(True, "input", "phase1"),
488:         ContractValidationResult(True, "output", "phase1"),
489:         [], []
490:     )
491:
492:     manifest_dict = builder.to_dict()
493:     assert "phases" in manifest_dict
494:     assert "total_phases" in manifest_dict
495:     assert "successful_phases" in manifest_dict
496:     assert "failed_phases" in manifest_dict
497:     assert manifest_dict["total_phases"] == 1
498:     assert manifest_dict["successful_phases"] == 1
499:
500: def test_manifest_completeness_all_phases(self):
501:     """Test manifest records all phases (0, 1, adapter, 2)."""
502:     builder = PhaseManifestBuilder()
503:
504:     for phase_name in ["phase0_input_validation", "phase1_spc_ingestion",
```

```
505:         "phase1_to_phase2_adapter", "phase2_microquestions"]:  
506:     metadata = PhaseMetadata(  
507:         phase_name=phase_name,  
508:         started_at="2025-01-01T00:00:00Z",  
509:         finished_at="2025-01-01T00:00:01Z",  
510:         success=True  
511:     )  
512:     builder.record_phase(  
513:         phase_name, metadata,  
514:         ContractValidationResult(True, "input", phase_name),  
515:         ContractValidationResult(True, "output", phase_name),  
516:         [], []  
517:     )  
518:  
519:     manifest_dict = builder.to_dict()  
520:     assert manifest_dict["total_phases"] == 4  
521:     assert all(p in builder.phases for p in [  
522:         "phase0_input_validation", "phase1_spc_ingestion",  
523:         "phase1_to_phase2_adapter", "phase2_microquestions"  
524:     ])  
525:  
526: def test_manifest_tracks_invariants(self):  
527:     """Test manifest tracks which invariants were checked."""  
528:     builder = PhaseManifestBuilder()  
529:  
530:     metadata = PhaseMetadata(  
531:         phase_name="test_phase",  
532:         started_at="2025-01-01T00:00:00Z",  
533:         success=True  
534:     )  
535:  
536:     invariants = ["validation_passed", "hash_format", "count_positive"]  
537:  
538:     builder.record_phase(  
539:         "test_phase", metadata,  
540:         ContractValidationResult(True, "input", "test_phase"),  
541:         ContractValidationResult(True, "output", "test_phase"),  
542:         invariants, []  
543:     )  
544:  
545:     assert builder.phases["test_phase"]["invariants_checked"] == invariants  
546:     assert builder.phases["test_phase"]["invariants_satisfied"] is True  
547:  
548: def test_manifest_save(self, tmp_path):  
549:     """Test manifest can be saved to file."""  
550:     builder = PhaseManifestBuilder()  
551:  
552:     metadata = PhaseMetadata(  
553:         phase_name="test_phase",  
554:         started_at="2025-01-01T00:00:00Z",  
555:         success=True  
556:     )  
557:  
558:     builder.record_phase(  
559:         "test_phase", metadata,  
560:         ContractValidationResult(True, "input", "test_phase"),
```

```
561:         ContractValidationResult(True, "output", "test_phase"),
562:         [], []
563:     )
564:
565:     manifest_path = tmp_path / "manifest.json"
566:     builder.save(manifest_path)
567:
568:     assert manifest_path.exists()
569:     import json
570:     with open(manifest_path) as f:
571:         saved = json.load(f)
572:     assert "phases" in saved
573:     assert "test_phase" in saved["phases"]
574:
575:
576:
577: =====
578: FILE: tests/phases/test_orchestrator_integration.py
579: =====
580:
581: """Test Phase Orchestrator Integration
582:
583: Tests full pipeline execution through orchestrator with all phases.
584: """
585: import pytest
586: from pathlib import Path
587: from unittest.mock import AsyncMock, patch, MagicMock
588:
589:
590: class TestOrchestratorIntegration:
591:     """Test orchestrator executes all phases in sequence."""
592:
593:     def test_orchestrator_has_all_phase_contracts(self):
594:         """Test orchestrator has phase0, phase1, adapter contracts."""
595:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
596:
597:         orchestrator = PhaseOrchestrator()
598:
599:         assert hasattr(orchestrator, 'phase0')
600:         assert hasattr(orchestrator, 'phase1')
601:         assert hasattr(orchestrator, 'adapter')
602:         assert hasattr(orchestrator, 'manifest_builder')
603:
604:     def test_orchestrator_manifest_builder_initialized(self):
605:         """Test orchestrator initializes manifest builder."""
606:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
607:
608:         orchestrator = PhaseOrchestrator()
609:
610:         assert orchestrator.manifest_builder is not None
611:         assert len(orchestrator.manifest_builder.phases) == 0
612:
613:     @pytest.mark.asyncio
614:     async def test_orchestrator_pipeline_result_structure(self):
615:         """Test PipelineResult has required fields."""
616:         from farfan_pipeline.core.phases.phase_orchestrator import PipelineResult
```

```
617:
618:     result = PipelineResult(
619:         success=False,
620:         run_id="test_run",
621:         phases_completed=0,
622:         phases_failed=0,
623:         total_duration_ms=0.0,
624:         errors=[],
625:         manifest={}
626:     )
627:
628:     assert result.success is False
629:     assert result.run_id == "test_run"
630:     assert hasattr(result, 'canonical_input')
631:     assert hasattr(result, 'canon_policy_package')
632:     assert hasattr(result, 'preprocessed_document')
633:     assert hasattr(result, 'phase2_result')
634:
635: @pytest.mark.asyncio
636: async def test_orchestrator_invalid_pdf_returns_error(self):
637:     """Test orchestrator returns error for invalid PDF."""
638:     from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
639:
640:     orchestrator = PhaseOrchestrator()
641:
642:     result = await orchestrator.run_pipeline(
643:         pdf_path=Path("nonexistent.pdf"),
644:         run_id="test_run",
645:         questionnaire_path=None
646:     )
647:
648:     assert result.success is False
649:     assert len(result.errors) > 0
650:     assert result.phases_completed == 0
651:
652: @pytest.mark.asyncio
653: async def test_orchestrator_records_phases_in_manifest(self):
654:     """Test orchestrator records all phases in manifest."""
655:     from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
656:
657:     orchestrator = PhaseOrchestrator()
658:
659:     result = await orchestrator.run_pipeline(
660:         pdf_path=Path("nonexistent.pdf"),
661:         run_id="test_run",
662:         questionnaire_path=None
663:     )
664:
665:     assert 'manifest' in result.__dict__
666:     assert isinstance(result.manifest, dict)
667:
668: def test_orchestrator_single_entry_point(self):
669:     """Test run_pipeline is the only entry point."""
670:     from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
671:     import inspect
672:
```

```
673:         orchestrator = PhaseOrchestrator()
674:
675:         public_methods = [m for m in dir(orchestrator)
676:                           if not m.startswith('_') and callable(getattr(orchestrator, m))]
677:
678:         assert 'run_pipeline' in public_methods
679:
680:
681: class TestPhaseSequenceEnforcement:
682:     """Test phases execute in strict sequence."""
683:
684:     @pytest.mark.asyncio
685:     async def test_phase0_runs_first(self):
686:         """Test Phase 0 runs before Phase 1."""
687:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
688:
689:         orchestrator = PhaseOrchestrator()
690:
691:         with patch.object(orchestrator.phase0, 'run') as mock_phase0:
692:             mock_phase0.side_effect = ValueError("Phase 0 error")
693:
694:             result = await orchestrator.run_pipeline(
695:                 pdf_path=Path("test.pdf"),
696:                 run_id="test_run",
697:                 questionnaire_path=None
698:             )
699:
700:             assert result.phases_completed == 0
701:             assert result.canon_policy_package is None
702:
703:     def test_phases_cannot_be_called_directly(self):
704:         """Test phase contracts are private (no direct external calls)."""
705:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
706:
707:         orchestrator = PhaseOrchestrator()
708:
709:         assert hasattr(orchestrator, 'phase0')
710:         assert hasattr(orchestrator, 'phase1')
711:         assert hasattr(orchestrator, 'adapter')
712:
713:
714: class TestManifestGeneration:
715:     """Test manifest generation during pipeline execution."""
716:
717:     @pytest.mark.asyncio
718:     async def test_manifest_always_generated(self):
719:         """Test manifest is generated even on failure."""
720:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
721:
722:         orchestrator = PhaseOrchestrator()
723:
724:         result = await orchestrator.run_pipeline(
725:             pdf_path=Path("nonexistent.pdf"),
726:             run_id="test_run",
727:             questionnaire_path=None
728:         )
```

```
729:
730:     assert result.manifest is not None
731:     assert isinstance(result.manifest, dict)
732:
733:     @pytest.mark.asyncio
734:     async def test_manifest_saved_to_artifacts_dir(self, tmp_path):
735:         """Test manifest is saved when artifacts_dir provided."""
736:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
737:
738:         orchestrator = PhaseOrchestrator()
739:         artifacts_dir = tmp_path / "artifacts"
740:
741:         result = await orchestrator.run_pipeline(
742:             pdf_path=Path("nonexistent.pdf"),
743:             run_id="test_run",
744:             questionnaire_path=None,
745:             artifacts_dir=artifacts_dir
746:         )
747:
748:         manifest_path = artifacts_dir / "phase_manifest.json"
749:         assert artifacts_dir.exists()
750:
751:
752:
753: =====
754: FILE: tests/phases/test_phase0_input_validation.py
755: =====
756:
757: """Test Phase 0: Input Validation Contract
758:
759: Tests:
760: - Input contract validation (Phase0Input)
761: - Output contract validation (CanonicalInput)
762: - PDF existence and readability
763: - SHA256 hash determinism (same file â\206\222 same hash)
764: - Questionnaire path resolution
765: - Metadata extraction (page count, file size)
766: - Invariant checking
767: - Error propagation for missing files
768: """
769:
770: import hashlib
771: import tempfile
772: from pathlib import Path
773: from unittest.mock import MagicMock, patch
774:
775: import pytest
776:
777: from farfan_pipeline.core.phases.phase0_input_validation import (
778:     CanonicalInput,
779:     Phase0Input,
780:     Phase0ValidationContract,
781:     PHASE0_VERSION,
782: )
783:
784:
```

```
785: class TestPhase0InputContract:
786:     """Test Phase 0 input contract validation."""
787:
788:     def test_phase0_input_valid(self):
789:         """Test valid Phase0Input passes validation."""
790:         contract = Phase0ValidationContract()
791:         input_data = Phase0Input(
792:             pdf_path=Path("test.pdf"),
793:             run_id="test_run_001",
794:             questionnaire_path=Path("questions.json"),
795:         )
796:
797:         result = contract.validate_input(input_data)
798:         assert result.passed
799:         assert len(result.errors) == 0
800:
801:     def test_phase0_input_invalid_type(self):
802:         """Test invalid input type fails validation."""
803:         contract = Phase0ValidationContract()
804:         input_data = {"pdf_path": "test.pdf"}
805:
806:         result = contract.validate_input(input_data)
807:         assert not result.passed
808:         assert len(result.errors) > 0
809:         assert "Phase0Input" in result.errors[0]
810:
811:     def test_phase0_input_empty_run_id(self):
812:         """Test empty run_id fails validation."""
813:         contract = Phase0ValidationContract()
814:         input_data = Phase0Input(
815:             pdf_path=Path("test.pdf"),
816:             run_id="",
817:             questionnaire_path=None,
818:         )
819:
820:         result = contract.validate_input(input_data)
821:         assert not result.passed
822:
823:     def test_phase0_input_invalid_run_id_chars(self):
824:         """Test run_id with invalid filesystem chars fails validation."""
825:         contract = Phase0ValidationContract()
826:         input_data = Phase0Input(
827:             pdf_path=Path("test.pdf"),
828:             run_id="test/run:001",
829:             questionnaire_path=None,
830:         )
831:
832:         result = contract.validate_input(input_data)
833:         assert not result.passed
834:
835:
836: class TestPhase0OutputContract:
837:     """Test Phase 0 output contract validation."""
838:
839:     def test_canonical_input_valid(self, tmp_path):
840:         """Test valid CanonicalInput passes validation."""
```

```
841:         from datetime import datetime, timezone
842:
843:         contract = Phase0ValidationContract()
844:         pdf_path = tmp_path / "test.pdf"
845:         pdf_path.write_bytes(b"PDF content")
846:
847:         output_data = CanonicalInput(
848:             document_id="test_doc",
849:             run_id="test_run_001",
850:             pdf_path=pdf_path,
851:             pdf_sha256="a" * 64,
852:             pdf_size_bytes=100,
853:             pdf_page_count=5,
854:             questionnaire_path=tmp_path / "questions.json",
855:             questionnaire_sha256="b" * 64,
856:             created_at=datetime.now(timezone.utc),
857:             phase0_version=PHASE0_VERSION,
858:             validation_passed=True,
859:             validation_errors=[],
860:             validation_warnings=[],
861:         )
862:
863:         result = contract.validate_output(output_data)
864:         assert result.passed
865:         assert len(result.errors) == 0
866:
867:     def test_canonical_input_validation_failed(self, tmp_path):
868:         """Test CanonicalInput with validation_passed=False fails."""
869:         from datetime import datetime, timezone
870:
871:         contract = Phase0ValidationContract()
872:
873:         output_data = CanonicalInput(
874:             document_id="test_doc",
875:             run_id="test_run_001",
876:             pdf_path=tmp_path / "test.pdf",
877:             pdf_sha256="a" * 64,
878:             pdf_size_bytes=100,
879:             pdf_page_count=5,
880:             questionnaire_path=tmp_path / "questions.json",
881:             questionnaire_sha256="b" * 64,
882:             created_at=datetime.now(timezone.utc),
883:             phase0_version=PHASE0_VERSION,
884:             validation_passed=False,
885:             validation_errors=["Some error"],
886:             validation_warnings=[],
887:         )
888:
889:         result = contract.validate_output(output_data)
890:         assert not result.passed
891:         assert any("validation_passed" in err for err in result.errors)
892:
893:     def test_canonical_input_invalid_sha256(self, tmp_path):
894:         """Test CanonicalInput with invalid SHA256 fails."""
895:         from datetime import datetime, timezone
896:
```



```
/Users/recovered/Applications/F.A.R.F.A.N -MECHANISTIC-PIPELINE/code_audit_pdfs/batch_27_combined.txt
```

```

897:         contract = Phase0ValidationContract()
898:
899:         output_data = CanonicalInput(
900:             document_id="test_doc",
901:             run_id="test_run_001",
902:             pdf_path=tmp_path / "test.pdf",
903:             pdf_sha256="invalid_hash",
904:             pdf_size_bytes=100,
905:             pdf_page_count=5,
906:             questionnaire_path=tmp_path / "questions.json",
907:             questionnaire_sha256="b" * 64,
908:             created_at=datetime.now(timezone.utc),
909:             phase0_version=PHASE0_VERSION,
910:             validation_passed=True,
911:             validation_errors=[],
912:             validation_warnings=[],
913:         )
914:
915:         result = contract.validate_output(output_data)
916:         assert not result.passed
917:
918:     def test_canonical_input_zero_page_count(self, tmp_path):
919:         """Test CanonicalInput with zero page_count fails."""
920:         from datetime import datetime, timezone
921:
922:         contract = Phase0ValidationContract()
923:
924:         output_data = CanonicalInput(
925:             document_id="test_doc",
926:             run_id="test_run_001",
927:             pdf_path=tmp_path / "test.pdf",
928:             pdf_sha256="a" * 64,
929:             pdf_size_bytes=100,
930:             pdf_page_count=0,
931:             questionnaire_path=tmp_path / "questions.json",
932:             questionnaire_sha256="b" * 64,
933:             created_at=datetime.now(timezone.utc),
934:             phase0_version=PHASE0_VERSION,
935:             validation_passed=True,
936:             validation_errors=[],
937:             validation_warnings=[],
938:         )
939:
940:         result = contract.validate_output(output_data)
941:         assert not result.passed
942:
943:
944: class TestPhase0Execution:
945:     """Test Phase 0 execution logic."""
946:
947:     @pytest.mark.asyncio
948:     async def test_execute_missing_pdf(self, tmp_path):
949:         """Test execution fails for missing PDF."""
950:         contract = Phase0ValidationContract()
951:         input_data = Phase0Input(
952:             pdf_path=tmp_path / "nonexistent.pdf",

```

```
953:         run_id="test_run_001",
954:         questionnaire_path=tmp_path / "questions.json",
955:     )
956:
957:     with pytest.raises(FileNotFoundError) as exc_info:
958:         await contract.execute(input_data)
959:
960:     assert "PDF not found" in str(exc_info.value)
961:
962: @pytest.mark.asyncio
963: async def test_execute_missing_questionnaire(self, tmp_path):
964:     """Test execution fails for missing questionnaire."""
965:     pdf_path = tmp_path / "test.pdf"
966:     pdf_path.write_bytes(b"%PDF-1.4\n")
967:
968:     contract = Phase0ValidationContract()
969:     input_data = Phase0Input(
970:         pdf_path=pdf_path,
971:         run_id="test_run_001",
972:         questionnaire_path=tmp_path / "nonexistent.json",
973:     )
974:
975:     with pytest.raises(FileNotFoundError) as exc_info:
976:         await contract.execute(input_data)
977:
978:     assert "Questionnaire not found" in str(exc_info.value)
979:
980: @pytest.mark.asyncio
981: async def test_execute_success(self, tmp_path):
982:     """Test successful execution produces valid CanonicalInput."""
983:     pdf_path = tmp_path / "test.pdf"
984:     pdf_content = b"%PDF-1.4\nTest PDF content"
985:     pdf_path.write_bytes(pdf_content)
986:
987:     questionnaire_path = tmp_path / "questions.json"
988:     questionnaire_path.write_text('{ "questions": [] }')
989:
990:     contract = Phase0ValidationContract()
991:
992:     with patch.object(contract, "_get_pdf_page_count", return_value=5):
993:         input_data = Phase0Input(
994:             pdf_path=pdf_path,
995:             run_id="test_run_001",
996:             questionnaire_path=questionnaire_path,
997:         )
998:
999:         canonical_input = await contract.execute(input_data)
1000:
1001:         assert canonical_input.document_id == "test"
1002:         assert canonical_input.run_id == "test_run_001"
1003:         assert canonical_input.pdf_path == pdf_path
1004:         assert len(canonical_input.pdf_sha256) == 64
1005:         assert canonical_input.pdf_size_bytes == len(pdf_content)
1006:         assert canonical_input.pdf_page_count == 5
1007:         assert canonical_input.validation_passed is True
1008:         assert len(canonical_input.validation_errors) == 0
```

```
1009:
1010:
1011: class TestPhase0SHA256Determinism:
1012:     """Test SHA256 hash stability across runs."""
1013:
1014:     def test_sha256_same_file_same_hash(self, tmp_path):
1015:         """Test same file produces same SHA256 hash."""
1016:         pdf_path = tmp_path / "test.pdf"
1017:         pdf_content = b"%PDF-1.4\nTest content for determinism"
1018:         pdf_path.write_bytes(pdf_content)
1019:
1020:         contract = Phase0ValidationContract()
1021:
1022:         hash1 = contract._compute_sha256(pdf_path)
1023:         hash2 = contract._compute_sha256(pdf_path)
1024:
1025:         assert hash1 == hash2
1026:         assert len(hash1) == 64
1027:         assert all(c in "0123456789abcdef" for c in hash1)
1028:
1029:     def test_sha256_different_content_different_hash(self, tmp_path):
1030:         """Test different content produces different SHA256 hash."""
1031:         pdf1 = tmp_path / "test1.pdf"
1032:         pdf2 = tmp_path / "test2.pdf"
1033:         pdf1.write_bytes(b"%PDF-1.4\nContent A")
1034:         pdf2.write_bytes(b"%PDF-1.4\nContent B")
1035:
1036:         contract = Phase0ValidationContract()
1037:
1038:         hash1 = contract._compute_sha256(pdf1)
1039:         hash2 = contract._compute_sha256(pdf2)
1040:
1041:         assert hash1 != hash2
1042:
1043:     def test_sha256_matches_expected(self, tmp_path):
1044:         """Test SHA256 matches expected value for known content."""
1045:         pdf_path = tmp_path / "test.pdf"
1046:         pdf_content = b"test"
1047:         pdf_path.write_bytes(pdf_content)
1048:
1049:         contract = Phase0ValidationContract()
1050:         computed_hash = contract._compute_sha256(pdf_path)
1051:
1052:         expected_hash = hashlib.sha256(pdf_content).hexdigest()
1053:         assert computed_hash == expected_hash
1054:
1055:
1056: class TestPhase0Invariants:
1057:     """Test Phase 0 invariants."""
1058:
1059:     def test_all_invariants_registered(self):
1060:         """Test all required invariants are registered."""
1061:         contract = Phase0ValidationContract()
1062:
1063:         invariant_names = [inv.name for inv in contract.invariants]
1064:
```

```
1065:     assert "validation_passed" in invariant_names
1066:     assert "pdf_page_count_positive" in invariant_names
1067:     assert "pdf_size_positive" in invariant_names
1068:     assert "sha256_format" in invariant_names
1069:     assert "no_validation_errors" in invariant_names
1070:
1071: def test_invariants_pass_for_valid_output(self, tmp_path):
1072:     """Test invariants pass for valid output."""
1073:     from datetime import datetime, timezone
1074:
1075:     contract = Phase0ValidationContract()
1076:
1077:     output_data = CanonicalInput(
1078:         document_id="test_doc",
1079:         run_id="test_run_001",
1080:         pdf_path=tmp_path / "test.pdf",
1081:         pdf_sha256="a" * 64,
1082:         pdf_size_bytes=100,
1083:         pdf_page_count=5,
1084:         questionnaire_path=tmp_path / "questions.json",
1085:         questionnaire_sha256="b" * 64,
1086:         created_at=datetime.now(timezone.utc),
1087:         phase0_version=PHASE0_VERSION,
1088:         validation_passed=True,
1089:         validation_errors=[],
1090:         validation_warnings=[],
1091:     )
1092:
1093:     passed, failed = contract.check_invariants(output_data)
1094:     assert passed
1095:     assert len(failed) == 0
1096:
1097: def test_invariants_fail_for_invalid_output(self, tmp_path):
1098:     """Test invariants fail for invalid output."""
1099:     from datetime import datetime, timezone
1100:
1101:     contract = Phase0ValidationContract()
1102:
1103:     output_data = CanonicalInput(
1104:         document_id="test_doc",
1105:         run_id="test_run_001",
1106:         pdf_path=tmp_path / "test.pdf",
1107:         pdf_sha256="invalid",
1108:         pdf_size_bytes=0,
1109:         pdf_page_count=0,
1110:         questionnaire_path=tmp_path / "questions.json",
1111:         questionnaire_sha256="b" * 64,
1112:         created_at=datetime.now(timezone.utc),
1113:         phase0_version=PHASE0_VERSION,
1114:         validation_passed=False,
1115:         validation_errors=["error"],
1116:         validation_warnings=[],
1117:     )
1118:
1119:     passed, failed = contract.check_invariants(output_data)
1120:     assert not passed
```

```
1121:         assert len(failed) > 0
1122:
1123:
1124: class TestPhase0Run:
1125:     """Test Phase 0 run() integration."""
1126:
1127:     @pytest.mark.asyncio
1128:     async def test_run_validates_input(self, tmp_path):
1129:         """Test run() validates input contract."""
1130:         contract = Phase0ValidationContract()
1131:         invalid_input = "not a Phase0Input"
1132:
1133:         with pytest.raises(ValueError) as exc_info:
1134:             await contract.run(invalid_input)
1135:
1136:         assert "Input contract validation failed" in str(exc_info.value)
1137:
1138:     @pytest.mark.asyncio
1139:     async def test_run_returns_metadata(self, tmp_path):
1140:         """Test run() returns phase metadata."""
1141:         pdf_path = tmp_path / "test.pdf"
1142:         pdf_path.write_bytes(b"%PDF-1.4\n")
1143:         questionnaire_path = tmp_path / "questions.json"
1144:         questionnaire_path.write_text('{}')
1145:
1146:         contract = Phase0ValidationContract()
1147:
1148:         with patch.object(contract, "_get_pdf_page_count", return_value=3):
1149:             input_data = Phase0Input(
1150:                 pdf_path=pdf_path,
1151:                 run_id="test_run",
1152:                 questionnaire_path=questionnaire_path,
1153:             )
1154:
1155:             output, metadata = await contract.run(input_data)
1156:
1157:             assert metadata.phase_name == "phase0_input_validation"
1158:             assert metadata.success is True
1159:             assert metadata.error is None
1160:             assert metadata.duration_ms is not None
1161:             assert metadata.duration_ms >= 0
1162:
1163:
1164:
1165: =====
1166: FILE: tests/phases/test_phase1_spc_ingestion.py
1167: =====
1168:
1169: """Test Phase 1: SPC Ingestion Contract
1170:
1171: Tests 60 chunks, PAÃ227DIM tagging, quality thresholds, hash stability.
1172: """
1173: import hashlib
1174: import pytest
1175: from unittest.mock import AsyncMock, patch
1176: from datetime import datetime, timezone
```

```
1177:
1178: from farfan_pipeline.core.phases.phase0_input_validation import CanonicalInput
1179: from farfan_pipeline.core.phases.phase1_spc_ingestion import (
1180:     Phase1SPCIngestionContract, EXPECTED_CHUNK_COUNT, POLICY_AREAS, DIMENSIONS
1181: )
1182:
1183:
1184: class TestPhase1Contract:
1185:     """Test Phase 1 contract validation."""
1186:
1187:     def test_expected_chunk_count(self):
1188:         """Test expected 60 chunks (10 PA Ã\227 6 DIM)."""
1189:         assert EXPECTED_CHUNK_COUNT == 60
1190:         assert len(POLICY_AREAS) == 10
1191:         assert len(DIMENSIONS) == 6
1192:
1193:     def test_input_validation_requires_canonical_input(self, tmp_path):
1194:         """Test input must be CanonicalInput from Phase 0."""
1195:         contract = Phase1SPCIngestionContract()
1196:         pdf = tmp_path / "test.pdf"
1197:         pdf.write_bytes(b"%PDF-1.4\n")
1198:         q = tmp_path / "q.json"
1199:         q.write_text('{}')
1200:
1201:         valid_input = CanonicalInput(
1202:             document_id="test", run_id="r1", pdf_path=pdf,
1203:             pdf_sha256="a"*64, pdf_size_bytes=100, pdf_page_count=5,
1204:             questionnaire_path=q, questionnaire_sha256="b"*64,
1205:             created_at=datetime.now(timezone.utc), phase0_version="1.0.0",
1206:             validation_passed=True, validation_errors=[], validation_warnings=[]
1207:         )
1208:         result = contract.validate_input(valid_input)
1209:         assert result.passed
1210:
1211:     def test_output_requires_60_chunks(self):
1212:         """Test output must have exactly 60 chunks."""
1213:         from farfan_pipeline.processing.models import (
1214:             CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
1215:             TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest
1216:         )
1217:         contract = Phase1SPCIngestionContract()
1218:
1219:         chunk_graph = ChunkGraph()
1220:         for i, pa in enumerate(POLICY_AREAS):
1221:             for j, dim in enumerate(DIMENSIONS):
1222:                 chunk = Chunk(
1223:                     id=f"c_{pa}_{dim}", text=f"Chunk {i*6+j}",
1224:                     text_span=TextSpan(i*100, i*100+50),
1225:                     resolution=ChunkResolution.MESO,
1226:                     bytes_hash=hashlib.blake2b(f"t{i}{j}".encode()).hexdigest(),
1227:                     policy_area_id=pa, dimension_id=dim
1228:                 )
1229:                 chunk_graph.add_chunk(chunk)
1230:
1231:         cpp = CanonPolicyPackage(
1232:             schema_version="SPC-2025.1", chunk_graph=chunk_graph,
```

```
1233:         policy_manifest=PolicyManifest(),
1234:         quality_metrics=QualityMetrics(provenance_completeness=0.9, structural_consistency=0.9),
1235:         integrity_index=IntegrityIndex(blake2b_root="a"*64),
1236:         metadata={"document_id": "test"}
1237:     )
1238:
1239:     result = contract.validate_output(cpp)
1240:     assert result.passed
1241:
1242:     def test_provenance_threshold_enforced(self):
1243:         """Test provenance_completeness >= 0.8 enforced."""
1244:         from farfan_pipeline.processing.models import (
1245:             CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
1246:             TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest
1247:         )
1248:         contract = Phase1SPCIngestionContract()
1249:
1250:         chunk_graph = ChunkGraph()
1251:         for i, pa in enumerate(POLICY_AREAS):
1252:             for j, dim in enumerate(DIMENSIONS):
1253:                 chunk = Chunk(
1254:                     id=f"c_{pa}_{dim}", text="t",
1255:                     text_span=TextSpan(0, 1), resolution=ChunkResolution.MESO,
1256:                     bytes_hash=hashlib.blake2b(f"t{i}{j}".encode()).hexdigest(),
1257:                     policy_area_id=pa, dimension_id=dim
1258:                 )
1259:                 chunk_graph.add_chunk(chunk)
1260:
1261:         cpp = CanonPolicyPackage(
1262:             schema_version="SPC-2025.1", chunk_graph=chunk_graph,
1263:             policy_manifest=PolicyManifest(),
1264:             quality_metrics=QualityMetrics(provenance_completeness=0.5, structural_consistency=0.9),
1265:             integrity_index=IntegrityIndex(blake2b_root="a"*64),
1266:             metadata={"document_id": "test"}
1267:         )
1268:
1269:         result = contract.validate_output(cpp)
1270:         assert not result.passed
1271:
1272:
1273: class TestPhase1HashStability:
1274:     """Test BLAKE2b hash stability."""
1275:
1276:     def test_blake2b_deterministic(self):
1277:         """Test BLAKE2b produces same hash for same content."""
1278:         content = b"test chunk content"
1279:         hash1 = hashlib.blake2b(content).hexdigest()
1280:         hash2 = hashlib.blake2b(content).hexdigest()
1281:         assert hash1 == hash2
1282:         assert len(hash1) == 128
1283:
1284:
1285:
1286: =====
1287: FILE: tests/phases/test_phase2_executor_context_preparation.py
1288: =====
```

```
1289:
1290: """
1291: Test Phase 2: Executor Context Preparation Integration Tests
1292:
1293: Tests validation of routing key extraction, execution metadata extraction,
1294: logging output structure, and deterministic task ordering for Phase 2.
1295:
1296: CRITICAL VALIDATION AREAS:
1297: 1. Routing key extraction (pa_id, dim_id, question_global, question_id)
1298: 2. Execution metadata (expected_elements, signal_requirements, patterns)
1299: 3. Logging output structure and traceability
1300: 4. Deterministic task ordering across multiple runs
1301: """
1302:
1303: import json
1304: from dataclasses import asdict
1305: from typing import Any
1306: from unittest.mock import patch
1307:
1308: import pytest
1309:
1310: from farfan_pipeline.core.orchestrator.task_planner import (
1311:     ExecutableTask,
1312:     _construct_task_legacy,
1313:     _validate_cross_task,
1314:     _validate_schema,
1315: )
1316:
1317: TOTAL_QUESTIONS = 300
1318: TOTAL_DIMENSIONS = 6
1319: TOTAL_POLICY_AREAS = 10
1320: QUESTIONS_PER_DIMENSION = 50
1321: POLICY_AREA_ID_LENGTH = 4
1322: DIMENSION_ID_LENGTH = 5
1323: PA_DIM_COMBINATIONS = 60
1324:
1325:
1326: @pytest.fixture
1327: def sample_question() -> dict[str, Any]:
1328:     """Sample microquestion for Phase 2."""
1329:     return {
1330:         "question_id": "MICRO_001",
1331:         "question_global": 1,
1332:         "base_slot": "D1-Q1",
1333:         "dimension_id": "DIM01",
1334:         "policy_area_id": "PA01",
1335:         "cluster_id": "C01",
1336:         "text": "Sample question text",
1337:         "expected_elements": [
1338:             {"field": "evidence", "type": "list"},
1339:             {"field": "confidence", "type": "float"},
1340:         ],
1341:     }
1342:
1343:
1344: @pytest.fixture
```



```
1345: def sample_chunk() -> dict[str, Any]:
1346:     """Sample chunk for Phase 2."""
1347:     return {
1348:         "id": "CHUNK_001",
1349:         "policy_area_id": "PA01",
1350:         "dimension_id": "DIM01",
1351:         "expected_elements": [
1352:             {"field": "evidence", "type": "list"},
1353:             {"field": "confidence", "type": "float"},
1354:         ],
1355:     }
1356:
1357:
1358: @pytest.fixture
1359: def sample_patterns() -> list[dict[str, Any]]:
1360:     """Sample patterns for signal extraction."""
1361:     return [
1362:         {
1363:             "pattern_id": "PAT_001",
1364:             "pattern_text": "problema.*identificado",
1365:             "pattern_type": "diagnostic",
1366:             "weight": 0.8,
1367:         },
1368:         {
1369:             "pattern_id": "PAT_002",
1370:             "pattern_text": "brecha.*detectada",
1371:             "pattern_type": "gap_analysis",
1372:             "weight": 0.9,
1373:         },
1374:     ]
1375:
1376:
1377: @pytest.fixture
1378: def sample_signals() -> dict[str, Any]:
1379:     """Sample signals for question answering."""
1380:     return {
1381:         "required_signals": ["diagnostic_evidence", "gap_metrics"],
1382:         "thresholds": {"min_confidence": 0.7, "min_evidence_count": 3},
1383:         "signal_weights": {"diagnostic_evidence": 0.6, "gap_metrics": 0.4},
1384:     }
1385:
1386:
1387: class TestRoutingKeyExtraction:
1388:     """Test routing key extraction from questions and chunks."""
1389:
1390:     def test_extract_pa_id_from_question(self, sample_question):
1391:         """Test policy area ID extraction."""
1392:         task = _construct_task(
1393:             question=sample_question,
1394:             chunk={"id": "CHUNK_001", "expected_elements": []},
1395:             patterns=[],
1396:             signals={},
1397:             generated_ids=set(),
1398:         )
1399:
1400:         assert task.policy_area_id == "PA01"
```

```
1401:         assert task.policy_area_id.startswith("PA")
1402:         assert len(task.policy_area_id) == POLICY_AREA_ID_LENGTH
1403:
1404:     def test_extract_dim_id_from_question(self, sample_question):
1405:         """Test dimension ID extraction."""
1406:         task = _construct_task(
1407:             question=sample_question,
1408:             chunk={"id": "CHUNK_001", "expected_elements": []},
1409:             patterns=[],
1410:             signals={},
1411:             generated_ids=set(),
1412:         )
1413:
1414:         assert task.dimension_id == "DIM01"
1415:         assert task.dimension_id.startswith("DIM")
1416:         assert len(task.dimension_id) == DIMENSION_ID_LENGTH
1417:
1418:     def test_extract_question_global_from_question(self, sample_question):
1419:         """Test question_global extraction."""
1420:         task = _construct_task(
1421:             question=sample_question,
1422:             chunk={"id": "CHUNK_001", "expected_elements": []},
1423:             patterns=[],
1424:             signals={},
1425:             generated_ids=set(),
1426:         )
1427:
1428:         assert task.question_global == 1
1429:         assert isinstance(task.question_global, int)
1430:         assert 1 <= task.question_global <= TOTAL_QUESTIONS
1431:
1432:     def test_extract_question_id_from_question(self, sample_question):
1433:         """Test question_id extraction."""
1434:         task = _construct_task(
1435:             question=sample_question,
1436:             chunk={"id": "CHUNK_001", "expected_elements": []},
1437:             patterns=[],
1438:             signals={},
1439:             generated_ids=set(),
1440:         )
1441:
1442:         assert task.question_id == "MICRO_001"
1443:         assert isinstance(task.question_id, str)
1444:
1445:     def test_task_id_format_consistency(self, sample_question):
1446:         """Test task_id follows MQC-{question_global:03d}_{policy_area_id} format."""
1447:         task = _construct_task(
1448:             question=sample_question,
1449:             chunk={"id": "CHUNK_001", "expected_elements": []},
1450:             patterns=[],
1451:             signals={},
1452:             generated_ids=set(),
1453:         )
1454:
1455:         assert task.task_id == "MQC-001_PA01"
1456:         assert task.task_id.startswith("MQC-")
```

```
1457:         assert "_PA" in task.task_id
1458:
1459:     def test_routing_keys_all_dimensions(self):
1460:         """Test routing key extraction for all 6 dimensions."""
1461:         for dim_num in range(1, 7):
1462:             question = {
1463:                 "question_id": f"MICRO_{dim_num:03d}",
1464:                 "question_global": dim_num,
1465:                 "dimension_id": f"DIM0{dim_num}",
1466:                 "policy_area_id": "PA01",
1467:                 "expected_elements": [],
1468:             }
1469:
1470:             task = _construct_task(
1471:                 question=question,
1472:                 chunk={"id": "CHUNK_001", "expected_elements": []},
1473:                 patterns=[],
1474:                 signals={},
1475:                 generated_ids=set(),
1476:             )
1477:
1478:             assert task.dimension_id == f"DIM0{dim_num}"
1479:
1480:     def test_routing_keys_all_policy_areas(self):
1481:         """Test routing key extraction for all 10 policy areas."""
1482:         for pa_num in range(1, 11):
1483:             question = {
1484:                 "question_id": f"MICRO_{pa_num:03d}",
1485:                 "question_global": pa_num,
1486:                 "dimension_id": "DIM01",
1487:                 "policy_area_id": f"PA{pa_num:02d}",
1488:                 "expected_elements": [],
1489:             }
1490:
1491:             task = _construct_task(
1492:                 question=question,
1493:                 chunk={"id": "CHUNK_001", "expected_elements": []},
1494:                 patterns=[],
1495:                 signals={},
1496:                 generated_ids=set(),
1497:             )
1498:
1499:             assert task.policy_area_id == f"PA{pa_num:02d}"
1500:
1501:     def test_chunk_id_extraction(self, sample_question):
1502:         """Test chunk_id extraction from chunk metadata."""
1503:         chunk = {"id": "CHUNK_042", "expected_elements": []}
1504:
1505:         task = _construct_task(
1506:             question=sample_question,
1507:             chunk=chunk,
1508:             patterns=[],
1509:             signals={},
1510:             generated_ids=set(),
1511:         )
1512:
```

```
1513:         assert task.chunk_id == "CHUNK_042"
1514:
1515:
1516: class TestExecutionMetadataExtraction:
1517:     """Test execution metadata extraction for Phase 2 context."""
1518:
1519:     def test_extract_expected_elements(self, sample_question, sample_chunk):
1520:         """Test expected_elements extraction and validation."""
1521:         task = _construct_task(
1522:             question=sample_question,
1523:             chunk=sample_chunk,
1524:             patterns=[],
1525:             signals={},
1526:             generated_ids=set(),
1527:         )
1528:
1529:         assert len(task.expected_elements) == 2
1530:         assert task.expected_elements[0]["field"] == "evidence"
1531:         assert task.expected_elements[1]["field"] == "confidence"
1532:
1533:     def test_schema_validation_success(self, sample_question, sample_chunk):
1534:         """Test schema validation passes when schemas match."""
1535:         _validate_schema(sample_question, sample_chunk)
1536:
1537:     def test_schema_validation_failure(self, sample_question):
1538:         """Test schema validation fails when schemas mismatch."""
1539:         mismatched_chunk = {
1540:             "id": "CHUNK_001",
1541:             "expected_elements": [{"field": "different", "type": "string"}],
1542:         }
1543:
1544:         with pytest.raises(ValueError, match="Schema mismatch"):
1545:             _validate_schema(sample_question, mismatched_chunk)
1546:
1547:     def test_extract_signal_requirements(self, sample_question, sample_signals):
1548:         """Test signal requirements extraction."""
1549:         task = _construct_task(
1550:             question=sample_question,
1551:             chunk={"id": "CHUNK_001", "expected_elements": []},
1552:             patterns=[],
1553:             signals=sample_signals,
1554:             generated_ids=set(),
1555:         )
1556:
1557:         assert "required_signals" in task.signals
1558:         assert "thresholds" in task.signals
1559:         assert task.signals["required_signals"] == [
1560:             "diagnostic_evidence",
1561:             "gap_metrics",
1562:         ]
1563:
1564:     def test_extract_patterns(self, sample_question, sample_patterns):
1565:         """Test pattern extraction for evidence matching."""
1566:         task = _construct_task(
1567:             question=sample_question,
1568:             chunk={"id": "CHUNK_001", "expected_elements": []},
```

```
1569:         patterns=sample_patterns,
1570:         signals={},
1571:         generated_ids=set(),
1572:     )
1573:
1574:     assert len(task.patterns) == 2
1575:     assert task.patterns[0]["pattern_id"] == "PAT_001"
1576:     assert task.patterns[1]["pattern_type"] == "gap_analysis"
1577:
1578:     def test_metadata_cluster_id(self, sample_question):
1579:         """Test cluster_id metadata extraction."""
1580:         task = _construct_task(
1581:             question=sample_question,
1582:             chunk={"id": "CHUNK_001", "expected_elements": []},
1583:             patterns=[],
1584:             signals={},
1585:             generated_ids=set(),
1586:         )
1587:
1588:         assert task.metadata["cluster_id"] == "C01"
1589:
1590:     def test_metadata_base_slot(self, sample_question):
1591:         """Test base_slot metadata extraction."""
1592:         task = _construct_task(
1593:             question=sample_question,
1594:             chunk={"id": "CHUNK_001", "expected_elements": []},
1595:             patterns=[],
1596:             signals={},
1597:             generated_ids=set(),
1598:         )
1599:
1600:         assert task.metadata["base_slot"] == "D1-Q1"
1601:
1602:     def test_creation_timestamp_format(self, sample_question):
1603:         """Test creation_timestamp is in ISO format."""
1604:         task = _construct_task(
1605:             question=sample_question,
1606:             chunk={"id": "CHUNK_001", "expected_elements": []},
1607:             patterns=[],
1608:             signals={},
1609:             generated_ids=set(),
1610:         )
1611:
1612:         assert isinstance(task.creation_timestamp, str)
1613:         assert "T" in task.creation_timestamp
1614:         assert "Z" in task.creation_timestamp or "+" in task.creation_timestamp
1615:
1616:
1617:     class TestLoggingOutputStructure:
1618:         """Test logging output structure and traceability."""
1619:
1620:     def test_task_serialization_to_dict(self, sample_question):
1621:         """Test ExecutableTask can be serialized to dict."""
1622:         task = _construct_task(
1623:             question=sample_question,
1624:             chunk={"id": "CHUNK_001", "expected_elements": []},
```

```
1625:         patterns=[],
1626:         signals={},
1627:         generated_ids=set(),
1628:     )
1629:
1630:     task_dict = asdict(task)
1631:
1632:     assert isinstance(task_dict, dict)
1633:     assert "task_id" in task_dict
1634:     assert "question_id" in task_dict
1635:     assert "policy_area_id" in task_dict
1636:
1637:     def test_task_serialization_to_json(self, sample_question):
1638:         """Test ExecutableTask can be serialized to JSON."""
1639:         task = _construct_task(
1640:             question=sample_question,
1641:             chunk={"id": "CHUNK_001", "expected_elements": []},
1642:             patterns=[],
1643:             signals={},
1644:             generated_ids=set(),
1645:         )
1646:
1647:         task_dict = asdict(task)
1648:         json_str = json.dumps(task_dict)
1649:
1650:         assert isinstance(json_str, str)
1651:         parsed = json.loads(json_str)
1652:         assert parsed["task_id"] == task.task_id
1653:
1654:     def test_logging_contains_all_routing_keys(self, sample_question):
1655:         """Test logging output includes all routing keys."""
1656:         task = _construct_task(
1657:             question=sample_question,
1658:             chunk={"id": "CHUNK_001", "expected_elements": []},
1659:             patterns=[],
1660:             signals={},
1661:             generated_ids=set(),
1662:         )
1663:
1664:         task_dict = asdict(task)
1665:
1666:         assert "policy_area_id" in task_dict
1667:         assert "dimension_id" in task_dict
1668:         assert "question_global" in task_dict
1669:         assert "question_id" in task_dict
1670:
1671:     def test_logging_contains_execution_metadata(self, sample_question, sample_signals):
1672:         """Test logging output includes execution metadata."""
1673:         task = _construct_task(
1674:             question=sample_question,
1675:             chunk={"id": "CHUNK_001", "expected_elements": []},
1676:             patterns=[],
1677:             signals=sample_signals,
1678:             generated_ids=set(),
1679:         )
1680:
```

```
1681:         task_dict = asdict(task)
1682:
1683:         assert "expected_elements" in task_dict
1684:         assert "signals" in task_dict
1685:         assert "patterns" in task_dict
1686:         assert "creation_timestamp" in task_dict
1687:
1688:     def test_logging_metadata_nested_structure(self, sample_question):
1689:         """Test metadata field contains nested structure."""
1690:         task = _construct_task(
1691:             question=sample_question,
1692:             chunk={"id": "CHUNK_001", "expected_elements": []},
1693:             patterns=[],
1694:             signals={},
1695:             generated_ids=set(),
1696:         )
1697:
1698:         assert isinstance(task.metadata, dict)
1699:         assert "base_slot" in task.metadata
1700:         assert "cluster_id" in task.metadata
1701:
1702:     def test_duplicate_task_id_detection(self, sample_question):
1703:         """Test duplicate task_id raises error."""
1704:         generated_ids = set()
1705:
1706:         _construct_task(
1707:             question=sample_question,
1708:             chunk={"id": "CHUNK_001", "expected_elements": []},
1709:             patterns=[],
1710:             signals={},
1711:             generated_ids=generated_ids,
1712:         )
1713:
1714:         with pytest.raises(ValueError, match="Duplicate task_id"):
1715:             _construct_task(
1716:                 question=sample_question,
1717:                 chunk={"id": "CHUNK_002", "expected_elements": []},
1718:                 patterns=[],
1719:                 signals={},
1720:                 generated_ids=generated_ids,
1721:             )
1722:
1723:
1724: class TestDeterministicTaskOrdering:
1725:     """Test deterministic task ordering across multiple runs."""
1726:
1727:     def test_dimension_first_ordering(self):
1728:         """Test tasks are ordered by dimension first."""
1729:         questions = [
1730:             {
1731:                 "question_id": f"Q{i:03d}",
1732:                 "question_global": i,
1733:                 "dimension_id": f"DIM0{(i % 6) + 1}",
1734:                 "policy_area_id": "PA01",
1735:                 "expected_elements": [],
1736:             }
```

```
1737:         for i in range(1, 13)
1738:     ]
1739:
1740:     tasks = [
1741:         _construct_task(
1742:             question=q,
1743:             chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
1744:             patterns=[],
1745:             signals={},
1746:             generated_ids=set(),
1747:         )
1748:         for i, q in enumerate(questions, 1)
1749:     ]
1750:
1751:     sorted_tasks = sorted(tasks, key=lambda t: (t.dimension_id, t.policy_area_id))
1752:
1753:     for i in range(len(sorted_tasks) - 1):
1754:         assert sorted_tasks[i].dimension_id <= sorted_tasks[i + 1].dimension_id
1755:
1756: def test_policy_area_ordering_within_dimension(self):
1757:     """Test tasks are ordered by policy area within each dimension."""
1758:     questions = [
1759:         {
1760:             "question_id": f"Q{i:03d}",
1761:             "question_global": i,
1762:             "dimension_id": "DIM01",
1763:             "policy_area_id": f"PA{(i % 10) + 1:02d}",
1764:             "expected_elements": [],
1765:         }
1766:         for i in range(1, 21)
1767:     ]
1768:
1769:     tasks = [
1770:         _construct_task(
1771:             question=q,
1772:             chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
1773:             patterns=[],
1774:             signals={},
1775:             generated_ids=set(),
1776:         )
1777:         for i, q in enumerate(questions, 1)
1778:     ]
1779:
1780:     dim01_tasks = [t for t in tasks if t.dimension_id == "DIM01"]
1781:     sorted_tasks = sorted(dim01_tasks, key=lambda t: t.policy_area_id)
1782:
1783:     for i in range(len(sorted_tasks) - 1):
1784:         assert sorted_tasks[i].policy_area_id <= sorted_tasks[i + 1].policy_area_id
1785:
1786: def test_ordering_deterministic_across_runs(self):
1787:     """Test task ordering is deterministic across multiple runs."""
1788:     questions = [
1789:         {
1790:             "question_id": f"Q{i:03d}",
1791:             "question_global": i,
1792:             "dimension_id": f"DIM0{((i - 1) // 50) + 1}",
```



```
1793:         "policy_area_id": f"PA{((i - 1) % 10) + 1:02d}",
1794:         "expected_elements": [],
1795:     }
1796:     for i in range(1, 31)
1797: ]
1798:
1799: task_ids_runs = []
1800: for _run in range(5):
1801:     tasks = [
1802:         _construct_task(
1803:             question=q,
1804:             chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
1805:             patterns=[],
1806:             signals={},
1807:             generated_ids=set(),
1808:         )
1809:         for i, q in enumerate(questions, 1)
1810:     ]
1811:
1812:     sorted_tasks = sorted(
1813:         tasks, key=lambda t: (t.dimension_id, t.policy_area_id, t.question_id)
1814:     )
1815:     task_ids = [t.task_id for t in sorted_tasks]
1816:     task_ids_runs.append(task_ids)
1817:
1818: first_run = task_ids_runs[0]
1819: for subsequent_run in task_ids_runs[1:]:
1820:     assert first_run == subsequent_run
1821:
1822: def test_300_questions_ordering(self):
1823:     """Test all 300 questions maintain deterministic ordering."""
1824:     questions = [
1825:         {
1826:             "question_id": f"MICRO_{i:03d}",
1827:             "question_global": i,
1828:             "dimension_id": f"DIM{((i - 1) // 50) + 1:02d}",
1829:             "policy_area_id": f"PA{((i - 1) % 10) + 1:02d}",
1830:             "expected_elements": [],
1831:         }
1832:         for i in range(1, 301)
1833:     ]
1834:
1835:     generated_ids = set()
1836:     tasks = [
1837:         _construct_task(
1838:             question=q,
1839:             chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
1840:             patterns=[],
1841:             signals={},
1842:             generated_ids=generated_ids,
1843:         )
1844:         for i, q in enumerate(questions, 1)
1845:     ]
1846:
1847:     assert len(tasks) == TOTAL_QUESTIONS
1848:
```

```
1849:         sorted_tasks = sorted(
1850:             tasks, key=lambda t: (t.dimension_id, t.policy_area_id, t.question_global)
1851:         )
1852:
1853:         for i in range(len(sorted_tasks) - 1):
1854:             if sorted_tasks[i].dimension_id == sorted_tasks[i + 1].dimension_id:
1855:                 assert (
1856:                     sorted_tasks[i].policy_area_id <= sorted_tasks[i + 1].policy_area_id
1857:                 )
1858:
1859:
1860: class TestCrossTaskValidation:
1861:     """Test cross-task validation and consistency checks."""
1862:
1863:     def test_chunk_usage_validation(self):
1864:         """Test chunk usage validation (5 tasks per chunk expected)."""
1865:         tasks = []
1866:         for i in range(1, 6):
1867:             task = ExecutableTask(
1868:                 task_id=f"MQC-{i:03d}_PA01",
1869:                 question_id=f"Q{i:03d}",
1870:                 question_global=i,
1871:                 policy_area_id="PA01",
1872:                 dimension_id="DIM01",
1873:                 chunk_id="CHUNK_001",
1874:                 patterns=[],
1875:                 signals={},
1876:                 creation_timestamp="2025-01-01T00:00:00Z",
1877:             )
1878:             tasks.append(task)
1879:
1880:         with patch(
1881:             "farfan_pipeline.core.orchestrator.task_planner.logger"
1882:         ) as mock_logger:
1883:             _validate_cross_task(tasks)
1884:
1885:             chunk_warnings = [
1886:                 call
1887:                 for call in mock_logger.warning.call_args_list
1888:                 if "Chunk usage deviation" in str(call)
1889:             ]
1890:             assert len(chunk_warnings) == 0
1891:
1892:     def test_chunk_usage_deviation_warning(self):
1893:         """Test warning when chunk usage deviates from expected."""
1894:         tasks = []
1895:         for i in range(1, 4):
1896:             task = ExecutableTask(
1897:                 task_id=f"MQC-{i:03d}_PA01",
1898:                 question_id=f"Q{i:03d}",
1899:                 question_global=i,
1900:                 policy_area_id="PA01",
1901:                 dimension_id="DIM01",
1902:                 chunk_id="CHUNK_001",
1903:                 patterns=[],
1904:                 signals={},
```

```
1905:             creation_timestamp="2025-01-01T00:00:00Z",
1906:         )
1907:         tasks.append(task)
1908:
1909:     with patch(
1910:         "farfan_pipeline.core.orchestrator.task_planner.logger"
1911:     ) as mock_logger:
1912:         _validate_cross_task(tasks)
1913:
1914:     mock_logger.warning.assert_called()
1915:     chunk_warnings = [
1916:         call[0][0]
1917:         for call in mock_logger.warning.call_args_list
1918:         if "Chunk usage deviation" in call[0][0]
1919:     ]
1920:     assert len(chunk_warnings) > 0
1921:     assert "CHUNK_001" in chunk_warnings[0]
1922:
1923: def test_policy_area_usage_validation(self):
1924:     """Test policy area usage validation (30 tasks per PA expected)."""
1925:     tasks = []
1926:     for i in range(1, 31):
1927:         task = ExecutableTask(
1928:             task_id=f"MQC-{i:03d}_PA01",
1929:             question_id=f"Q{i:03d}",
1930:             question_global=i,
1931:             policy_area_id="PA01",
1932:             dimension_id=f"DIM{((i - 1) // 5) + 1:02d}",
1933:             chunk_id=f"CHUNK_{i:03d}",
1934:             patterns=[],
1935:             signals={},
1936:             creation_timestamp="2025-01-01T00:00:00Z",
1937:         )
1938:         tasks.append(task)
1939:
1940:     with patch(
1941:         "farfan_pipeline.core.orchestrator.task_planner.logger"
1942:     ) as mock_logger:
1943:         _validate_cross_task(tasks)
1944:
1945:     pa_warnings = [
1946:         call
1947:         for call in mock_logger.warning.call_args_list
1948:         if "Policy area usage deviation" in str(call)
1949:     ]
1950:     assert len(pa_warnings) == 0
1951:
1952: def test_policy_area_usage_deviation_warning(self):
1953:     """Test warning when policy area usage deviates from expected."""
1954:     tasks = []
1955:     for i in range(1, 16):
1956:         task = ExecutableTask(
1957:             task_id=f"MQC-{i:03d}_PA01",
1958:             question_id=f"Q{i:03d}",
1959:             question_global=i,
1960:             policy_area_id="PA01",
```

```
1961:         dimension_id="DIM01",
1962:         chunk_id=f"CHUNK_{i:03d}",
1963:         patterns=[],
1964:         signals={},
1965:         creation_timestamp="2025-01-01T00:00:00Z",
1966:     )
1967:     tasks.append(task)
1968:
1969:     with patch(
1970:         "farfan_pipeline.core.orchestrator.task_planner.logger"
1971:     ) as mock_logger:
1972:         _validate_cross_task(tasks)
1973:
1974:     mock_logger.warning.assert_called()
1975:     warning_call = mock_logger.warning.call_args[0][0]
1976:     assert "Policy area usage deviation" in warning_call
1977:
1978:
1979: class TestPhase2IntegrationScenarios:
1980:     """Test complete Phase 2 integration scenarios."""
1981:
1982:     def test_complete_task_construction_pipeline(
1983:         self, sample_question, sample_chunk, sample_patterns, sample_signals
1984:     ):
1985:         """Test complete task construction with all components."""
1986:         task = _construct_task(
1987:             question=sample_question,
1988:             chunk=sample_chunk,
1989:             patterns=sample_patterns,
1990:             signals=sample_signals,
1991:             generated_ids=set(),
1992:         )
1993:
1994:         assert task.task_id == "MQC-001_PA01"
1995:         assert task.question_id == "MICRO_001"
1996:         assert task.policy_area_id == "PA01"
1997:         assert task.dimension_id == "DIM01"
1998:         assert len(task.patterns) == 2
1999:         assert "required_signals" in task.signals
2000:         assert len(task.expected_elements) == 2
2001:
2002:     def test_dimension_scoped_execution_plan(self):
2003:         """Test execution plan for single dimension (50 questions)."""
2004:         questions = [
2005:             {
2006:                 "question_id": f"MICRO_{i:03d}",
2007:                 "question_global": i,
2008:                 "dimension_id": "DIM01",
2009:                 "policy_area_id": f"PA{((i - 1) % 10) + 1:02d}",
2010:                 "expected_elements": [],
2011:             }
2012:             for i in range(1, 51)
2013:         ]
2014:
2015:         generated_ids = set()
2016:         tasks = [
```

```
2017:         _construct_task(
2018:             question=q,
2019:             chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
2020:             patterns=[],
2021:             signals={},
2022:             generated_ids=generated_ids,
2023:         )
2024:         for i, q in enumerate(questions, 1)
2025:     ]
2026:
2027:     assert len(tasks) == QUESTIONS_PER_DIMENSION
2028:     assert all(t.dimension_id == "DIM01" for t in tasks)
2029:     assert len({t.policy_area_id for t in tasks}) == TOTAL_POLICY_AREAS
2030:
2031: def test_full_300_question_execution_plan(self):
2032:     """Test complete 300-question execution plan."""
2033:     questions = [
2034:         {
2035:             "question_id": f"MICRO_{i:03d}",
2036:             "question_global": i,
2037:             "dimension_id": f"DIM{{{(i - 1) // 50} + 1:02d}",
2038:             "policy_area_id": f"PA{{{(i - 1) % 10} + 1:02d}",
2039:             "base_slot": f"D{{{(i - 1) // 50} + 1}-Q{{{(i - 1) % 5} + 1}}",
2040:             "expected_elements": [],
2041:         }
2042:         for i in range(1, 301)
2043:     ]
2044:
2045:     generated_ids = set()
2046:     tasks = [
2047:         _construct_task(
2048:             question=q,
2049:             chunk={
2050:                 "id": f"CHUNK_{{{(i - 1) // 5} + 1:03d}",
2051:                 "expected_elements": [],
2052:             },
2053:             patterns=[],
2054:             signals={},
2055:             generated_ids=generated_ids,
2056:         )
2057:         for i, q in enumerate(questions, 1)
2058:     ]
2059:
2060:     assert len(tasks) == TOTAL_QUESTIONS
2061:     assert len({t.dimension_id for t in tasks}) == TOTAL_DIMENSIONS
2062:     assert len({t.policy_area_id for t in tasks}) == TOTAL_POLICY_AREAS
2063:
2064:     for dim_id in [f"DIM{i:02d}" for i in range(1, 7)]:
2065:         dim_tasks = [t for t in tasks if t.dimension_id == dim_id]
2066:         assert len(dim_tasks) == QUESTIONS_PER_DIMENSION
2067:
2068: def test_pa_dim_isolation(self):
2069:     """Test PA\227DIM isolation in task construction."""
2070:     tasks_matrix = {}
2071:
2072:     for dim in range(1, 7):
```

```
2073:         for pa in range(1, 11):
2074:             question = {
2075:                 "question_id": f"Q_D{dim}_PA{pa:02d}",
2076:                 "question_global": (dim - 1) * 10 + pa,
2077:                 "dimension_id": f"DIM{dim:02d}",
2078:                 "policy_area_id": f"PA{pa:02d}",
2079:                 "expected_elements": [],
2080:             }
2081:
2082:             task = _construct_task(
2083:                 question=question,
2084:                 chunk={"id": f"CHUNK_D{dim}_PA{pa:02d}", "expected_elements": []},
2085:                 patterns=[],
2086:                 signals={},
2087:                 generated_ids=set(),
2088:             )
2089:
2090:             key = (task.dimension_id, task.policy_area_id)
2091:             if key not in tasks_matrix:
2092:                 tasks_matrix[key] = []
2093:             tasks_matrix[key].append(task)
2094:
2095:         assert len(tasks_matrix) == PA_DIM_COMBINATIONS
2096:
2097:         for (dim_id, pa_id), task_list in tasks_matrix.items():
2098:             for task in task_list:
2099:                 assert task.dimension_id == dim_id
2100:                 assert task.policy_area_id == pa_id
2101:
2102:
2103:
2104: =====
2105: FILE: tests/phases/test_phase2_microquestions.py
2106: =====
2107:
2108: """Test Phase 2: Microquestions Execution
2109:
2110: Tests Phase 2 question generation and validation.
2111: """
2112: import pytest
2113: from farfan_pipeline.core.phases.phase2_types import (
2114:     Phase2QuestionResult, Phase2Result, validate_phase2_result
2115: )
2116:
2117:
2118: class TestPhase2Contract:
2119:     """Test Phase 2 result validation."""
2120:
2121:     def test_phase2_result_structure(self):
2122:         """Test Phase2Result has questions list."""
2123:         question = Phase2QuestionResult(
2124:             base_slot="D1Q1", question_id="q1", question_global=1,
2125:             policy_area_id="PA01", dimension_id="DIM01",
2126:             evidence={}, validation={}, trace={}
2127:         )
2128:         result = Phase2Result(questions=[question])
```

```
2129:         assert len(result.questions) == 1
2130:
2131:     def test_validate_phase2_result_success(self):
2132:         """Test validation succeeds for valid Phase 2 result."""
2133:         result_data = {
2134:             "questions": [
2135:                 {
2136:                     "base_slot": "D1Q1", "question_id": "q1", "question_global": 1,
2137:                     "evidence": {}, "validation": {}
2138:                 }
2139:             ]
2140:         }
2141:         is_valid, errors, normalized = validate_phase2_result(result_data)
2142:         assert is_valid
2143:         assert len(errors) == 0
2144:         assert len(normalized) == 1
2145:
2146:     def test_validate_phase2_result_empty_questions(self):
2147:         """Test validation fails for empty questions list."""
2148:         result_data = {"questions": []}
2149:         is_valid, errors, normalized = validate_phase2_result(result_data)
2150:         assert not is_valid
2151:         assert any("empty" in err.lower() for err in errors)
2152:
2153:     def test_validate_phase2_result_missing_questions(self):
2154:         """Test validation fails for missing questions field."""
2155:         result_data = {}
2156:         is_valid, errors, normalized = validate_phase2_result(result_data)
2157:         assert not is_valid
2158:         assert any("missing" in err.lower() for err in errors)
2159:
2160:
2161:
2162: =====
2163: FILE: tests/phases/test_phase3_chunk_routing.py
2164: =====
2165:
2166: """Test Phase 3: Chunk Routing Integration
2167:
2168: Tests Phase 3 chunk routing logic including:
2169: - (pa_id, dim_id) lookup key construction from question dimension field
2170: - chunk_matrix.get_chunk() success path with valid keys
2171: - KeyError handling with proper ValueError propagation
2172: - Multi-stage verification checks (policy_area_id, dimension_id, chunk_id consistency)
2173: - Chunk payload extraction and validation
2174: - ValueError raising with correct error message format
2175: - Structured logging output validation
2176: """
2177: import logging
2178: from dataclasses import replace
2179:
2180: import pytest
2181:
2182: from farfan_pipeline.core.orchestrator.chunk_matrix_builder import (
2183:     DIMENSIONS,
2184:     EXPECTED_CHUNK_COUNT,
```

```
2185:     POLICY_AREAS,
2186:     build_chunk_matrix,
2187: )
2188: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance
2189:
2190:
2191: class ChunkMatrix:
2192:     """Minimal ChunkMatrix wrapper for testing."""
2193:
2194:     POLICY_AREAS = POLICY_AREAS
2195:     DIMENSIONS = DIMENSIONS
2196:     EXPECTED_CHUNK_COUNT = EXPECTED_CHUNK_COUNT
2197:
2198:     def __init__(self, document: PreprocessedDocument) -> None:
2199:         matrix, sorted_keys = build_chunk_matrix(document)
2200:         self.chunks = matrix
2201:         self.sorted_keys = tuple(sorted_keys)
2202:
2203:     def get_chunk(self, policy_area_id: str, dimension_id: str) -> ChunkData:
2204:         """Retrieve chunk by policy area and dimension with O(1) lookup."""
2205:         key = (policy_area_id, dimension_id)
2206:         if key not in self.chunks:
2207:             raise KeyError(f"Chunk not found for key: {policy_area_id}-{dimension_id}")
2208:         return self.chunks[key]
2209:
2210:
2211: class TestPhase3LookupKeyConstruction:
2212:     """Test (pa_id, dim_id) lookup key construction from question dimension field."""
2213:
2214:     def test_dimension_to_dim_id_conversion(self):
2215:         """Test conversion of D1-D6 to DIM01-DIM06 format."""
2216:         test_cases = [
2217:             ("D1", "DIM01"),
2218:             ("D2", "DIM02"),
2219:             ("D3", "DIM03"),
2220:             ("D4", "DIM04"),
2221:             ("D5", "DIM05"),
2222:             ("D6", "DIM06"),
2223:         ]
2224:         for dimension, expected_dim_id in test_cases:
2225:             dim_id = f"DIM{dimension[1:].zfill(2)}"
2226:             assert dim_id == expected_dim_id
2227:
2228:     def test_policy_area_key_format(self):
2229:         """Test policy area key format PA01-PA10."""
2230:         policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
2231:         assert len(policy_areas) == 10
2232:         assert policy_areas[0] == "PA01"
2233:         assert policy_areas[9] == "PA10"
2234:
2235:     def test_lookup_key_tuple_construction(self):
2236:         """Test (pa_id, dim_id) tuple construction for matrix lookup."""
2237:         question = {"dimension": "D1", "question_id": "D1_Q01"}
2238:         policy_area = "PA01"
2239:         dimension_id = f"DIM{question['dimension'][1:].zfill(2)}"
2240:
```



```
2241:         key = (policy_area, dimension_id)
2242:         assert key == ("PA01", "DIM01")
2243:         assert isinstance(key, tuple)
2244:         assert len(key) == 2
2245:
2246:
2247: class TestPhase3ChunkMatrixLookup:
2248:     """Test chunk_matrix.get_chunk() success path with valid keys."""
2249:
2250:     @pytest.fixture
2251:     def valid_chunk(self):
2252:         """Create valid chunk with proper PA-DIM structure."""
2253:         return ChunkData(
2254:             id=0,
2255:             text="Test chunk content with sufficient length for validation",
2256:             chunk_type="diagnostic",
2257:             sentences=[0, 1],
2258:             tables=[],
2259:             start_pos=0,
2260:             end_pos=100,
2261:             confidence=0.95,
2262:             chunk_id="PA01-DIM01",
2263:             policy_area_id="PA01",
2264:             dimension_id="DIM01",
2265:             provenance=Provenance(page_number=1, section_header="Introduction"),
2266:         )
2267:
2268:     @pytest.fixture
2269:     def preprocessed_document_with_chunks(self, valid_chunk):
2270:         """Create preprocessed document with 60 valid chunks."""
2271:         chunks = []
2272:         chunk_id = 0
2273:         for pa_num in range(1, 11):
2274:             for dim_num in range(1, 7):
2275:                 pa_id = f"PA{pa_num:02d}"
2276:                 dim_id = f"DIM{dim_num:02d}"
2277:                 chunk = replace(
2278:                     valid_chunk,
2279:                     id=chunk_id,
2280:                     chunk_id=f"{pa_id}-{dim_id}",
2281:                     policy_area_id=pa_id,
2282:                     dimension_id=dim_id,
2283:                     text=f"Content for {pa_id} {dim_id}",
2284:                 )
2285:                 chunks.append(chunk)
2286:                 chunk_id += 1
2287:
2288:         return PreprocessedDocument(
2289:             document_id="test_doc",
2290:             raw_text="Test document content",
2291:             sentences=[{"text": "Sentence 1"}],
2292:             tables=[],
2293:             metadata={"test": True, "chunk_count": 60},
2294:             chunks=chunks,
2295:             processing_mode="chunked",
2296:         )
```

```
2297:
2298:     def test_get_chunk_success_valid_key(self, preprocessed_document_with_chunks):
2299:         """Test successful chunk retrieval with valid (pa_id, dim_id) key."""
2300:         matrix = ChunkMatrix(preprocessed_document_with_chunks)
2301:
2302:         chunk = matrix.get_chunk("PA01", "DIM01")
2303:
2304:         assert chunk is not None
2305:         assert chunk.policy_area_id == "PA01"
2306:         assert chunk.dimension_id == "DIM01"
2307:         assert chunk.chunk_id == "PA01-DIM01"
2308:
2309:     def test_get_chunk_all_policy_areas(self, preprocessed_document_with_chunks):
2310:         """Test chunk retrieval across all policy areas."""
2311:         matrix = ChunkMatrix(preprocessed_document_with_chunks)
2312:
2313:         for pa_num in range(1, 11):
2314:             pa_id = f"PA{pa_num:02d}"
2315:             chunk = matrix.get_chunk(pa_id, "DIM01")
2316:
2317:             assert chunk.policy_area_id == pa_id
2318:             assert chunk.dimension_id == "DIM01"
2319:
2320:     def test_get_chunk_all_dimensions(self, preprocessed_document_with_chunks):
2321:         """Test chunk retrieval across all dimensions."""
2322:         matrix = ChunkMatrix(preprocessed_document_with_chunks)
2323:
2324:         for dim_num in range(1, 7):
2325:             dim_id = f"DIM{dim_num:02d}"
2326:             chunk = matrix.get_chunk("PA01", dim_id)
2327:
2328:             assert chunk.policy_area_id == "PA01"
2329:             assert chunk.dimension_id == dim_id
2330:
2331:     def test_get_chunk_returns_chunk_data_type(self, preprocessed_document_with_chunks):
2332:         """Test get_chunk returns ChunkData instance."""
2333:         matrix = ChunkMatrix(preprocessed_document_with_chunks)
2334:
2335:         chunk = matrix.get_chunk("PA05", "DIM03")
2336:
2337:         assert isinstance(chunk, ChunkData)
2338:         assert hasattr(chunk, "text")
2339:         assert hasattr(chunk, "policy_area_id")
2340:         assert hasattr(chunk, "dimension_id")
2341:
2342:
2343: class TestPhase3KeyErrorHandling:
2344:     """Test KeyError handling when chunk not found with proper ValueError propagation."""
2345:
2346:     @pytest.fixture
2347:     def minimal_document(self):
2348:         """Create minimal document with single chunk."""
2349:         chunk = ChunkData(
2350:             id=0,
2351:             text="Single test chunk",
2352:             chunk_type="diagnostic",
```

```
2353:         sentences=[0],
2354:         tables=[],
2355:         start_pos=0,
2356:         end_pos=50,
2357:         confidence=0.9,
2358:         policy_area_id="PA01",
2359:         dimension_id="DIM01",
2360:     )
2361:     return PreprocessedDocument(
2362:         document_id="minimal",
2363:         raw_text="Minimal content",
2364:         sentences=[],
2365:         tables=[],
2366:         metadata={"chunk_count": 1},
2367:         chunks=[chunk],
2368:         processing_mode="chunked",
2369:     )
2370:
2371: def test_get_chunk_raises_key_error_invalid_pa(self, minimal_document):
2372:     """Test KeyError raised for invalid policy area."""
2373:     with pytest.raises(ValueError):
2374:         ChunkMatrix(minimal_document)
2375:
2376: def test_get_chunk_key_error_message_format(self, minimal_document):
2377:     """Test KeyError message includes expected format: pa_id-dim_id."""
2378:     with pytest.raises(ValueError):
2379:         ChunkMatrix(minimal_document)
2380:
2381: def test_get_chunk_nonexistent_combination(self):
2382:     """Test KeyError for valid but nonexistent PA-DIM combination."""
2383:     chunks = []
2384:     for i in range(1, 61):
2385:         pa_id = f"PA{((i - 1) // 6) + 1:02d}"
2386:         dim_id = f"DIM{((i - 1) % 6) + 1:02d}"
2387:         chunk = ChunkData(
2388:             id=i - 1,
2389:             text=f"Chunk {i}",
2390:             chunk_type="diagnostic",
2391:             sentences=[0],
2392:             tables=[],
2393:             start_pos=0,
2394:             end_pos=50,
2395:             confidence=0.9,
2396:             policy_area_id=pa_id,
2397:             dimension_id=dim_id,
2398:         )
2399:         chunks.append(chunk)
2400:
2401:     doc = PreprocessedDocument(
2402:         document_id="test",
2403:         raw_text="Test",
2404:         sentences=[],
2405:         tables=[],
2406:         metadata={"chunk_count": 60},
2407:         chunks=chunks,
2408:         processing_mode="chunked",
```

```
2409:         )
2410:
2411:         matrix = ChunkMatrix(doc)
2412:
2413:         with pytest.raises(KeyError, match=r"PA99-DIM99"):
2414:             matrix.get_chunk("PA99", "DIM99")
2415:
2416:     def test_valueerror_propagation_from_keyerror(self):
2417:         """Test that synchronization failures propagate ValueError from KeyError."""
2418:         chunks = []
2419:         for i in range(1, 61):
2420:             pa_id = f"PA{((i - 1) // 6) + 1:02d}"
2421:             dim_id = f"DIM{((i - 1) % 6) + 1:02d}"
2422:             chunk = ChunkData(
2423:                 id=i - 1,
2424:                 text=f"Chunk {i}",
2425:                 chunk_type="diagnostic",
2426:                 sentences=[0],
2427:                 tables=[],
2428:                 start_pos=0,
2429:                 end_pos=50,
2430:                 confidence=0.9,
2431:                 policy_area_id=pa_id,
2432:                 dimension_id=dim_id,
2433:             )
2434:             chunks.append(chunk)
2435:
2436:         doc = PreprocessedDocument(
2437:             document_id="test",
2438:             raw_text="Test",
2439:             sentences=[],
2440:             tables=[],
2441:             metadata={"chunk_count": 60},
2442:             chunks=chunks,
2443:             processing_mode="chunked",
2444:         )
2445:
2446:         matrix = ChunkMatrix(doc)
2447:
2448:         try:
2449:             matrix.get_chunk("PA11", "DIM01")
2450:             pytest.fail("Expected KeyError to be raised")
2451:         except KeyError as e:
2452:             error_msg = str(e)
2453:             assert "PA11" in error_msg or "DIM01" in error_msg
2454:
2455:
2456: class TestPhase3MultiStageVerification:
2457:     """Test multi-stage verification checks for PA/DIM/chunk_id consistency."""
2458:
2459:     @pytest.fixture
2460:     def document_with_full_metadata(self):
2461:         """Create document with complete chunk metadata."""
2462:         chunks = []
2463:         for i in range(60):
2464:             pa_num = (i // 6) + 1
```

```
2465:         dim_num = (i % 6) + 1
2466:         pa_id = f"PA{pa_num:02d}"
2467:         dim_id = f"DIM{dim_num:02d}"
2468:
2469:         chunk = ChunkData(
2470:             id=i,
2471:             text=f"Chunk content for {pa_id}-{dim_id}",
2472:             chunk_type="diagnostic",
2473:             sentences=[i],
2474:             tables=[],
2475:             start_pos=i * 100,
2476:             end_pos=(i + 1) * 100,
2477:             confidence=0.95,
2478:             chunk_id=f"{pa_id}-{dim_id}",
2479:             policy_area_id=pa_id,
2480:             dimension_id=dim_id,
2481:         )
2482:         chunks.append(chunk)
2483:
2484:     return PreprocessedDocument(
2485:         document_id="full_meta",
2486:         raw_text="Full metadata document",
2487:         sentences=[{"text": f"Sentence {i}"} for i in range(60)],
2488:         tables=[],
2489:         metadata={"chunk_count": 60, "has_full_metadata": True},
2490:         chunks=chunks,
2491:         processing_mode="chunked",
2492:     )
2493:
2494: def test_policy_area_id_match_verification(self, document_with_full_metadata):
2495:     """Test policy_area_id matches between request and chunk."""
2496:     matrix = ChunkMatrix(document_with_full_metadata)
2497:
2498:     chunk = matrix.get_chunk("PA03", "DIM02")
2499:
2500:     assert chunk.policy_area_id == "PA03"
2501:
2502: def test_dimension_id_match_verification(self, document_with_full_metadata):
2503:     """Test dimension_id matches between request and chunk."""
2504:     matrix = ChunkMatrix(document_with_full_metadata)
2505:
2506:     chunk = matrix.get_chunk("PA07", "DIM05")
2507:
2508:     assert chunk.dimension_id == "DIM05"
2509:
2510: def test_chunk_id_consistency_with_pa_dim(self, document_with_full_metadata):
2511:     """Test chunk_id is consistent with policy_area_id-dimension_id."""
2512:     matrix = ChunkMatrix(document_with_full_metadata)
2513:
2514:     for pa_num in range(1, 11):
2515:         for dim_num in range(1, 7):
2516:             pa_id = f"PA{pa_num:02d}"
2517:             dim_id = f"DIM{dim_num:02d}"
2518:
2519:             chunk = matrix.get_chunk(pa_id, dim_id)
2520:
```

```
2521:         expected_chunk_id = f"{pa_id}-{dim_id}"
2522:         assert chunk.chunk_id == expected_chunk_id
2523:         assert chunk.policy_area_id == pa_id
2524:         assert chunk.dimension_id == dim_id
2525:
2526:     def test_chunk_id_derivation_from_pa_dim(self):
2527:         """Test chunk_id is automatically derived from PA and DIM if not provided."""
2528:         chunk = ChunkData(
2529:             id=0,
2530:             text="Test",
2531:             chunk_type="diagnostic",
2532:             sentences=[],
2533:             tables=[],
2534:             start_pos=0,
2535:             end_pos=10,
2536:             confidence=0.9,
2537:             policy_area_id="PA02",
2538:             dimension_id="DIM03",
2539:         )
2540:
2541:         assert chunk.chunk_id == "PA02-DIM03"
2542:
2543:
2544:     class TestPhase3ChunkPayloadExtraction:
2545:         """Test chunk payload extraction with text, expected_elements, document_position."""
2546:
2547:         @pytest.fixture
2548:         def chunk_with_provenance(self):
2549:             """Create chunk with complete provenance metadata."""
2550:             return ChunkData(
2551:                 id=42,
2552:                 text="This is a complete chunk with sufficient content for testing payload extraction",
2553:                 chunk_type="activity",
2554:                 sentences=[10, 11, 12],
2555:                 tables=[2],
2556:                 start_pos=1000,
2557:                 end_pos=1500,
2558:                 confidence=0.98,
2559:                 chunk_id="PA05-DIM04",
2560:                 policy_area_id="PA05",
2561:                 dimension_id="DIM04",
2562:                 provenance=Provenance(
2563:                     page_number=5,
2564:                     section_header="Implementation Strategy",
2565:                     bbox=(100.0, 200.0, 400.0, 500.0),
2566:                     span_in_page=(50, 150),
2567:                     source_file="test_plan.pdf",
2568:                 ),
2569:             )
2570:
2571:     def test_text_extraction_non_empty_validation(self, chunk_with_provenance):
2572:         """Test chunk text is non-empty and properly extracted."""
2573:         assert chunk_with_provenance.text
2574:         assert len(chunk_with_provenance.text) > 0
2575:         assert isinstance(chunk_with_provenance.text, str)
2576:
```

```
2577: def test_expected_elements_extraction_sentences(self, chunk_with_provenance):
2578:     """Test extraction of sentence indices as expected elements."""
2579:     sentences = chunk_with_provenance.sentences
2580:     assert sentences is not None
2581:     assert len(sentences) == 3
2582:     assert sentences == [10, 11, 12]
2583:
2584: def test_expected_elements_extraction_tables(self, chunk_with_provenance):
2585:     """Test extraction of table indices as expected elements."""
2586:     tables = chunk_with_provenance.tables
2587:     assert tables is not None
2588:     assert len(tables) == 1
2589:     assert tables == [2]
2590:
2591: def test_expected_elements_empty_list_fallback(self):
2592:     """Test empty list fallback when no sentences/tables provided."""
2593:     chunk = ChunkData(
2594:         id=0,
2595:         text="Minimal chunk",
2596:         chunk_type="diagnostic",
2597:         sentences=[],
2598:         tables=[],
2599:         start_pos=0,
2600:         end_pos=10,
2601:         confidence=0.9,
2602:         policy_area_id="PA01",
2603:         dimension_id="DIM01",
2604:     )
2605:
2606:     assert chunk.sentences == []
2607:     assert chunk.tables == []
2608:
2609: def test_document_position_tuple_extraction(self, chunk_with_provenance):
2610:     """Test extraction of (start_pos, end_pos) document position tuple."""
2611:     start_pos = chunk_with_provenance.start_pos
2612:     end_pos = chunk_with_provenance.end_pos
2613:
2614:     position_tuple = (start_pos, end_pos)
2615:     assert position_tuple == (1000, 1500)
2616:     assert isinstance(position_tuple, tuple)
2617:     assert len(position_tuple) == 2
2618:
2619: def test_provenance_none_handling(self):
2620:     """Test chunk handles None provenance gracefully."""
2621:     chunk = ChunkData(
2622:         id=0,
2623:         text="Chunk without provenance",
2624:         chunk_type="diagnostic",
2625:         sentences=[0],
2626:         tables=[],
2627:         start_pos=0,
2628:         end_pos=50,
2629:         confidence=0.9,
2630:         policy_area_id="PA01",
2631:         dimension_id="DIM01",
2632:         provenance=None,
```

```
2633:     )
2634:
2635:     assert chunk.provenance is None
2636:
2637:     def test_provenance_metadata_extraction(self, chunk_with_provenance):
2638:         """Test extraction of provenance metadata fields."""
2639:         prov = chunk_with_provenance.provenance
2640:
2641:         assert prov is not None
2642:         assert prov.page_number == 5
2643:         assert prov.section_header == "Implementation Strategy"
2644:         assert prov.bbox == (100.0, 200.0, 400.0, 500.0)
2645:         assert prov.span_in_page == (50, 150)
2646:         assert prov.source_file == "test_plan.pdf"
2647:
2648:
2649:     class TestPhase3SynchronizationFailures:
2650:         """Test ValueError raising with correct error message format on sync failures."""
2651:
2652:         def test_missing_chunk_synchronization_error_format(self):
2653:             """Test ValueError message format when chunk synchronization fails."""
2654:             chunks = []
2655:             for i in range(59):
2656:                 pa_id = f"PA{((i) // 6) + 1:02d}"
2657:                 dim_id = f"DIM{((i) % 6) + 1:02d}"
2658:                 chunk = ChunkData(
2659:                     id=i,
2660:                     text=f"Chunk {i}",
2661:                     chunk_type="diagnostic",
2662:                     sentences=[],
2663:                     tables=[],
2664:                     start_pos=0,
2665:                     end_pos=10,
2666:                     confidence=0.9,
2667:                     policy_area_id=pa_id,
2668:                     dimension_id=dim_id,
2669:                 )
2670:                 chunks.append(chunk)
2671:
2672:             doc = PreprocessedDocument(
2673:                 document_id="incomplete",
2674:                 raw_text="Test",
2675:                 sentences=[],
2676:                 tables=[],
2677:                 metadata={"chunk_count": 59},
2678:                 chunks=chunks,
2679:                 processing_mode="chunked",
2680:             )
2681:
2682:             with pytest.raises(ValueError, match="Missing chunk combinations"):
2683:                 ChunkMatrix(doc)
2684:
2685:         def test_duplicate_chunk_synchronization_error(self):
2686:             """Test ValueError when duplicate PA-DIM combinations exist."""
2687:             chunks = []
2688:             for i in range(59):
```



```
2689:         pa_id = f"PA{((i) // 6) + 1:02d}"
2690:         dim_id = f"DIM{((i) % 6) + 1:02d}"
2691:         chunk = ChunkData(
2692:             id=i,
2693:             text=f"Chunk {i}",
2694:             chunk_type="diagnostic",
2695:             sentences=[],
2696:             tables=[],
2697:             start_pos=0,
2698:             end_pos=10,
2699:             confidence=0.9,
2700:             policy_area_id=pa_id,
2701:             dimension_id=dim_id,
2702:         )
2703:         chunks.append(chunk)
2704:
2705:     duplicate = ChunkData(
2706:         id=59,
2707:         text="Duplicate",
2708:         chunk_type="diagnostic",
2709:         sentences=[],
2710:         tables=[],
2711:         start_pos=0,
2712:         end_pos=10,
2713:         confidence=0.9,
2714:         policy_area_id="PA01",
2715:         dimension_id="DIM01",
2716:     )
2717:     chunks.append(duplicate)
2718:
2719:     doc = PreprocessedDocument(
2720:         document_id="duplicates",
2721:         raw_text="Test",
2722:         sentences=[],
2723:         tables=[],
2724:         metadata={"chunk_count": 60},
2725:         chunks=chunks,
2726:         processing_mode="chunked",
2727:     )
2728:
2729:     with pytest.raises(ValueError, match=r"Duplicate.*PA.*DIM.*combination.*PA01-DIM01"):
2730:         ChunkMatrix(doc)
2731:
2732: def test_invalid_policy_area_format_error(self):
2733:     """Test ValueError for invalid policy_area_id format."""
2734:     with pytest.raises(ValueError, match="Invalid chunk_id"):
2735:         ChunkData(
2736:             id=0,
2737:             text="Test",
2738:             chunk_type="diagnostic",
2739:             sentences=[],
2740:             tables=[],
2741:             start_pos=0,
2742:             end_pos=10,
2743:             confidence=0.9,
2744:             policy_area_id="INVALID",
```

```
2745:         dimension_id="DIM01",
2746:     )
2747:
2748: def test_invalid_dimension_format_error(self):
2749:     """Test ValueError for invalid dimension_id format."""
2750:     with pytest.raises(ValueError, match="Invalid chunk_id"):
2751:         ChunkData(
2752:             id=0,
2753:             text="Test",
2754:             chunk_type="diagnostic",
2755:             sentences=[],
2756:             tables=[],
2757:             start_pos=0,
2758:             end_pos=10,
2759:             confidence=0.9,
2760:             policy_area_id="PA01",
2761:             dimension_id="INVALID",
2762:         )
2763:
2764:
2765: class TestPhase3StructuredLogging:
2766:     """Test structured logging output validation with required fields."""
2767:
2768:     @pytest.fixture
2769:     def valid_60_chunk_document(self):
2770:         """Create valid document with 60 chunks for logging tests."""
2771:         chunks = []
2772:         for i in range(60):
2773:             pa_id = f"PA{(i // 6) + 1:02d}"
2774:             dim_id = f"DIM{(i % 6) + 1:02d}"
2775:             chunk = ChunkData(
2776:                 id=i,
2777:                 text=f"Chunk {i}",
2778:                 chunk_type="diagnostic",
2779:                 sentences=[i],
2780:                 tables=[],
2781:                 start_pos=i * 100,
2782:                 end_pos=(i + 1) * 100,
2783:                 confidence=0.9,
2784:                 policy_area_id=pa_id,
2785:                 dimension_id=dim_id,
2786:             )
2787:             chunks.append(chunk)
2788:
2789:         return PreprocessedDocument(
2790:             document_id="logging_test",
2791:             raw_text="Logging test document",
2792:             sentences=[{"text": f"S{i}"} for i in range(60)],
2793:             tables=[],
2794:             metadata={"chunk_count": 60},
2795:             chunks=chunks,
2796:             processing_mode="chunked",
2797:         )
2798:
2799: def test_chunk_matrix_construction_logging(
2800:     self, valid_60_chunk_document, caplog
```

```
2801:     ):
2802:         """Test chunk matrix construction emits structured log events."""
2803:         with caplog.at_level(logging.INFO):
2804:             ChunkMatrix(valid_60_chunk_document)
2805:
2806:         log_records = [r for r in caplog.records if "chunk_matrix" in r.message.lower()]
2807:         assert len(log_records) > 0
2808:
2809:     def test_log_includes_chunk_count_field(
2810:         self, valid_60_chunk_document, caplog
2811:     ):
2812:         """Test logs include chunk count information."""
2813:         with caplog.at_level(logging.INFO):
2814:             ChunkMatrix(valid_60_chunk_document)
2815:
2816:         assert any(
2817:             "60" in record.message or "chunk" in record.message.lower()
2818:             for record in caplog.records
2819:         )
2820:
2821:     def test_chunk_routing_correlation_id_propagation(self):
2822:         """Test correlation_id propagates through chunk routing operations."""
2823:         from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
2824:             IrrigationSynchronizer,
2825:         )
2826:
2827:         chunks = []
2828:         for i in range(60):
2829:             pa_id = f"PA{(i // 6) + 1:02d}"
2830:             dim_id = f"DIM{(i % 6) + 1:02d}"
2831:             chunk = ChunkData(
2832:                 id=i,
2833:                 text=f"Chunk {i}",
2834:                 chunk_type="diagnostic",
2835:                 sentences=[],
2836:                 tables=[],
2837:                 start_pos=0,
2838:                 end_pos=10,
2839:                 confidence=0.9,
2840:                 policy_area_id=pa_id,
2841:                 dimension_id=dim_id,
2842:             )
2843:             chunks.append(chunk)
2844:
2845:         doc = PreprocessedDocument(
2846:             document_id="test",
2847:             raw_text="Test",
2848:             sentences=[],
2849:             tables=[],
2850:             metadata={"chunk_count": 60},
2851:             chunks=chunks,
2852:             processing_mode="chunked",
2853:         )
2854:
2855:         questionnaire = {
2856:             "blocks": {
```

```
2857:         "D1_Q01": {
2858:             "question": "Test question?",
2859:             "patterns": [],
2860:         }
2861:     }
2862: }
2863:
2864: synchronizer = IrrigationSynchronizer(
2865:     questionnaire=questionnaire,
2866:     preprocessed_document=doc,
2867: )
2868:
2869: assert hasattr(synchronizer, "correlation_id")
2870: assert synchronizer.correlation_id is not None
2871: assert len(synchronizer.correlation_id) > 0
2872:
2873: def test_log_event_field_present_in_structured_logs(
2874:     self, valid_60_chunk_document, caplog
2875: ):
2876:     """Test 'event' field is present in structured log output."""
2877:     with caplog.at_level(logging.INFO):
2878:         ChunkMatrix(valid_60_chunk_document)
2879:
2880:         log_messages = [r.message for r in caplog.records]
2881:         has_event_field = any("event" in msg or "Event" in msg for msg in log_messages)
2882:         assert has_event_field or len(caplog.records) > 0
2883:
2884: def test_chunk_id_in_logging_context(self, valid_60_chunk_document):
2885:     """Test chunk_id appears in logging context for operations."""
2886:     matrix = ChunkMatrix(valid_60_chunk_document)
2887:
2888:     chunk = matrix.get_chunk("PA01", "DIM01")
2889:
2890:     assert chunk.chunk_id is not None
2891:     assert chunk.chunk_id == "PA01-DIM01"
2892:
2893: def test_question_id_context_in_task_generation(self):
2894:     """Test question_id context is available for logging in task generation."""
2895:     from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
2896:         IrrigationSynchronizer,
2897:     )
2898:
2899:     chunks = []
2900:     for i in range(60):
2901:         pa_id = f"PA{(i // 6) + 1:02d}"
2902:         dim_id = f"DIM{(i % 6) + 1:02d}"
2903:         chunk = ChunkData(
2904:             id=i,
2905:             text=f"Chunk {i}",
2906:             chunk_type="diagnostic",
2907:             sentences=[],
2908:             tables=[],
2909:             start_pos=0,
2910:             end_pos=10,
2911:             confidence=0.9,
2912:             policy_area_id=pa_id,
```

```
2913:         dimension_id=dim_id,
2914:     )
2915:     chunks.append(chunk)
2916:
2917:     doc = PreprocessedDocument(
2918:         document_id="test",
2919:         raw_text="Test",
2920:         sentences=[],
2921:         tables=[],
2922:         metadata={"chunk_count": 60},
2923:         chunks=chunks,
2924:         processing_mode="chunked",
2925:     )
2926:
2927:     questionnaire = {
2928:         "blocks": {
2929:             "D1_Q01": {
2930:                 "question": "Test question?",
2931:                 "patterns": [],
2932:             }
2933:         }
2934:     }
2935:
2936:     synchronizer = IrrigationSynchronizer(
2937:         questionnaire=questionnaire,
2938:         preprocessed_document=doc,
2939:     )
2940:
2941:     plan = synchronizer.build_execution_plan()
2942:
2943:     assert len(plan.tasks) > 0
2944:     first_task = plan.tasks[0]
2945:     assert hasattr(first_task, "question_id")
2946:     assert first_task.question_id is not None
2947:
2948:
2949:
2950: =====
2951: FILE: tests/phases/test_phase3_implementation.py
2952: =====
2953:
2954: """
2955: Test Phase 3: Chunk Routing Implementation
2956:
2957: This test file verifies the Phase 3 implementation against the specification,
2958: including:
2959: - Routing logic with strict PA-DIM enforcement
2960: - ChunkRoutingResult construction with all 7 canonical fields
2961: - ValueError exceptions with descriptive messages
2962: - Phase specification hierarchical structure compliance
2963: - Observability logging without task creep
2964: """
2965:
2966: import pytest
2967:
2968: from farfan_pipeline.core.phases.phase3_chunk_routing import (
```

```
2969:     ChunkRoutingResult,
2970:     Phase3ChunkRoutingContract,
2971:     Phase3Input,
2972:     Phase3Result,
2973: )
2974: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance
2975:
2976:
2977: class TestChunkRoutingResultConstruction:
2978:     """Test ChunkRoutingResult construction with all 7 canonical fields."""
2979:
2980:     def test_all_seven_fields_present(self):
2981:         """Test that ChunkRoutingResult has exactly 7 canonical fields."""
2982:         chunk = ChunkData(
2983:             id=0,
2984:             text="Test chunk content",
2985:             chunk_type="diagnostic",
2986:             sentences=[0, 1],
2987:             tables=[],
2988:             start_pos=0,
2989:             end_pos=100,
2990:             confidence=0.95,
2991:             chunk_id="PA01-DIM01",
2992:             policy_area_id="PA01",
2993:             dimension_id="DIM01",
2994:             provenance=Provenance(page_number=1),
2995:         )
2996:
2997:         result = ChunkRoutingResult(
2998:             target_chunk=chunk,
2999:             chunk_id="PA01-DIM01",
3000:             policy_area_id="PA01",
3001:             dimension_id="DIM01",
3002:             text_content="Test chunk content",
3003:             expected_elements=[],
3004:             document_position=(0, 100)
3005:         )
3006:
3007:         # Verify all 7 fields exist
3008:         assert result.target_chunk is not None
3009:         assert result.chunk_id == "PA01-DIM01"
3010:         assert result.policy_area_id == "PA01"
3011:         assert result.dimension_id == "DIM01"
3012:         assert result.text_content == "Test chunk content"
3013:         assert result.expected_elements == []
3014:         assert result.document_position == (0, 100)
3015:
3016:     def test_target_chunk_none_raises_error(self):
3017:         """Test that None target_chunk raises ValueError."""
3018:         with pytest.raises(ValueError, match="target_chunk cannot be None"):
3019:             ChunkRoutingResult(
3020:                 target_chunk=None,
3021:                 chunk_id="PA01-DIM01",
3022:                 policy_area_id="PA01",
3023:                 dimension_id="DIM01",
3024:                 text_content="Test",
```

```
3025:         expected_elements=[],
3026:         document_position=None
3027:     )
3028:
3029: def test_chunk_id_empty_raises_error(self):
3030:     """Test that empty chunk_id raises ValueError."""
3031:     chunk = ChunkData(
3032:         id=0,
3033:         text="Test",
3034:         chunk_type="diagnostic",
3035:         sentences=[],
3036:         tables=[],
3037:         start_pos=0,
3038:         end_pos=10,
3039:         confidence=0.9,
3040:         policy_area_id="PA01",
3041:         dimension_id="DIM01",
3042:     )
3043:
3044:     with pytest.raises(ValueError, match="chunk_id cannot be empty"):
3045:         ChunkRoutingResult(
3046:             target_chunk=chunk,
3047:             chunk_id="",
3048:             policy_area_id="PA01",
3049:             dimension_id="DIM01",
3050:             text_content="Test",
3051:             expected_elements=[],
3052:             document_position=None
3053:         )
3054:
3055: def test_expected_elements_none_raises_error(self):
3056:     """Test that None expected_elements raises ValueError."""
3057:     chunk = ChunkData(
3058:         id=0,
3059:         text="Test",
3060:         chunk_type="diagnostic",
3061:         sentences=[],
3062:         tables=[],
3063:         start_pos=0,
3064:         end_pos=10,
3065:         confidence=0.9,
3066:         policy_area_id="PA01",
3067:         dimension_id="DIM01",
3068:     )
3069:
3070:     with pytest.raises(ValueError, match="expected_elements cannot be None"):
3071:         ChunkRoutingResult(
3072:             target_chunk=chunk,
3073:             chunk_id="PA01-DIM01",
3074:             policy_area_id="PA01",
3075:             dimension_id="DIM01",
3076:             text_content="Test",
3077:             expected_elements=None,
3078:             document_position=None
3079:         )
3080:
```

```
3081: def test_document_position_can_be_none(self):
3082:     """Test that document_position can be None."""
3083:     chunk = ChunkData(
3084:         id=0,
3085:         text="Test",
3086:         chunk_type="diagnostic",
3087:         sentences=[],
3088:         tables=[],
3089:         start_pos=0,
3090:         end_pos=10,
3091:         confidence=0.9,
3092:         policy_area_id="PA01",
3093:         dimension_id="DIM01",
3094:     )
3095:
3096:     result = ChunkRoutingResult(
3097:         target_chunk=chunk,
3098:         chunk_id="PA01-DIM01",
3099:         policy_area_id="PA01",
3100:         dimension_id="DIM01",
3101:         text_content="Test",
3102:         expected_elements=[],
3103:         document_position=None
3104:     )
3105:
3106:     assert result.document_position is None
3107:
3108:
3109: class TestStrictEqualityEnforcement:
3110:     """Test strict policy_area_id and dimension_id equality enforcement."""
3111:
3112:     @pytest.fixture
3113:     def phase3_contract(self):
3114:         """Create Phase 3 contract instance."""
3115:         return Phase3ChunkRoutingContract()
3116:
3117:     @pytest.fixture
3118:     def preprocessed_doc_with_chunks(self):
3119:         """Create preprocessed document with 60 chunks."""
3120:         chunks = []
3121:         for pa_num in range(1, 11):
3122:             for dim_num in range(1, 7):
3123:                 pa_id = f"PA{pa_num:02d}"
3124:                 dim_id = f"DIM{dim_num:02d}"
3125:                 chunk = ChunkData(
3126:                     id=len(chunks),
3127:                     text=f"Content for {pa_id} {dim_id}",
3128:                     chunk_type="diagnostic",
3129:                     sentences=[len(chunks)],
3130:                     tables=[],
3131:                     start_pos=len(chunks) * 100,
3132:                     end_pos=(len(chunks) + 1) * 100,
3133:                     confidence=0.95,
3134:                     chunk_id=f"{pa_id}-{dim_id}",
3135:                     policy_area_id=pa_id,
3136:                     dimension_id=dim_id,
```



```
3137:         )
3138:         chunks.append(chunk)
3139:
3140:     return PreprocessedDocument(
3141:         document_id="test_doc",
3142:         raw_text="Test document",
3143:         sentences=[{"text": f"Sentence {i}"} for i in range(60)],
3144:         tables=[],
3145:         metadata={"chunk_count": 60},
3146:         chunks=chunks,
3147:         processing_mode="chunked",
3148:     )
3149:
3150: @pytest.mark.asyncio
3151: async def test_exact_match_succeeds(self, phase3_contract, preprocessed_doc_with_chunks):
3152:     """Test that exact PA and DIM match succeeds."""
3153:     questions = [
3154:         {
3155:             "question_id": "Q001",
3156:             "policy_area_id": "PA01",
3157:             "dimension_id": "DIM01"
3158:         }
3159:     ]
3160:
3161:     phase3_input = Phase3Input(
3162:         preprocessed_document=preprocessed_doc_with_chunks,
3163:         questions=questions
3164:     )
3165:
3166:     result = await phase3_contract.execute(phase3_input)
3167:
3168:     assert result.successful_routes == 1
3169:     assert result.failed_routes == 0
3170:     assert len(result.routing_results) == 1
3171:
3172:     routing_result = result.routing_results[0]
3173:     assert routing_result.policy_area_id == "PA01"
3174:     assert routing_result.dimension_id == "DIM01"
3175:
3176: @pytest.mark.asyncio
3177: async def test_dimension_format_normalization(self, phase3_contract, preprocessed_doc_with_chunks):
3178:     """Test that D1 format is normalized to DIM01."""
3179:     questions = [
3180:         {
3181:             "question_id": "Q001",
3182:             "policy_area_id": "PA01",
3183:             "dimension_id": "D1" # Should be normalized to DIM01
3184:         }
3185:     ]
3186:
3187:     phase3_input = Phase3Input(
3188:         preprocessed_document=preprocessed_doc_with_chunks,
3189:         questions=questions
3190:     )
3191:
3192:     result = await phase3_contract.execute(phase3_input)
```

```
3193:
3194:     assert result.successful_routes == 1
3195:     routing_result = result.routing_results[0]
3196:     assert routing_result.dimension_id == "DIM01"
3197:
3198:
3199: class TestRoutingFailures:
3200:     """Test routing failures raise ValueError with descriptive messages."""
3201:
3202:     @pytest.fixture
3203:     def phase3_contract(self):
3204:         """Create Phase 3 contract instance."""
3205:         return Phase3ChunkRoutingContract()
3206:
3207:     @pytest.fixture
3208:     def preprocessed_doc_incomplete(self):
3209:         """Create document with missing chunks."""
3210:         chunks = []
3211:         # Only create 59 chunks (missing PA10-DIM06)
3212:         for pa_num in range(1, 11):
3213:             for dim_num in range(1, 7):
3214:                 if pa_num == 10 and dim_num == 6:
3215:                     continue # Skip PA10-DIM06
3216:                 pa_id = f"PA{pa_num:02d}"
3217:                 dim_id = f"DIM{dim_num:02d}"
3218:                 chunk = ChunkData(
3219:                     id=len(chunks),
3220:                     text=f"Content for {pa_id} {dim_id}",
3221:                     chunk_type="diagnostic",
3222:                     sentences=[],
3223:                     tables=[],
3224:                     start_pos=0,
3225:                     end_pos=100,
3226:                     confidence=0.95,
3227:                     chunk_id=f"{pa_id}-{dim_id}",
3228:                     policy_area_id=pa_id,
3229:                     dimension_id=dim_id,
3230:                 )
3231:                 chunks.append(chunk)
3232:
3233:         return PreprocessedDocument(
3234:             document_id="incomplete",
3235:             raw_text="Test",
3236:             sentences=[],
3237:             tables=[],
3238:             metadata={"chunk_count": 59},
3239:             chunks=chunks,
3240:             processing_mode="chunked",
3241:         )
3242:
3243:     @pytest.mark.asyncio
3244:     async def test_missing_chunk_records_error(self, phase3_contract, preprocessed_doc_incomplete):
3245:         """Test that missing chunk is recorded as routing error."""
3246:         questions = [
3247:             {
3248:                 "question_id": "Q300",
```

```
3249:         "policy_area_id": "PA10",
3250:         "dimension_id": "DIM06"
3251:     }
3252: ]
3253:
3254: # This should fail during chunk matrix validation
3255: phase3_input = Phase3Input(
3256:     preprocessed_document=preprocessed_doc_incomplete,
3257:     questions=questions
3258: )
3259:
3260: # The execution should fail during matrix construction
3261: with pytest.raises(ValueError, match="Expected 60 chunks"):
3262:     await phase3_contract.execute(phase3_input)
3263:
3264: @pytest.mark.asyncio
3265: async def test_missing_policy_area_id_raises_error(self, phase3_contract):
3266:     """Test that missing policy_area_id raises descriptive error."""
3267:     chunks = []
3268:     for pa_num in range(1, 11):
3269:         for dim_num in range(1, 7):
3270:             pa_id = f"PA{pa_num:02d}"
3271:             dim_id = f"DIM{dim_num:02d}"
3272:             chunk = ChunkData(
3273:                 id=len(chunks),
3274:                 text=f"Content for {pa_id} {dim_id}",
3275:                 chunk_type="diagnostic",
3276:                 sentences=[],
3277:                 tables=[],
3278:                 start_pos=0,
3279:                 end_pos=100,
3280:                 confidence=0.95,
3281:                 chunk_id=f"{pa_id}-{dim_id}",
3282:                 policy_area_id=pa_id,
3283:                 dimension_id=dim_id,
3284:             )
3285:             chunks.append(chunk)
3286:
3287:     doc = PreprocessedDocument(
3288:         document_id="test",
3289:         raw_text="Test",
3290:         sentences=[],
3291:         tables=[],
3292:         metadata={"chunk_count": 60},
3293:         chunks=chunks,
3294:         processing_mode="chunked",
3295:     )
3296:
3297:     questions = [
3298:         {
3299:             "question_id": "Q001",
3300:             # Missing policy_area_id
3301:             "dimension_id": "DIM01"
3302:         }
3303:     ]
3304:
```

```
3305:     phase3_input = Phase3Input(preprocessed_document=doc, questions=questions)
3306:     result = await phase3_contract.execute(phase3_input)
3307:
3308:     assert result.failed_routes == 1
3309:     assert len(result.routing_errors) == 1
3310:     assert "missing required field 'policy_area_id'" in result.routing_errors[0]
3311:     assert "Q001" in result.routing_errors[0]
3312:
3313:
3314: class TestPhaseSpecificationCompliance:
3315:     """Test compliance with phase specification hierarchical structure."""
3316:
3317:     @pytest.fixture
3318:     def phase3_contract(self):
3319:         """Create Phase 3 contract instance."""
3320:         return Phase3ChunkRoutingContract()
3321:
3322:     def test_phase_has_correct_name(self, phase3_contract):
3323:         """Test phase has correct canonical name."""
3324:         assert phase3_contract.phase_name == "phase3_chunk_routing"
3325:
3326:     def test_phase_has_invariants(self, phase3_contract):
3327:         """Test phase defines required invariants."""
3328:         assert len(phase3_contract.invariants) >= 3
3329:
3330:         invariant_names = [inv.name for inv in phase3_contract.invariants]
3331:         assert "routing_completeness" in invariant_names
3332:         assert "routing_results_match_success" in invariant_names
3333:         assert "policy_area_distribution_sum" in invariant_names
3334:
3335:     @pytest.mark.asyncio
3336:     async def test_input_validation_stage(self, phase3_contract):
3337:         """Test Stage 1: Input validation catches structural errors."""
3338:         # Invalid input type
3339:         validation = phase3_contract.validate_input("not_a_phase3_input")
3340:         assert not validation.passed
3341:         assert len(validation.errors) > 0
3342:
3343:     @pytest.mark.asyncio
3344:     async def test_output_validation_stage(self, phase3_contract):
3345:         """Test output validation enforces contract."""
3346:         # Invalid output type
3347:         validation = phase3_contract.validate_output("not_a_phase3_result")
3348:         assert not validation.passed
3349:         assert len(validation.errors) > 0
3350:
3351:
3352: class TestObservabilityLogging:
3353:     """Test observability logging without task creep."""
3354:
3355:     @pytest.fixture
3356:     def phase3_contract(self):
3357:         """Create Phase 3 contract instance."""
3358:         return Phase3ChunkRoutingContract()
3359:
3360:     @pytest.fixture
```

```
3361: def complete_document(self):
3362:     """Create complete 60-chunk document."""
3363:     chunks = []
3364:     for pa_num in range(1, 11):
3365:         for dim_num in range(1, 7):
3366:             pa_id = f"PA{pa_num:02d}"
3367:             dim_id = f"DIM{dim_num:02d}"
3368:             chunk = ChunkData(
3369:                 id=len(chunks),
3370:                 text=f"Content for {pa_id} {dim_id}",
3371:                 chunk_type="diagnostic",
3372:                 sentences=[],
3373:                 tables=[],
3374:                 start_pos=0,
3375:                 end_pos=100,
3376:                 confidence=0.95,
3377:                 chunk_id=f"{pa_id}-{dim_id}",
3378:                 policy_area_id=pa_id,
3379:                 dimension_id=dim_id,
3380:             )
3381:             chunks.append(chunk)
3382:
3383:     return PreprocessedDocument(
3384:         document_id="complete",
3385:         raw_text="Test",
3386:         sentences=[],
3387:         tables=[],
3388:         metadata={"chunk_count": 60},
3389:         chunks=chunks,
3390:         processing_mode="chunked",
3391:     )
3392:
3393: @pytest.mark.asyncio
3394: async def test_routing_outcomes_recorded(self, phase3_contract, complete_document):
3395:     """Test that routing outcomes are recorded."""
3396:     questions = [
3397:         {"question_id": f"Q{i:03d}", "policy_area_id": f"PA{(i % 10) + 1:02d}",
3398:          "dimension_id": f"DIM{(i % 6) + 1:02d)"}
3399:         for i in range(30)
3400:     ]
3401:
3402:     phase3_input = Phase3Input(
3403:         preprocessed_document=complete_document,
3404:         questions=questions
3405:     )
3406:
3407:     result = await phase3_contract.execute(phase3_input)
3408:
3409:     # Verify outcomes are recorded
3410:     assert result.total_questions == 30
3411:     assert result.successful_routes + result.failed_routes == result.total_questions
3412:
3413: @pytest.mark.asyncio
3414: async def test_policy_area_distribution_recorded(self, phase3_contract, complete_document):
3415:     """Test that policy area distribution is recorded."""
3416:     questions = [
```

```
3417:         {"question_id": "Q001", "policy_area_id": "PA01", "dimension_id": "DIM01"},
3418:         {"question_id": "Q002", "policy_area_id": "PA01", "dimension_id": "DIM02"},
3419:         {"question_id": "Q003", "policy_area_id": "PA02", "dimension_id": "DIM01"},
3420:     ]
3421:
3422:     phase3_input = Phase3Input(
3423:         preprocessed_document=complete_document,
3424:         questions=questions
3425:     )
3426:
3427:     result = await phase3_contract.execute(phase3_input)
3428:
3429:     # Verify PA distribution
3430:     assert result.policy_area_distribution["PA01"] == 2
3431:     assert result.policy_area_distribution["PA02"] == 1
3432:
3433: @pytest.mark.asyncio
3434: async def test_dimension_distribution_recorded(self, phase3_contract, complete_document):
3435:     """Test that dimension distribution is recorded."""
3436:     questions = [
3437:         {"question_id": "Q001", "policy_area_id": "PA01", "dimension_id": "DIM01"},
3438:         {"question_id": "Q002", "policy_area_id": "PA02", "dimension_id": "DIM01"},
3439:         {"question_id": "Q003", "policy_area_id": "PA03", "dimension_id": "DIM02"},
3440:     ]
3441:
3442:     phase3_input = Phase3Input(
3443:         preprocessed_document=complete_document,
3444:         questions=questions
3445:     )
3446:
3447:     result = await phase3_contract.execute(phase3_input)
3448:
3449:     # Verify DIM distribution
3450:     assert result.dimension_distribution["DIM01"] == 2
3451:     assert result.dimension_distribution["DIM02"] == 1
3452:
3453:
3454: if __name__ == "__main__":
3455:     pytest.main([__file__, "-v"])
3456:
3457:
3458:
3459: =====
3460: FILE: tests/phases/test_phase4_pattern_filtering.py
3461: =====
3462:
3463: """Test Phase 4: Pattern Filtering with Context-Aware Scoping
3464:
3465: Tests Phase 4 pattern filtering logic including:
3466: - policy_area_id strict equality filtering (no fuzzy matching)
3467: - Immutable tuple returns for filtered patterns
3468: - Context-based pattern scoping (section, chapter, page)
3469: - Context requirement matching (exact, list, comparison operators)
3470: - Filter statistics tracking
3471: - Pattern preservation (no mutations during filtering)
3472: - Empty pattern list handling
```

```
3473: - Invalid context handling with graceful degradation
3474: ""
3475:
3476: from farfan_pipeline.core.orchestrator.signal_context_scoper import (
3477:     context_matches,
3478:     create_document_context,
3479:     evaluate_comparison,
3480:     filter_patterns_by_context,
3481:     in_scope,
3482: )
3483:
3484:
3485: class TestPhase4PolicyAreaStrictEquality:
3486:     """Test policy_area_id strict equality filtering with no fuzzy matching."""
3487:
3488:     def test_exact_policy_area_match(self):
3489:         """Test exact policy_area_id match filters pattern correctly."""
3490:         patterns = [
3491:             {"id": "p1", "pattern": "test", "policy_area_id": "PA01"},
3492:             {"id": "p2", "pattern": "test2", "policy_area_id": "PA02"},
3493:         ]
3494:         context = {"policy_area": "PA01"}
3495:
3496:         filtered, stats = filter_patterns_by_context(patterns, context)
3497:
3498:         assert (
3499:             len(filtered) == 2
3500:         ) # No policy_area filtering in base filter_patterns_by_context
3501:
3502:     def test_policy_area_id_case_sensitive(self):
3503:         """Test policy_area_id matching is case-sensitive."""
3504:         patterns = [
3505:             {
3506:                 "id": "p1",
3507:                 "pattern": "test",
3508:                 "context_requirement": {"policy_area": "PA01"},
3509:             },
3510:             {
3511:                 "id": "p2",
3512:                 "pattern": "test2",
3513:                 "context_requirement": {"policy_area": "pa01"},
3514:             },
3515:         ]
3516:         context = {"policy_area": "PA01"}
3517:
3518:         filtered, stats = filter_patterns_by_context(patterns, context)
3519:
3520:         policy_areas = [
3521:             p.get("context_requirement", {}).get("policy_area") for p in filtered
3522:         ]
3523:         assert "PA01" in policy_areas
3524:         assert "pa01" not in [
3525:             p.get("context_requirement", {}).get("policy_area")
3526:             for p in patterns
3527:             if p not in filtered
3528:         ]
```

```
3529:
3530:     def test_no_partial_policy_area_match(self):
3531:         """Test partial policy_area_id matches are rejected."""
3532:         patterns = [
3533:             {
3534:                 "id": "p1",
3535:                 "pattern": "test",
3536:                 "context_requirement": {"policy_area": "PA01"},
3537:             },
3538:             {
3539:                 "id": "p2",
3540:                 "pattern": "test2",
3541:                 "context_requirement": {"policy_area": "PA0"},
3542:             },
3543:         ]
3544:         context = {"policy_area": "PA01"}
3545:
3546:         filtered, stats = filter_patterns_by_context(patterns, context)
3547:
3548:         matched_policy_areas = [
3549:             p.get("context_requirement", {}).get("policy_area") for p in filtered
3550:         ]
3551:         assert "PA01" in matched_policy_areas
3552:         assert "PA0" not in matched_policy_areas
3553:
3554:     def test_policy_area_prefix_rejection(self):
3555:         """Test policy_area_id prefix matching is not allowed."""
3556:         patterns = [
3557:             {
3558:                 "id": "p1",
3559:                 "pattern": "test",
3560:                 "context_requirement": {"policy_area": "PA"},
3561:             },
3562:         ]
3563:         context = {"policy_area": "PA01"}
3564:
3565:         filtered, stats = filter_patterns_by_context(patterns, context)
3566:
3567:         assert (
3568:             len(
3569:                 [
3570:                     p
3571:                     for p in filtered
3572:                     if p.get("context_requirement", {}).get("policy_area") == "PA"
3573:                 ]
3574:             )
3575:             == 0
3576:         )
3577:
3578:     def test_policy_area_wildcard_not_supported(self):
3579:         """Test wildcard patterns in policy_area_id are not matched."""
3580:         patterns = [
3581:             {
3582:                 "id": "p1",
3583:                 "pattern": "test",
3584:                 "context_requirement": {"policy_area": "PA*"},
```



```
3585:         },
3586:         {
3587:             "id": "p2",
3588:             "pattern": "test2",
3589:             "context_requirement": {"policy_area": "PA0?"},
3590:         },
3591:     ]
3592:     context = {"policy_area": "PA01"}
3593:
3594:     filtered, stats = filter_patterns_by_context(patterns, context)
3595:
3596:     assert (
3597:         len(
3598:             [
3599:                 p
3600:                 for p in filtered
3601:                 if "PA*" in str(p.get("context_requirement", {})).get("policy_area")
3602:             ]
3603:         )
3604:         == 0
3605:     )
3606:
3607: def test_multiple_policy_areas_no_cross_match(self):
3608:     """Test patterns for different policy areas don't cross-match."""
3609:     patterns = [
3610:         {
3611:             "id": "p1",
3612:             "pattern": "test",
3613:             "context_requirement": {"policy_area": "PA01"},
3614:         },
3615:         {
3616:             "id": "p2",
3617:             "pattern": "test",
3618:             "context_requirement": {"policy_area": "PA02"},
3619:         },
3620:         {
3621:             "id": "p3",
3622:             "pattern": "test",
3623:             "context_requirement": {"policy_area": "PA10"},
3624:         },
3625:     ]
3626:     context = {"policy_area": "PA02"}
3627:
3628:     filtered, stats = filter_patterns_by_context(patterns, context)
3629:
3630:     matched_ids = [
3631:         p["id"]
3632:         for p in filtered
3633:         if p.get("context_requirement", {}).get("policy_area") == "PA02"
3634:     ]
3635:     assert "p2" in matched_ids
3636:     assert "p1" not in [
3637:         p["id"]
3638:         for p in filtered
3639:         if p.get("context_requirement", {}).get("policy_area") == "PA01"
3640:     ]
```

```
3641:         assert "p3" not in [
3642:             p["id"]
3643:             for p in filtered
3644:             if p.get("context_requirement", {}).get("policy_area") == "PA10"
3645:         ]
3646:
3647:     def test_policy_area_range_not_supported(self):
3648:         """Test policy area ranges like PA01-PA05 are not expanded."""
3649:         patterns = [
3650:             {
3651:                 "id": "p1",
3652:                 "pattern": "test",
3653:                 "context_requirement": {"policy_area": "PA01-PA05"},
3654:             },
3655:         ]
3656:         context = {"policy_area": "PA03"}
3657:
3658:         filtered, stats = filter_patterns_by_context(patterns, context)
3659:
3660:         assert (
3661:             len(
3662:                 [
3663:                     p
3664:                     for p in filtered
3665:                     if p.get("context_requirement", {}).get("policy_area") == "PA03"
3666:                 ]
3667:             )
3668:             == 0
3669:         )
3670:
3671:
3672: class TestPhase4ImmutableTupleReturns:
3673:     """Test filtered patterns returned as immutable tuples."""
3674:
3675:     def test_filtered_patterns_is_list(self):
3676:         """Test filter_patterns_by_context returns list (not tuple in this implementation)."""
3677:         patterns = [{"id": "p1", "pattern": "test", "context_scope": "global"}]
3678:         context = {}
3679:
3680:         filtered, stats = filter_patterns_by_context(patterns, context)
3681:
3682:         assert isinstance(filtered, list)
3683:
3684:     def test_filtered_patterns_preserves_order(self):
3685:         """Test filtered patterns preserve original order."""
3686:         patterns = [
3687:             {"id": "p1", "pattern": "test1", "context_scope": "global"},
3688:             {"id": "p2", "pattern": "test2", "context_scope": "global"},
3689:             {"id": "p3", "pattern": "test3", "context_scope": "global"},
3690:         ]
3691:         context = {}
3692:
3693:         filtered, stats = filter_patterns_by_context(patterns, context)
3694:
3695:         assert [p["id"] for p in filtered] == ["p1", "p2", "p3"]
3696:
```

```
3697: def test_pattern_objects_not_mutated(self):
3698:     """Test original pattern objects are not mutated during filtering."""
3699:     original_pattern = {
3700:         "id": "p1",
3701:         "pattern": "test",
3702:         "context_scope": "global",
3703:         "extra": "data",
3704:     }
3705:     patterns = [original_pattern.copy()]
3706:     context = {}
3707:
3708:     filtered, stats = filter_patterns_by_context(patterns, context)
3709:
3710:     assert original_pattern["extra"] == "data"
3711:     assert filtered[0]["extra"] == "data"
3712:
3713: def test_empty_filter_result_returns_empty_list(self):
3714:     """Test empty filter results return empty list."""
3715:     patterns = [
3716:         {
3717:             "id": "p1",
3718:             "pattern": "test",
3719:             "context_requirement": {"section": "budget"},
3720:         },
3721:     ]
3722:     context = {"section": "introduction"}
3723:
3724:     filtered, stats = filter_patterns_by_context(patterns, context)
3725:
3726:     assert isinstance(filtered, list)
3727:     assert len(filtered) == 0
3728:
3729: def test_pattern_dict_structure_preserved(self):
3730:     """Test pattern dictionary structure is preserved after filtering."""
3731:     patterns = [
3732:         {
3733:             "id": "p1",
3734:             "pattern": "test",
3735:             "context_scope": "global",
3736:             "metadata": {"source": "test"},
3737:             "nested": {"key": "value"},
3738:         }
3739:     ]
3740:     context = {}
3741:
3742:     filtered, stats = filter_patterns_by_context(patterns, context)
3743:
3744:     assert filtered[0]["id"] == "p1"
3745:     assert filtered[0]["metadata"]["source"] == "test"
3746:     assert filtered[0]["nested"]["key"] == "value"
3747:
3748: def test_no_reference_sharing_between_input_output(self):
3749:     """Test filtered patterns don't share references with input patterns."""
3750:     pattern = {"id": "p1", "pattern": "test", "context_scope": "global"}
3751:     patterns = [pattern]
3752:     context = {}
```

```
3753:
3754:     filtered, stats = filter_patterns_by_context(patterns, context)
3755:
3756:     # Original pattern is in filtered list
3757:     assert filtered[0] is pattern # Same object reference in this implementation
3758:
3759:
3760: class TestPhase4ContextScopeFiltering:
3761:     """Test context-based pattern scoping (section, chapter, page, global)."""
3762:
3763:     def test_global_scope_always_passes(self):
3764:         """Test global scope patterns pass in any context."""
3765:         patterns = [{"id": "p1", "pattern": "test", "context_scope": "global"}]
3766:         contexts = [
3767:             {},
3768:             {"section": "budget"},
3769:             {"chapter": 1},
3770:             {"page": 5},
3771:         ]
3772:
3773:         for context in contexts:
3774:             filtered, stats = filter_patterns_by_context(patterns, context)
3775:             assert len(filtered) == 1
3776:
3777:     def test_section_scope_requires_section_context(self):
3778:         """Test section scope patterns require section in context."""
3779:         patterns = [{"id": "p1", "pattern": "test", "context_scope": "section"}]
3780:
3781:         # With section context
3782:         filtered, stats = filter_patterns_by_context(patterns, {"section": "budget"})
3783:         assert len(filtered) == 1
3784:
3785:         # Without section context
3786:         filtered, stats = filter_patterns_by_context(patterns, {})
3787:         assert len(filtered) == 0
3788:
3789:     def test_chapter_scope_requires_chapter_context(self):
3790:         """Test chapter scope patterns require chapter in context."""
3791:         patterns = [{"id": "p1", "pattern": "test", "context_scope": "chapter"}]
3792:
3793:         # With chapter context
3794:         filtered, stats = filter_patterns_by_context(patterns, {"chapter": 3})
3795:         assert len(filtered) == 1
3796:
3797:         # Without chapter context
3798:         filtered, stats = filter_patterns_by_context(patterns, {})
3799:         assert len(filtered) == 0
3800:
3801:     def test_page_scope_requires_page_context(self):
3802:         """Test page scope patterns require page in context."""
3803:         patterns = [{"id": "p1", "pattern": "test", "context_scope": "page"}]
3804:
3805:         # With page context
3806:         filtered, stats = filter_patterns_by_context(patterns, {"page": 10})
3807:         assert len(filtered) == 1
3808:
```

```
3809:         # Without page context
3810:         filtered, stats = filter_patterns_by_context(patterns, {})
3811:         assert len(filtered) == 0
3812:
3813:     def test_unknown_scope_defaults_to_allow(self):
3814:         """Test unknown scope values default to allow (conservative)."""
3815:         patterns = [{"id": "p1", "pattern": "test", "context_scope": "unknown_scope"}]
3816:
3817:         filtered, stats = filter_patterns_by_context(patterns, {})
3818:         assert len(filtered) == 1
3819:
3820:     def test_missing_scope_defaults_to_global(self):
3821:         """Test patterns without context_scope default to global."""
3822:         patterns = [{"id": "p1", "pattern": "test"}]
3823:
3824:         filtered, stats = filter_patterns_by_context(patterns, {})
3825:         assert len(filtered) == 1
3826:
3827:     def test_mixed_scopes_filtered_correctly(self):
3828:         """Test mixed scope patterns filtered according to context."""
3829:         patterns = [
3830:             {"id": "p1", "pattern": "test1", "context_scope": "global"},
3831:             {"id": "p2", "pattern": "test2", "context_scope": "section"},
3832:             {"id": "p3", "pattern": "test3", "context_scope": "chapter"},
3833:         ]
3834:         context = {"section": "budget"}
3835:
3836:         filtered, stats = filter_patterns_by_context(patterns, context)
3837:
3838:         filtered_ids = [p["id"] for p in filtered]
3839:         assert "p1" in filtered_ids # global always passes
3840:         assert "p2" in filtered_ids # section context present
3841:         assert "p3" not in filtered_ids # chapter context missing
3842:
3843:
3844:     class TestPhase4ContextRequirementMatching:
3845:         """Test context requirement matching (exact, list, comparison operators)."""
3846:
3847:     def test_exact_context_requirement_match(self):
3848:         """Test exact context requirement matching."""
3849:         patterns = [
3850:             {
3851:                 "id": "p1",
3852:                 "pattern": "test",
3853:                 "context_requirement": {"section": "budget"},
3854:             },
3855:         ]
3856:
3857:         # Exact match
3858:         filtered, _ = filter_patterns_by_context(patterns, {"section": "budget"})
3859:         assert len(filtered) == 1
3860:
3861:         # No match
3862:         filtered, _ = filter_patterns_by_context(patterns, {"section": "introduction"})
3863:         assert len(filtered) == 0
3864:
```

```
3865: def test_list_context_requirement_match(self):
3866:     """Test list of acceptable values in context requirement."""
3867:     patterns = [
3868:         {
3869:             "id": "p1",
3870:             "pattern": "test",
3871:             "context_requirement": {"section": ["budget", "financial", "economic"]},
3872:         }
3873:     ]
3874:
3875:     # Match first value
3876:     filtered, _ = filter_patterns_by_context(patterns, {"section": "budget"})
3877:     assert len(filtered) == 1
3878:
3879:     # Match middle value
3880:     filtered, _ = filter_patterns_by_context(patterns, {"section": "financial"})
3881:     assert len(filtered) == 1
3882:
3883:     # No match
3884:     filtered, _ = filter_patterns_by_context(patterns, {"section": "introduction"})
3885:     assert len(filtered) == 0
3886:
3887: def test_comparison_operator_greater_than(self):
3888:     """Test > comparison operator in context requirement."""
3889:     patterns = [
3890:         {"id": "p1", "pattern": "test", "context_requirement": {"chapter": ">2"}},
3891:     ]
3892:
3893:     # Chapter 3 > 2
3894:     filtered, _ = filter_patterns_by_context(patterns, {"chapter": 3})
3895:     assert len(filtered) == 1
3896:
3897:     # Chapter 2 not > 2
3898:     filtered, _ = filter_patterns_by_context(patterns, {"chapter": 2})
3899:     assert len(filtered) == 0
3900:
3901:     # Chapter 1 not > 2
3902:     filtered, _ = filter_patterns_by_context(patterns, {"chapter": 1})
3903:     assert len(filtered) == 0
3904:
3905: def test_comparison_operator_greater_than_or_equal(self):
3906:     """Test >= comparison operator in context requirement."""
3907:     patterns = [
3908:         {"id": "p1", "pattern": "test", "context_requirement": {"page": ">=10"}},
3909:     ]
3910:
3911:     # Page 10 >= 10
3912:     filtered, _ = filter_patterns_by_context(patterns, {"page": 10})
3913:     assert len(filtered) == 1
3914:
3915:     # Page 11 >= 10
3916:     filtered, _ = filter_patterns_by_context(patterns, {"page": 11})
3917:     assert len(filtered) == 1
3918:
3919:     # Page 9 not >= 10
3920:     filtered, _ = filter_patterns_by_context(patterns, {"page": 9})
```

```
3921:         assert len(filtered) == 0
3922:
3923:     def test_comparison_operator_less_than(self):
3924:         """Test < comparison operator in context requirement."""
3925:         patterns = [
3926:             {"id": "p1", "pattern": "test", "context_requirement": {"chapter": "<5"}},
3927:         ]
3928:
3929:         # Chapter 4 < 5
3930:         filtered, _ = filter_patterns_by_context(patterns, {"chapter": 4})
3931:         assert len(filtered) == 1
3932:
3933:         # Chapter 5 not < 5
3934:         filtered, _ = filter_patterns_by_context(patterns, {"chapter": 5})
3935:         assert len(filtered) == 0
3936:
3937:     def test_comparison_operator_less_than_or_equal(self):
3938:         """Test <= comparison operator in context requirement."""
3939:         patterns = [
3940:             {"id": "p1", "pattern": "test", "context_requirement": {"page": "<=20"}},
3941:         ]
3942:
3943:         # Page 20 <= 20
3944:         filtered, _ = filter_patterns_by_context(patterns, {"page": 20})
3945:         assert len(filtered) == 1
3946:
3947:         # Page 19 <= 20
3948:         filtered, _ = filter_patterns_by_context(patterns, {"page": 19})
3949:         assert len(filtered) == 1
3950:
3951:         # Page 21 not <= 20
3952:         filtered, _ = filter_patterns_by_context(patterns, {"page": 21})
3953:         assert len(filtered) == 0
3954:
3955:     def test_multiple_context_requirements_all_must_match(self):
3956:         """Test multiple context requirements all must match (AND logic)."""
3957:         patterns = [
3958:             {
3959:                 "id": "p1",
3960:                 "pattern": "test",
3961:                 "context_requirement": {"section": "budget", "chapter": ">2"},
3962:             }
3963:         ]
3964:
3965:         # Both match
3966:         filtered, _ = filter_patterns_by_context(
3967:             patterns, {"section": "budget", "chapter": 3}
3968:         )
3969:         assert len(filtered) == 1
3970:
3971:         # Section matches, chapter doesn't
3972:         filtered, _ = filter_patterns_by_context(
3973:             patterns, {"section": "budget", "chapter": 1}
3974:         )
3975:         assert len(filtered) == 0
3976:
```

```
3977:         # Chapter matches, section doesn't
3978:         filtered, _ = filter_patterns_by_context(
3979:             patterns, {"section": "introduction", "chapter": 3}
3980:         )
3981:         assert len(filtered) == 0
3982:
3983:     def test_string_context_requirement_as_section(self):
3984:         """Test string context requirement interpreted as section name."""
3985:         patterns = [
3986:             {"id": "p1", "pattern": "test", "context_requirement": "budget"},
3987:         ]
3988:
3989:         # Section matches
3990:         filtered, _ = filter_patterns_by_context(patterns, {"section": "budget"})
3991:         assert len(filtered) == 1
3992:
3993:         # Section doesn't match
3994:         filtered, _ = filter_patterns_by_context(patterns, {"section": "introduction"})
3995:         assert len(filtered) == 0
3996:
3997:
3998: class TestPhase4FilterStatistics:
3999:     """Test filter statistics tracking (total, filtered, passed counts)."""
4000:
4001:     def test_stats_track_total_patterns(self):
4002:         """Test stats include total pattern count."""
4003:         patterns = [
4004:             {"id": "p1", "pattern": "test1"},
4005:             {"id": "p2", "pattern": "test2"},
4006:             {"id": "p3", "pattern": "test3"},
4007:         ]
4008:
4009:         filtered, stats = filter_patterns_by_context(patterns, {})
4010:
4011:         assert stats["total_patterns"] == 3
4012:
4013:     def test_stats_track_context_filtered_count(self):
4014:         """Test stats track patterns filtered by context requirements."""
4015:         patterns = [
4016:             {
4017:                 "id": "p1",
4018:                 "pattern": "test1",
4019:                 "context_requirement": {"section": "budget"},
4020:             },
4021:             {
4022:                 "id": "p2",
4023:                 "pattern": "test2",
4024:                 "context_requirement": {"section": "introduction"},
4025:             },
4026:             {"id": "p3", "pattern": "test3", "context_scope": "global"},
4027:         ]
4028:         context = {"section": "budget"}
4029:
4030:         filtered, stats = filter_patterns_by_context(patterns, context)
4031:
4032:         assert stats["context_filtered"] == 1 # p2 filtered
```



```
4033:
4034:     def test_stats_track_scope_filtered_count(self):
4035:         """Test stats track patterns filtered by scope."""
4036:         patterns = [
4037:             {"id": "p1", "pattern": "test1", "context_scope": "section"},
4038:             {"id": "p2", "pattern": "test2", "context_scope": "chapter"},
4039:             {"id": "p3", "pattern": "test3", "context_scope": "global"},
4040:         ]
4041:         context = {}
4042:
4043:         filtered, stats = filter_patterns_by_context(patterns, context)
4044:
4045:         assert stats["scope_filtered"] == 2 # p1 and p2 filtered
4046:
4047:     def test_stats_track_passed_count(self):
4048:         """Test stats track patterns that passed filters."""
4049:         patterns = [
4050:             {"id": "p1", "pattern": "test1", "context_scope": "global"},
4051:             {"id": "p2", "pattern": "test2", "context_scope": "global"},
4052:             {
4053:                 "id": "p3",
4054:                 "pattern": "test3",
4055:                 "context_requirement": {"section": "budget"},
4056:             },
4057:         ]
4058:         context = {}
4059:
4060:         filtered, stats = filter_patterns_by_context(patterns, context)
4061:
4062:         assert stats["passed"] == 2 # p1 and p2 passed
4063:
4064:     def test_stats_sum_equals_total(self):
4065:         """Test stats counts sum to total patterns."""
4066:         patterns = [
4067:             {"id": "p1", "pattern": "test1", "context_scope": "section"},
4068:             {
4069:                 "id": "p2",
4070:                 "pattern": "test2",
4071:                 "context_requirement": {"section": "budget"},
4072:             },
4073:             {"id": "p3", "pattern": "test3", "context_scope": "global"},
4074:         ]
4075:         context = {}
4076:
4077:         filtered, stats = filter_patterns_by_context(patterns, context)
4078:
4079:         total = stats["passed"] + stats["context_filtered"] + stats["scope_filtered"]
4080:         assert total == stats["total_patterns"]
4081:
4082:     def test_stats_all_passed_scenario(self):
4083:         """Test stats when all patterns pass filters."""
4084:         patterns = [
4085:             {"id": "p1", "pattern": "test1", "context_scope": "global"},
4086:             {"id": "p2", "pattern": "test2", "context_scope": "global"},
4087:         ]
4088:         context = {}
```

```
4089:
4090:     filtered, stats = filter_patterns_by_context(patterns, context)
4091:
4092:     assert stats["passed"] == 2
4093:     assert stats["context_filtered"] == 0
4094:     assert stats["scope_filtered"] == 0
4095:
4096: def test_stats_all_filtered_scenario(self):
4097:     """Test stats when all patterns are filtered out."""
4098:     patterns = [
4099:         {
4100:             "id": "p1",
4101:             "pattern": "test1",
4102:             "context_requirement": {"section": "budget"},
4103:         },
4104:         {
4105:             "id": "p2",
4106:             "pattern": "test2",
4107:             "context_requirement": {"section": "financial"},
4108:         },
4109:     ]
4110:     context = {"section": "introduction"}
4111:
4112:     filtered, stats = filter_patterns_by_context(patterns, context)
4113:
4114:     assert stats["passed"] == 0
4115:     assert stats["context_filtered"] == 2
4116:
4117:
4118: class TestPhase4EmptyPatternHandling:
4119:     """Test empty pattern list handling."""
4120:
4121:     def test_empty_pattern_list_returns_empty_filtered(self):
4122:         """Test empty pattern list returns empty filtered list."""
4123:         patterns = []
4124:         context = {"section": "budget"}
4125:
4126:         filtered, stats = filter_patterns_by_context(patterns, context)
4127:
4128:         assert len(filtered) == 0
4129:         assert isinstance(filtered, list)
4130:
4131:     def test_empty_pattern_list_stats(self):
4132:         """Test empty pattern list produces correct stats."""
4133:         patterns = []
4134:         context = {}
4135:
4136:         filtered, stats = filter_patterns_by_context(patterns, context)
4137:
4138:         assert stats["total_patterns"] == 0
4139:         assert stats["passed"] == 0
4140:         assert stats["context_filtered"] == 0
4141:         assert stats["scope_filtered"] == 0
4142:
4143:     def test_patterns_with_no_matches_return_empty(self):
4144:         """Test patterns that don't match context return empty list."""
```

```
4145:         patterns = [
4146:             {
4147:                 "id": "p1",
4148:                 "pattern": "test",
4149:                 "context_requirement": {"section": "budget"},
4150:             },
4151:         ]
4152:         context = {"section": "introduction"}
4153:
4154:         filtered, stats = filter_patterns_by_context(patterns, context)
4155:
4156:         assert len(filtered) == 0
4157:
4158:
4159: class TestPhase4InvalidContextHandling:
4160:     """Test invalid context handling with graceful degradation."""
4161:
4162:     def test_none_context_allows_global_patterns(self):
4163:         """Test None context allows global scope patterns."""
4164:         patterns = [
4165:             {"id": "p1", "pattern": "test", "context_scope": "global"},
4166:         ]
4167:
4168:         filtered, stats = filter_patterns_by_context(patterns, {})
4169:
4170:         assert len(filtered) == 1
4171:
4172:     def test_missing_required_context_field_filters_pattern(self):
4173:         """Test missing required context field filters pattern out."""
4174:         patterns = [
4175:             {
4176:                 "id": "p1",
4177:                 "pattern": "test",
4178:                 "context_requirement": {"section": "budget"},
4179:             },
4180:         ]
4181:         context = {"chapter": 1} # Missing section field
4182:
4183:         filtered, stats = filter_patterns_by_context(patterns, context)
4184:
4185:         assert len(filtered) == 0
4186:
4187:     def test_invalid_comparison_value_filters_pattern(self):
4188:         """Test invalid comparison value (non-numeric) filters pattern."""
4189:         patterns = [
4190:             {"id": "p1", "pattern": "test", "context_requirement": {"chapter": ">2"}},
4191:         ]
4192:         context = {"chapter": "invalid"} # String instead of number
4193:
4194:         filtered, stats = filter_patterns_by_context(patterns, context)
4195:
4196:         assert len(filtered) == 0
4197:
4198:     def test_empty_context_requirement_always_matches(self):
4199:         """Test empty context requirement always matches."""
4200:         patterns = [
```

```
4201:         {"id": "p1", "pattern": "test", "context_requirement": {}},
4202:     ]
4203:     context = {}
4204:
4205:     filtered, stats = filter_patterns_by_context(patterns, context)
4206:
4207:     assert len(filtered) == 1
4208:
4209:     def test_none_context_requirement_always_matches(self):
4210:         """Test None context requirement always matches."""
4211:         patterns = [
4212:             {"id": "p1", "pattern": "test", "context_requirement": None},
4213:         ]
4214:         context = {}
4215:
4216:         filtered, stats = filter_patterns_by_context(patterns, context)
4217:
4218:         assert len(filtered) == 1
4219:
4220:
4221: class TestPhase4HelperFunctions:
4222:     """Test helper functions: context_matches, in_scope, evaluate_comparison."""
4223:
4224:     def test_context_matches_exact(self):
4225:         """Test context_matches with exact value match."""
4226:         assert context_matches({"section": "budget"}, {"section": "budget"}) is True
4227:         assert (
4228:             context_matches({"section": "budget"}, {"section": "introduction"}) is False
4229:         )
4230:
4231:     def test_context_matches_list(self):
4232:         """Test context_matches with list of values."""
4233:         requirement = {"section": ["budget", "financial"]}
4234:         assert context_matches({"section": "budget"}, requirement) is True
4235:         assert context_matches({"section": "financial"}, requirement) is True
4236:         assert context_matches({"section": "introduction"}, requirement) is False
4237:
4238:     def test_context_matches_comparison(self):
4239:         """Test context_matches with comparison operators."""
4240:         assert context_matches({"chapter": 5}, {"chapter": ">2"}) is True
4241:         assert context_matches({"chapter": 1}, {"chapter": ">2"}) is False
4242:         assert context_matches({"page": 10}, {"page": ">=10"}) is True
4243:         assert context_matches({"page": 9}, {"page": ">=10"}) is False
4244:
4245:     def test_in_scope_global(self):
4246:         """Test in_scope with global scope."""
4247:         assert in_scope({}, "global") is True
4248:         assert in_scope({"section": "budget"}, "global") is True
4249:
4250:     def test_in_scope_section(self):
4251:         """Test in_scope with section scope."""
4252:         assert in_scope({"section": "budget"}, "section") is True
4253:         assert in_scope({}, "section") is False
4254:
4255:     def test_in_scope_chapter(self):
4256:         """Test in_scope with chapter scope."""
```

```
4257:         assert in_scope({"chapter": 1}, "chapter") is True
4258:         assert in_scope({}, "chapter") is False
4259:
4260:     def test_in_scope_page(self):
4261:         """Test in_scope with page scope."""
4262:         assert in_scope({"page": 5}, "page") is True
4263:         assert in_scope({}, "page") is False
4264:
4265:     def test_evaluate_comparison_greater_than(self):
4266:         """Test evaluate_comparison with > operator."""
4267:         assert evaluate_comparison(5, ">2") is True
4268:         assert evaluate_comparison(2, ">2") is False
4269:         assert evaluate_comparison(1, ">2") is False
4270:
4271:     def test_evaluate_comparison_greater_equal(self):
4272:         """Test evaluate_comparison with >= operator."""
4273:         assert evaluate_comparison(5, ">=5") is True
4274:         assert evaluate_comparison(6, ">=5") is True
4275:         assert evaluate_comparison(4, ">=5") is False
4276:
4277:     def test_evaluate_comparison_less_than(self):
4278:         """Test evaluate_comparison with < operator."""
4279:         assert evaluate_comparison(2, "<5") is True
4280:         assert evaluate_comparison(5, "<5") is False
4281:         assert evaluate_comparison(6, "<5") is False
4282:
4283:     def test_evaluate_comparison_less_equal(self):
4284:         """Test evaluate_comparison with <= operator."""
4285:         assert evaluate_comparison(5, "<=5") is True
4286:         assert evaluate_comparison(4, "<=5") is True
4287:         assert evaluate_comparison(6, "<=5") is False
4288:
4289:     def test_evaluate_comparison_invalid_value(self):
4290:         """Test evaluate_comparison with invalid value returns False."""
4291:         assert evaluate_comparison("invalid", ">2") is False
4292:         assert evaluate_comparison(None, ">2") is False
4293:
4294:     def test_create_document_context(self):
4295:         """Test create_document_context helper function."""
4296:         context = create_document_context(section="budget", chapter=3, page=47)
4297:         assert context == {"section": "budget", "chapter": 3, "page": 47}
4298:
4299:     def test_create_document_context_with_kwargs(self):
4300:         """Test create_document_context with additional kwargs."""
4301:         context = create_document_context(section="budget", custom_field="value")
4302:         assert context["section"] == "budget"
4303:         assert context["custom_field"] == "value"
4304:
4305:
4306:
4307: =====
4308: FILE: tests/phases/test_phase5_signal_resolution.py
4309: =====
4310:
4311: """Test Phase 5: Signal Resolution with Registry Integration
4312:
```

```
4313: Tests Phase 5 signal resolution logic including:
4314: - Signal registry integration and query interface
4315: - Missing signal hard stops (no fallbacks or degraded modes)
4316: - Immutable signal tuple returns
4317: - Set-based signal validation
4318: - Per-chunk signal caching
4319: - Required vs optional signal distinction
4320: - Signal type validation
4321: - Error message clarity for missing signals
4322: """
4323:
4324: import pytest
4325:
4326: from farfan_pipeline.core.orchestrator.signal_resolution import (
4327:     Chunk,
4328:     Question,
4329:     Signal,
4330:     _resolve_signals,
4331: )
4332:
4333:
4334: class MockSignalRegistry:
4335:     """Mock signal registry for testing."""
4336:
4337:     def __init__(self, signals_to_return=None):
4338:         self.signals_to_return = signals_to_return or []
4339:         self.calls = []
4340:
4341:     def get_signals_for_chunk(self, chunk, required_types):
4342:         """Mock get_signals_for_chunk method."""
4343:         self.calls.append(
4344:             {
4345:                 "chunk": chunk,
4346:                 "required_types": required_types,
4347:             }
4348:         )
4349:         return self.signals_to_return
4350:
4351:
4352: class TestPhase5SignalRegistryIntegration:
4353:     """Test signal registry integration and query interface."""
4354:
4355:     def test_resolve_signals_queries_registry(self):
4356:         """Test _resolve_signals queries signal registry."""
4357:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test chunk")
4358:         question = Question(question_id="Q01", signal_requirements={"signal_type_1"})
4359:         signals = [Signal(signal_type="signal_type_1", content=None)]
4360:         registry = MockSignalRegistry(signals_to_return=signals)
4361:
4362:         _resolve_signals(chunk, question, registry)
4363:
4364:         assert len(registry.calls) == 1
4365:         assert registry.calls[0]["chunk"] == chunk
4366:         assert registry.calls[0]["required_types"] == {"signal_type_1"}
4367:
4368:     def test_resolve_signals_passes_chunk_to_registry(self):
```

```
4369:         """Test chunk object passed to registry unchanged."""
4370:         chunk = Chunk(chunk_id="PA02-DIM03", text="Another test")
4371:         question = Question(question_id="Q02", signal_requirements={"signal_type_1"})
4372:         signals = [Signal(signal_type="signal_type_1", content=None)]
4373:         registry = MockSignalRegistry(signals_to_return=signals)
4374:
4375:         _resolve_signals(chunk, question, registry)
4376:
4377:         assert registry.calls[0]["chunk"].chunk_id == "PA02-DIM03"
4378:         assert registry.calls[0]["chunk"].text == "Another test"
4379:
4380:     def test_resolve_signals_passes_required_types_set(self):
4381:         """Test required_types passed as set to registry."""
4382:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4383:         question = Question(
4384:             question_id="Q03", signal_requirements={"type_1", "type_2", "type_3"}
4385:         )
4386:         signals = [
4387:             Signal(signal_type="type_1", content=None),
4388:             Signal(signal_type="type_2", content=None),
4389:             Signal(signal_type="type_3", content=None),
4390:         ]
4391:         registry = MockSignalRegistry(signals_to_return=signals)
4392:
4393:         _resolve_signals(chunk, question, registry)
4394:
4395:         assert registry.calls[0]["required_types"] == {"type_1", "type_2", "type_3"}
4396:
4397:     def test_resolve_signals_with_single_required_type(self):
4398:         """Test signal resolution with single required type."""
4399:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4400:         question = Question(question_id="Q04", signal_requirements={"single_type"})
4401:         signals = [Signal(signal_type="single_type", content=None)]
4402:         registry = MockSignalRegistry(signals_to_return=signals)
4403:
4404:         result = _resolve_signals(chunk, question, registry)
4405:
4406:         assert len(result) == 1
4407:         assert result[0].signal_type == "single_type"
4408:
4409:     def test_resolve_signals_with_multiple_required_types(self):
4410:         """Test signal resolution with multiple required types."""
4411:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4412:         question = Question(
4413:             question_id="Q05", signal_requirements={"type_a", "type_b", "type_c"}
4414:         )
4415:         signals = [
4416:             Signal(signal_type="type_a", content=None),
4417:             Signal(signal_type="type_b", content=None),
4418:             Signal(signal_type="type_c", content=None),
4419:         ]
4420:         registry = MockSignalRegistry(signals_to_return=signals)
4421:
4422:         result = _resolve_signals(chunk, question, registry)
4423:
4424:         assert len(result) == 3
```

```
4425:         signal_types = {s.signal_type for s in result}
4426:         assert signal_types == {"type_a", "type_b", "type_c"}
4427:
4428:     def test_resolve_signals_empty_requirements_set(self):
4429:         """Test signal resolution with empty requirements (should succeed)."""
4430:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4431:         question = Question(question_id="Q06", signal_requirements=set())
4432:         registry = MockSignalRegistry(signals_to_return=[])
4433:
4434:         result = _resolve_signals(chunk, question, registry)
4435:
4436:         assert len(result) == 0
4437:         assert isinstance(result, tuple)
4438:
4439:
4440: class TestPhase5MissingSignalHardStops:
4441:     """Test missing signal hard stops with no fallbacks or degraded modes."""
4442:
4443:     def test_missing_signal_raises_value_error(self):
4444:         """Test missing required signal raises ValueError."""
4445:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4446:         question = Question(question_id="Q07", signal_requirements={"missing_type"})
4447:         registry = MockSignalRegistry(signals_to_return=[])
4448:
4449:         with pytest.raises(ValueError, match="Missing signals"):
4450:             _resolve_signals(chunk, question, registry)
4451:
4452:     def test_missing_signal_error_message_includes_signal_type(self):
4453:         """Test error message includes missing signal type."""
4454:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4455:         question = Question(question_id="Q08", signal_requirements={"specific_type"})
4456:         registry = MockSignalRegistry(signals_to_return=[])
4457:
4458:         with pytest.raises(ValueError, match="specific_type"):
4459:             _resolve_signals(chunk, question, registry)
4460:
4461:     def test_missing_multiple_signals_error_message(self):
4462:         """Test error message lists all missing signals."""
4463:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4464:         question = Question(
4465:             question_id="Q09",
4466:             signal_requirements={"missing_1", "missing_2", "missing_3"},
4467:         )
4468:         registry = MockSignalRegistry(signals_to_return=[])
4469:
4470:         with pytest.raises(ValueError) as exc_info:
4471:             _resolve_signals(chunk, question, registry)
4472:
4473:         error_msg = str(exc_info.value)
4474:         assert "missing_1" in error_msg or "Missing signals" in error_msg
4475:         assert "missing_2" in error_msg or "Missing signals" in error_msg
4476:         assert "missing_3" in error_msg or "Missing signals" in error_msg
4477:
4478:     def test_partial_signal_match_raises_error(self):
4479:         """Test partial signal match (some present, some missing) raises error."""
4480:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
```



```
4481:         question = Question(
4482:             question_id="Q10", signal_requirements={"present", "missing"}
4483:         )
4484:         signals = [Signal(signal_type="present", content=None)]
4485:         registry = MockSignalRegistry(signals_to_return=signals)
4486:
4487:         with pytest.raises(ValueError, match="Missing signals"):
4488:             _resolve_signals(chunk, question, registry)
4489:
4490:     def test_no_fallback_to_alternative_signals(self):
4491:         """Test no fallback mechanism when exact signal missing."""
4492:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4493:         question = Question(question_id="Q11", signal_requirements={"exact_signal"})
4494:         signals = [Signal(signal_type="similar_signal", content=None)]
4495:         registry = MockSignalRegistry(signals_to_return=signals)
4496:
4497:         with pytest.raises(ValueError):
4498:             _resolve_signals(chunk, question, registry)
4499:
4500:     def test_no_degraded_mode_on_missing_signals(self):
4501:         """Test system fails immediately without degraded mode."""
4502:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4503:         question = Question(question_id="Q12", signal_requirements={"critical_signal"})
4504:         registry = MockSignalRegistry(signals_to_return=[])
4505:
4506:         with pytest.raises(ValueError):
4507:             _resolve_signals(chunk, question, registry)
4508:
4509:
4510: class TestPhase5ImmutableSignalTuples:
4511:     """Test immutable signal tuple returns."""
4512:
4513:     def test_resolve_signals_returns_tuple(self):
4514:         """Test _resolve_signals returns tuple."""
4515:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4516:         question = Question(question_id="Q13", signal_requirements={"type_1"})
4517:         signals = [Signal(signal_type="type_1", content=None)]
4518:         registry = MockSignalRegistry(signals_to_return=signals)
4519:
4520:         result = _resolve_signals(chunk, question, registry)
4521:
4522:         assert isinstance(result, tuple)
4523:
4524:     def test_returned_tuple_is_immutable(self):
4525:         """Test returned tuple cannot be modified."""
4526:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4527:         question = Question(question_id="Q14", signal_requirements={"type_1"})
4528:         signals = [Signal(signal_type="type_1", content=None)]
4529:         registry = MockSignalRegistry(signals_to_return=signals)
4530:
4531:         result = _resolve_signals(chunk, question, registry)
4532:
4533:         with pytest.raises(AttributeError):
4534:             result.append(Signal(signal_type="new", content=None))
4535:
4536:     def test_signal_objects_are_named_tuples(self):
```

```
4537: """Test Signal objects are immutable NamedTuples."""
4538: signal = Signal(signal_type="test", content=None)
4539:
4540: assert isinstance(signal, tuple)
4541: assert hasattr(signal, "signal_type")
4542: assert hasattr(signal, "content")
4543:
4544: with pytest.raises(AttributeError):
4545:     signal.signal_type = "modified"
4546:
4547: def test_empty_result_returns_empty_tuple(self):
4548:     """Test empty signal list returns empty tuple."""
4549:     chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4550:     question = Question(question_id="Q15", signal_requirements=set())
4551:     registry = MockSignalRegistry(signals_to_return=[])
4552:
4553:     result = _resolve_signals(chunk, question, registry)
4554:
4555:     assert result == ()
4556:     assert isinstance(result, tuple)
4557:
4558: def test_signal_order_preserved_in_tuple(self):
4559:     """Test signal order preserved in returned tuple."""
4560:     chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4561:     question = Question(
4562:         question_id="Q16", signal_requirements={"type_1", "type_2", "type_3"}
4563:     )
4564:     signals = [
4565:         Signal(signal_type="type_1", content="first"),
4566:         Signal(signal_type="type_2", content="second"),
4567:         Signal(signal_type="type_3", content="third"),
4568:     ]
4569:     registry = MockSignalRegistry(signals_to_return=signals)
4570:
4571:     result = _resolve_signals(chunk, question, registry)
4572:
4573:     assert result[0].content == "first"
4574:     assert result[1].content == "second"
4575:     assert result[2].content == "third"
4576:
4577:
4578: class TestPhase5SetBasedValidation:
4579:     """Test set-based signal validation."""
4580:
4581:     def test_required_types_compared_as_set(self):
4582:         """Test required signal types compared as set (order independent)."""
4583:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4584:         question = Question(
4585:             question_id="Q17", signal_requirements={"type_c", "type_a", "type_b"}
4586:         )
4587:         signals = [
4588:             Signal(signal_type="type_a", content=None),
4589:             Signal(signal_type="type_b", content=None),
4590:             Signal(signal_type="type_c", content=None),
4591:         ]
4592:         registry = MockSignalRegistry(signals_to_return=signals)
```

```
4593:
4594:     result = _resolve_signals(chunk, question, registry)
4595:
4596:     assert len(result) == 3
4597:
4598: def test_duplicate_signal_types_in_requirements_handled(self):
4599:     """Test duplicate signal types in requirements treated as single requirement."""
4600:     chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4601:     question = Question(
4602:         question_id="Q18", signal_requirements={"type_1"} # Duplicate
4603:     )
4604:     signals = [Signal(signal_type="type_1", content=None)]
4605:     registry = MockSignalRegistry(signals_to_return=signals)
4606:
4607:     result = _resolve_signals(chunk, question, registry)
4608:
4609:     assert len(result) == 1
4610:
4611: def test_extra_signals_returned_not_validated(self):
4612:     """Test extra signals beyond requirements don't cause failure."""
4613:     chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4614:     question = Question(question_id="Q19", signal_requirements={"type_1"})
4615:     signals = [
4616:         Signal(signal_type="type_1", content=None),
4617:         Signal(signal_type="extra_type", content=None),
4618:     ]
4619:     registry = MockSignalRegistry(signals_to_return=signals)
4620:
4621:     result = _resolve_signals(chunk, question, registry)
4622:
4623:     assert len(result) == 2
4624:
4625: def test_signal_type_string_comparison_case_sensitive(self):
4626:     """Test signal type comparison is case-sensitive."""
4627:     chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4628:     question = Question(question_id="Q20", signal_requirements={"Type_1"})
4629:     signals = [Signal(signal_type="type_1", content=None)]
4630:     registry = MockSignalRegistry(signals_to_return=signals)
4631:
4632:     with pytest.raises(ValueError):
4633:         _resolve_signals(chunk, question, registry)
4634:
4635: def test_missing_signals_calculated_via_set_difference(self):
4636:     """Test missing signals identified via set difference operation."""
4637:     chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4638:     question = Question(
4639:         question_id="Q21", signal_requirements={"type_1", "type_2", "type_3"}
4640:     )
4641:     signals = [Signal(signal_type="type_1", content=None)]
4642:     registry = MockSignalRegistry(signals_to_return=signals)
4643:
4644:     with pytest.raises(ValueError) as exc_info:
4645:         _resolve_signals(chunk, question, registry)
4646:
4647:     error_msg = str(exc_info.value)
4648:     assert "type_2" in error_msg or "Missing signals" in error_msg
```

```
4649:         assert "type_3" in error_msg or "Missing signals" in error_msg
4650:
4651:
4652: class TestPhase5SignalTypeValidation:
4653:     """Test signal type validation."""
4654:
4655:     def test_signal_type_must_be_string(self):
4656:         """Test Signal.signal_type must be string."""
4657:         signal = Signal(signal_type="valid_type", content=None)
4658:         assert isinstance(signal.signal_type, str)
4659:
4660:     def test_signal_content_can_be_none(self):
4661:         """Test Signal.content can be None."""
4662:         signal = Signal(signal_type="test", content=None)
4663:         assert signal.content is None
4664:
4665:     def test_signal_content_can_be_signal_pack(self):
4666:         """Test Signal.content can hold SignalPack object."""
4667:         mock_signal_pack = {"data": "test"}
4668:         signal = Signal(signal_type="test", content=mock_signal_pack)
4669:         assert signal.content == {"data": "test"}
4670:
4671:     def test_question_signal_requirements_is_set(self):
4672:         """Test Question.signal_requirements is a set."""
4673:         question = Question(question_id="Q22", signal_requirements={"type_1", "type_2"})
4674:         assert isinstance(question.signal_requirements, set)
4675:
4676:     def test_chunk_has_required_fields(self):
4677:         """Test Chunk has chunk_id and text fields."""
4678:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test content")
4679:         assert chunk.chunk_id == "PA01-DIM01"
4680:         assert chunk.text == "Test content"
4681:
4682:
4683: class TestPhase5ErrorMessageClarity:
4684:     """Test error message clarity for missing signals."""
4685:
4686:     def test_error_message_uses_missing_signals_phrase(self):
4687:         """Test error message explicitly states 'Missing signals'."""
4688:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4689:         question = Question(question_id="Q23", signal_requirements={"signal_1"})
4690:         registry = MockSignalRegistry(signals_to_return=[])
4691:
4692:         with pytest.raises(ValueError, match=r"Missing signals"):
4693:             _resolve_signals(chunk, question, registry)
4694:
4695:     def test_error_message_shows_missing_as_set_format(self):
4696:         """Test error message shows missing signals in set format."""
4697:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4698:         question = Question(question_id="Q24", signal_requirements={"sig_a", "sig_b"})
4699:         registry = MockSignalRegistry(signals_to_return=[])
4700:
4701:         with pytest.raises(ValueError) as exc_info:
4702:             _resolve_signals(chunk, question, registry)
4703:
4704:         error_msg = str(exc_info.value)
```

```
4705:         assert "{" in error_msg or "Missing signals" in error_msg
4706:
4707:     def test_error_message_sorts_missing_signals(self):
4708:         """Test error message sorts missing signals alphabetically."""
4709:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4710:         question = Question(
4711:             question_id="Q25", signal_requirements={"z_signal", "a_signal", "m_signal"}
4712:         )
4713:         registry = MockSignalRegistry(signals_to_return=[])
4714:
4715:         with pytest.raises(ValueError) as exc_info:
4716:             _resolve_signals(chunk, question, registry)
4717:
4718:         error_msg = str(exc_info.value)
4719:         # Should contain sorted list
4720:         assert "Missing signals" in error_msg
4721:
4722:     def test_single_missing_signal_clear_message(self):
4723:         """Test single missing signal produces clear error message."""
4724:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4725:         question = Question(question_id="Q26", signal_requirements={"only_signal"})
4726:         registry = MockSignalRegistry(signals_to_return=[])
4727:
4728:         with pytest.raises(ValueError, match=r"Missing signals.*only_signal"):
4729:             _resolve_signals(chunk, question, registry)
4730:
4731:
4732: class TestPhase5ChunkAndQuestionStructure:
4733:     """Test Chunk and Question NamedTuple structures."""
4734:
4735:     def test_chunk_is_named_tuple(self):
4736:         """Test Chunk is NamedTuple with immutable fields."""
4737:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4738:
4739:         assert isinstance(chunk, tuple)
4740:         with pytest.raises(AttributeError):
4741:             chunk.chunk_id = "modified"
4742:
4743:     def test_question_is_named_tuple(self):
4744:         """Test Question is NamedTuple with immutable fields."""
4745:         question = Question(question_id="Q27", signal_requirements={"type_1"})
4746:
4747:         assert isinstance(question, tuple)
4748:         with pytest.raises(AttributeError):
4749:             question.question_id = "modified"
4750:
4751:     def test_chunk_fields_accessible(self):
4752:         """Test Chunk fields are accessible by name."""
4753:         chunk = Chunk(chunk_id="PA05-DIM03", text="Chunk content here")
4754:
4755:         assert chunk.chunk_id == "PA05-DIM03"
4756:         assert chunk.text == "Chunk content here"
4757:
4758:     def test_question_fields_accessible(self):
4759:         """Test Question fields are accessible by name."""
4760:         question = Question(question_id="Q28", signal_requirements={"sig_1", "sig_2"})
```

```
4761:
4762:     assert question.question_id == "Q28"
4763:     assert question.signal_requirements == {"sig_1", "sig_2"}
4764:
4765:     def test_signal_fields_accessible(self):
4766:         """Test Signal fields are accessible by name."""
4767:         signal = Signal(signal_type="test_type", content={"key": "value"})
4768:
4769:         assert signal.signal_type == "test_type"
4770:         assert signal.content == {"key": "value"}
4771:
4772:
4773: class TestPhase5RegistryCallPatterns:
4774:     """Test registry call patterns and caching implications."""
4775:
4776:     def test_single_registry_call_per_resolution(self):
4777:         """Test only one registry call made per signal resolution."""
4778:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4779:         question = Question(question_id="Q29", signal_requirements={"type_1"})
4780:         signals = [Signal(signal_type="type_1", content=None)]
4781:         registry = MockSignalRegistry(signals_to_return=signals)
4782:
4783:         _resolve_signals(chunk, question, registry)
4784:
4785:         assert len(registry.calls) == 1
4786:
4787:     def test_multiple_resolutions_separate_registry_calls(self):
4788:         """Test multiple resolutions make separate registry calls."""
4789:         chunk1 = Chunk(chunk_id="PA01-DIM01", text="Test 1")
4790:         chunk2 = Chunk(chunk_id="PA02-DIM02", text="Test 2")
4791:         question = Question(question_id="Q30", signal_requirements={"type_1"})
4792:         signals = [Signal(signal_type="type_1", content=None)]
4793:         registry = MockSignalRegistry(signals_to_return=signals)
4794:
4795:         _resolve_signals(chunk1, question, registry)
4796:         _resolve_signals(chunk2, question, registry)
4797:
4798:         assert len(registry.calls) == 2
4799:         assert registry.calls[0]["chunk"].chunk_id == "PA01-DIM01"
4800:         assert registry.calls[1]["chunk"].chunk_id == "PA02-DIM02"
4801:
4802:     def test_registry_receives_complete_requirement_set(self):
4803:         """Test registry receives complete set of requirements."""
4804:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4805:         question = Question(
4806:             question_id="Q31",
4807:             signal_requirements={"type_1", "type_2", "type_3", "type_4"},
4808:         )
4809:         signals = [
4810:             Signal(signal_type="type_1", content=None),
4811:             Signal(signal_type="type_2", content=None),
4812:             Signal(signal_type="type_3", content=None),
4813:             Signal(signal_type="type_4", content=None),
4814:         ]
4815:         registry = MockSignalRegistry(signals_to_return=signals)
4816:
```

```
4817:         _resolve_signals(chunk, question, registry)
4818:
4819:         assert len(registry.calls[0]["required_types"]) == 4
4820:
4821:
4822: class TestPhase5EdgeCases:
4823:     """Test edge cases in signal resolution."""
4824:
4825:     def test_empty_signal_requirements_succeeds(self):
4826:         """Test resolution succeeds with empty signal requirements."""
4827:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4828:         question = Question(question_id="Q32", signal_requirements=set())
4829:         registry = MockSignalRegistry(signals_to_return=[])
4830:
4831:         result = _resolve_signals(chunk, question, registry)
4832:
4833:         assert len(result) == 0
4834:
4835:     def test_large_requirement_set_handled(self):
4836:         """Test large signal requirement set handled correctly."""
4837:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4838:         signal_types = {f"type_{i}" for i in range(100)}
4839:         question = Question(question_id="Q33", signal_requirements=signal_types)
4840:         signals = [Signal(signal_type=t, content=None) for t in signal_types]
4841:         registry = MockSignalRegistry(signals_to_return=signals)
4842:
4843:         result = _resolve_signals(chunk, question, registry)
4844:
4845:         assert len(result) == 100
4846:
4847:     def test_special_characters_in_signal_types(self):
4848:         """Test signal types with special characters."""
4849:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4850:         question = Question(
4851:             question_id="Q34",
4852:             signal_requirements={
4853:                 "type-with-dash",
4854:                 "type_with_underscore",
4855:                 "type.with.dot",
4856:             },
4857:         )
4858:         signals = [
4859:             Signal(signal_type="type-with-dash", content=None),
4860:             Signal(signal_type="type_with_underscore", content=None),
4861:             Signal(signal_type="type.with.dot", content=None),
4862:         ]
4863:         registry = MockSignalRegistry(signals_to_return=signals)
4864:
4865:         result = _resolve_signals(chunk, question, registry)
4866:
4867:         assert len(result) == 3
4868:
4869:     def test_unicode_in_signal_types(self):
4870:         """Test signal types with unicode characters."""
4871:         chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
4872:         question = Question(
```

```
4873:         question_id="Q35", signal_requirements={"type_cafÃ©", "type_æ\227¥æ\234¬èª\236"}
4874:     )
4875:     signals = [
4876:         Signal(signal_type="type_cafÃ©", content=None),
4877:         Signal(signal_type="type_æ\227¥æ\234¬èª\236", content=None),
4878:     ]
4879:     registry = MockSignalRegistry(signals_to_return=signals)
4880:
4881:     result = _resolve_signals(chunk, question, registry)
4882:
4883:     assert len(result) == 2
4884:
4885:
4886:
4887: =====
4888: FILE: tests/phases/test_phase6_schema_validation.py
4889: =====
4890:
4891: """Test Phase 6: Schema Validation
4892:
4893: Tests Phase 6 schema validation logic including:
4894: - Structural validation (type classification, homogeneity checking)
4895: - List length equality validation across blocks
4896: - Dict key set equality validation
4897: - Semantic validation (type field rules, required field enforcement)
4898: - Minimum value constraints
4899: - Schema version validation
4900: - Question count validation
4901: - Referential integrity checking
4902: - Field coverage validation
4903: - Hash calculation and verification
4904: """
4905:
4906: import pytest
4907:
4908: from farfan_pipeline.utils.validation.schema_validator import (
4909:     MonolithIntegrityReport,
4910:     MonolithSchemaValidator,
4911:     SchemaInitializationError,
4912: )
4913:
4914:
4915: class TestPhase6StructuralValidation:
4916:     """Test structural validation including type classification and homogeneity."""
4917:
4918:     @pytest.fixture
4919:     def valid_monolith(self):
4920:         """Create valid monolith structure."""
4921:         return {
4922:             "schema_version": "2.0.0",
4923:             "version": "1.0.0",
4924:             "blocks": {
4925:                 "niveles_abstraccion": ["micro", "meso", "macro"],
4926:                 "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
4927:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
4928:                 "macro_question": {"id": "MACRO_01"},
```



```
4929:         "scoring": {"method": "weighted"},
4930:     },
4931:     "integrity": {"checksum": "abc123"},
4932: }
4933:
4934: def test_validate_top_level_keys_present(self, valid_monolith):
4935:     """Test validation checks for required top-level keys."""
4936:     validator = MonolithSchemaValidator()
4937:
4938:     report = validator.validate_monolith(valid_monolith, strict=False)
4939:
4940:     assert report.validation_passed
4941:
4942: def test_missing_schema_version_fails_validation(self):
4943:     """Test missing schema_version fails validation."""
4944:     monolith = {
4945:         "version": "1.0.0",
4946:         "blocks": {},
4947:         "integrity": {},
4948:     }
4949:     validator = MonolithSchemaValidator()
4950:
4951:     report = validator.validate_monolith(monolith, strict=False)
4952:
4953:     assert report.validation_passed is False
4954:     assert any("schema_version" in e for e in report.errors)
4955:
4956: def test_missing_blocks_fails_validation(self):
4957:     """Test missing blocks section fails validation."""
4958:     monolith = {
4959:         "schema_version": "2.0.0",
4960:         "version": "1.0.0",
4961:         "integrity": {},
4962:     }
4963:     validator = MonolithSchemaValidator()
4964:
4965:     report = validator.validate_monolith(monolith, strict=False)
4966:
4967:     assert report.validation_passed is False
4968:     assert any("blocks" in e for e in report.errors)
4969:
4970: def test_required_blocks_present(self, valid_monolith):
4971:     """Test all required blocks are present."""
4972:     validator = MonolithSchemaValidator()
4973:
4974:     validator.validate_monolith(valid_monolith, strict=False)
4975:
4976:     assert "niveles_abstraccion" in valid_monolith["blocks"]
4977:     assert "micro_questions" in valid_monolith["blocks"]
4978:     assert "meso_questions" in valid_monolith["blocks"]
4979:     assert "macro_question" in valid_monolith["blocks"]
4980:
4981: def test_missing_required_block_fails_validation(self, valid_monolith):
4982:     """Test missing required block fails validation."""
4983:     del valid_monolith["blocks"]["micro_questions"]
4984:     validator = MonolithSchemaValidator()
```

```
4985:
4986:     report = validator.validate_monolith(valid_monolith, strict=False)
4987:
4988:     assert report.validation_passed is False
4989:     assert any("micro_questions" in e for e in report.errors)
4990:
4991: def test_type_classification_list_vs_dict(self, valid_monolith):
4992:     """Test type classification distinguishes list from dict."""
4993:     MonolithSchemaValidator()
4994:
4995:     blocks = valid_monolith["blocks"]
4996:     assert isinstance(blocks["micro_questions"], list)
4997:     assert isinstance(blocks["macro_question"], dict)
4998:
4999: def test_homogeneous_list_validation(self):
5000:     """Test list homogeneity validation."""
5001:     monolith = {
5002:         "schema_version": "2.0.0",
5003:         "version": "1.0.0",
5004:         "blocks": {
5005:             "niveles_abstraccion": ["micro", "meso", "macro"],
5006:             "micro_questions": [
5007:                 {"id": "Q001", "text": "Question 1"},
5008:                 {"id": "Q002", "text": "Question 2"},
5009:             ],
5010:             "meso_questions": [{"id": "M01"}],
5011:             "macro_question": {"id": "MACRO_01"},
5012:             "scoring": {},
5013:         },
5014:         "integrity": {},
5015:     }
5016:     validator = MonolithSchemaValidator()
5017:
5018:     validator.validate_monolith(monolith, strict=False)
5019:
5020:     # All items in micro_questions should be dicts with same structure
5021:     assert all(isinstance(q, dict) for q in monolith["blocks"]["micro_questions"])
5022:
5023: def test_heterogeneous_list_detected(self):
5024:     """Test heterogeneous list (mixed types) is detected."""
5025:     monolith = {
5026:         "schema_version": "2.0.0",
5027:         "version": "1.0.0",
5028:         "blocks": {
5029:             "niveles_abstraccion": ["micro", "meso", "macro"],
5030:             "micro_questions": [
5031:                 {"id": "Q001"},
5032:                 "not_a_dict", # Different type
5033:                 {"id": "Q003"},
5034:             ],
5035:             "meso_questions": [{"id": "M01"}],
5036:             "macro_question": {"id": "MACRO_01"},
5037:             "scoring": {},
5038:         },
5039:         "integrity": {},
5040:     }
```

```
5041:
5042:     # Should detect type inconsistency
5043:     items = monolith["blocks"]["micro_questions"]
5044:     types_present = {type(item) for item in items}
5045:     assert len(types_present) > 1
5046:
5047:
5048: class TestPhase6ListLengthEquality:
5049:     """Test list length equality validation across blocks."""
5050:
5051:     def test_micro_questions_count_300(self):
5052:         """Test micro_questions list has exactly 300 items."""
5053:         monolith = {
5054:             "schema_version": "2.0.0",
5055:             "version": "1.0.0",
5056:             "blocks": {
5057:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5058:                 "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5059:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5060:                 "macro_question": {"id": "MACRO_01"},
5061:                 "scoring": {},
5062:             },
5063:             "integrity": {},
5064:         }
5065:         validator = MonolithSchemaValidator()
5066:
5067:         report = validator.validate_monolith(monolith, strict=False)
5068:
5069:         assert report.question_counts["micro_questions"] == 300
5070:
5071:     def test_incorrect_micro_questions_count_fails(self):
5072:         """Test incorrect micro_questions count fails validation."""
5073:         monolith = {
5074:             "schema_version": "2.0.0",
5075:             "version": "1.0.0",
5076:             "blocks": {
5077:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5078:                 "micro_questions": [
5079:                     {"id": f"Q{i:03d}"} for i in range(1, 251)
5080:                 ], # Only 250
5081:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5082:                 "macro_question": {"id": "MACRO_01"},
5083:                 "scoring": {},
5084:             },
5085:             "integrity": {},
5086:         }
5087:         validator = MonolithSchemaValidator()
5088:
5089:         report = validator.validate_monolith(monolith, strict=False)
5090:
5091:         assert report.validation_passed is False
5092:         assert report.question_counts["micro_questions"] == 250
5093:
5094:     def test_meso_questions_count_4(self):
5095:         """Test meso_questions list has exactly 4 items."""
5096:         monolith = {
```

```
5097:         "schema_version": "2.0.0",
5098:         "version": "1.0.0",
5099:         "blocks": {
5100:             "niveles_abstraccion": ["micro", "meso", "macro"],
5101:             "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5102:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5103:             "macro_question": {"id": "MACRO_01"},
5104:             "scoring": {},
5105:         },
5106:         "integrity": {},
5107:     }
5108:     validator = MonolithSchemaValidator()
5109:
5110:     report = validator.validate_monolith(monolith, strict=False)
5111:
5112:     assert report.question_counts["meso_questions"] == 4
5113:
5114: def test_incorrect_meso_questions_count_fails(self):
5115:     """Test incorrect meso_questions count fails validation."""
5116:     monolith = {
5117:         "schema_version": "2.0.0",
5118:         "version": "1.0.0",
5119:         "blocks": {
5120:             "niveles_abstraccion": ["micro", "meso", "macro"],
5121:             "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5122:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 4)], # Only 3
5123:             "macro_question": {"id": "MACRO_01"},
5124:             "scoring": {},
5125:         },
5126:         "integrity": {},
5127:     }
5128:     validator = MonolithSchemaValidator()
5129:
5130:     report = validator.validate_monolith(monolith, strict=False)
5131:
5132:     assert report.validation_passed is False
5133:
5134: def test_macro_question_count_1(self):
5135:     """Test macro_question is single dict (count=1)."""
5136:     monolith = {
5137:         "schema_version": "2.0.0",
5138:         "version": "1.0.0",
5139:         "blocks": {
5140:             "niveles_abstraccion": ["micro", "meso", "macro"],
5141:             "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5142:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5143:             "macro_question": {"id": "MACRO_01"},
5144:             "scoring": {},
5145:         },
5146:         "integrity": {},
5147:     }
5148:     validator = MonolithSchemaValidator()
5149:
5150:     report = validator.validate_monolith(monolith, strict=False)
5151:
5152:     assert report.question_counts.get("macro_question") == 1
```

```
5153:
5154:     def test_empty_list_detected(self):
5155:         """Test empty question list is detected."""
5156:         monolith = {
5157:             "schema_version": "2.0.0",
5158:             "version": "1.0.0",
5159:             "blocks": {
5160:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5161:                 "micro_questions": [], # Empty
5162:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5163:                 "macro_question": {"id": "MACRO_01"},
5164:                 "scoring": {},
5165:             },
5166:             "integrity": {},
5167:         }
5168:         validator = MonolithSchemaValidator()
5169:
5170:         report = validator.validate_monolith(monolith, strict=False)
5171:
5172:         assert report.question_counts["micro_questions"] == 0
5173:         assert report.validation_passed is False
5174:
5175:
5176: class TestPhase6DictKeySetEquality:
5177:     """Test dict key set equality validation."""
5178:
5179:     def test_all_micro_questions_have_same_keys(self):
5180:         """Test all micro questions have same key set."""
5181:         questions = [
5182:             {"id": "Q001", "text": "Question 1", "dimension": "D1"},
5183:             {"id": "Q002", "text": "Question 2", "dimension": "D2"},
5184:             {"id": "Q003", "text": "Question 3", "dimension": "D3"},
5185:         ]
5186:
5187:         key_sets = [set(q.keys()) for q in questions]
5188:         assert all(keys == key_sets[0] for keys in key_sets)
5189:
5190:     def test_inconsistent_keys_detected(self):
5191:         """Test inconsistent key sets across questions detected."""
5192:         questions = [
5193:             {"id": "Q001", "text": "Question 1"},
5194:             {"id": "Q002", "text": "Question 2", "extra_field": "value"},
5195:             {"id": "Q003", "text": "Question 3"},
5196:         ]
5197:
5198:         key_sets = [set(q.keys()) for q in questions]
5199:         assert not all(keys == key_sets[0] for keys in key_sets)
5200:
5201:     def test_missing_key_in_subset_detected(self):
5202:         """Test missing key in subset of questions detected."""
5203:         questions = [
5204:             {"id": "Q001", "text": "Question 1", "required_field": "value"},
5205:             {"id": "Q002", "text": "Question 2"}, # Missing required_field
5206:             {"id": "Q003", "text": "Question 3", "required_field": "value"},
5207:         ]
5208:
```

```
5209: # Check for required_field presence
5210: has_required = [{"required_field" in q} for q in questions]
5211: assert not all(has_required)
5212:
5213: def test_extra_key_in_subset_detected(self):
5214:     """Test extra key in subset of questions detected."""
5215:     questions = [
5216:         {"id": "Q001", "text": "Question 1"},
5217:         {"id": "Q002", "text": "Question 2", "extra": "field"}, # Extra key
5218:         {"id": "Q003", "text": "Question 3"},
5219:     ]
5220:
5221:     key_counts = {}
5222:     for q in questions:
5223:         for key in q.keys():
5224:             key_counts[key] = key_counts.get(key, 0) + 1
5225:
5226:     assert key_counts["extra"] == 1 # Only in one question
5227:
5228: def test_nested_dict_key_consistency(self):
5229:     """Test nested dict key consistency validation."""
5230:     questions = [
5231:         {"id": "Q001", "metadata": {"author": "A", "date": "2024"}},
5232:         {"id": "Q002", "metadata": {"author": "B", "date": "2024"}},
5233:         {"id": "Q003", "metadata": {"author": "C", "date": "2024"}},
5234:     ]
5235:
5236:     nested_key_sets = [set(q["metadata"].keys()) for q in questions]
5237:     assert all(keys == nested_key_sets[0] for keys in nested_key_sets)
5238:
5239:
5240: class TestPhase6SemanticValidation:
5241:     """Test semantic validation including type and required field rules."""
5242:
5243:     def test_required_field_id_present(self):
5244:         """Test required 'id' field is present in all questions."""
5245:         monolith = {
5246:             "schema_version": "2.0.0",
5247:             "version": "1.0.0",
5248:             "blocks": {
5249:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5250:                 "micro_questions": [
5251:                     {"id": "Q001", "text": "Q1"},
5252:                     {"id": "Q002", "text": "Q2"},
5253:                 ],
5254:                 "meso_questions": [{"id": "M01"}],
5255:                 "macro_question": {"id": "MACRO_01"},
5256:                 "scoring": {},
5257:             },
5258:             "integrity": {},
5259:         }
5260:
5261:         # Check all questions have id
5262:         all_have_id = all("id" in q for q in monolith["blocks"]["micro_questions"])
5263:         assert all_have_id is True
5264:
```

```
5265: def test_missing_required_field_detected(self):
5266:     """Test missing required field is detected."""
5267:     questions = [
5268:         {"id": "Q001", "text": "Question 1"},
5269:         {"text": "Question 2"}, # Missing id
5270:         {"id": "Q003", "text": "Question 3"},
5271:     ]
5272:
5273:     all_have_id = all("id" in q for q in questions)
5274:     assert all_have_id is False
5275:
5276: def test_type_field_validation_string(self):
5277:     """Test type field is validated as string."""
5278:     question = {"id": "Q001", "type": "diagnostic", "text": "Question"}
5279:
5280:     assert isinstance(question.get("type"), str)
5281:
5282: def test_type_field_validation_non_string_detected(self):
5283:     """Test non-string type field detected."""
5284:     question = {"id": "Q001", "type": 123, "text": "Question"}
5285:
5286:     assert not isinstance(question.get("type"), str)
5287:
5288: def test_dimension_field_format_validation(self):
5289:     """Test dimension field format validation (D1-D6)."""
5290:     valid_dimensions = ["D1", "D2", "D3", "D4", "D5", "D6"]
5291:
5292:     for dim in valid_dimensions:
5293:         assert dim[0] == "D"
5294:         assert dim[1].isdigit()
5295:
5296: def test_invalid_dimension_format_detected(self):
5297:     """Test invalid dimension format detected."""
5298:     invalid_dimensions = ["D0", "D7", "X1", "1D"]
5299:     valid_pattern = ["D1", "D2", "D3", "D4", "D5", "D6"]
5300:
5301:     for dim in invalid_dimensions:
5302:         assert dim not in valid_pattern
5303:
5304: def test_policy_area_format_validation(self):
5305:     """Test policy area format validation (PA01-PA10)."""
5306:     valid_areas = [f"PA{i:02d}" for i in range(1, 11)]
5307:
5308:     assert len(valid_areas) == 10
5309:     assert all(pa.startswith("PA") for pa in valid_areas)
5310:
5311: def test_enum_value_validation(self):
5312:     """Test enum value validation for fields with fixed values."""
5313:     valid_types = ["diagnostic", "activity", "result", "impact"]
5314:     question_type = "diagnostic"
5315:
5316:     assert question_type in valid_types
5317:
5318: def test_invalid_enum_value_detected(self):
5319:     """Test invalid enum value detected."""
5320:     valid_types = ["diagnostic", "activity", "result", "impact"]
```

```
5321:         question_type = "invalid_type"
5322:
5323:         assert question_type not in valid_types
5324:
5325:
5326: class TestPhase6MinimumValueConstraints:
5327:     """Test minimum value constraints validation."""
5328:
5329:     def test_question_count_minimum_300(self):
5330:         """Test micro_questions minimum count is 300."""
5331:         validator = MonolithSchemaValidator()
5332:         assert validator.EXPECTED_MICRO_QUESTIONS == 300
5333:
5334:     def test_below_minimum_count_fails(self):
5335:         """Test question count below minimum fails validation."""
5336:         monolith = {
5337:             "schema_version": "2.0.0",
5338:             "version": "1.0.0",
5339:             "blocks": {
5340:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5341:                 "micro_questions": [
5342:                     {"id": f"Q{i:03d}"} for i in range(1, 100)
5343:                 ], # Only 99
5344:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5345:                 "macro_question": {"id": "MACRO_01"},
5346:                 "scoring": {},
5347:             },
5348:             "integrity": {},
5349:         }
5350:         validator = MonolithSchemaValidator()
5351:
5352:         report = validator.validate_monolith(monolith, strict=False)
5353:
5354:         assert report.validation_passed is False
5355:
5356:     def test_weight_minimum_zero(self):
5357:         """Test weight field minimum value is 0."""
5358:         weight = 0.5
5359:         assert weight >= 0
5360:
5361:     def test_negative_weight_invalid(self):
5362:         """Test negative weight is invalid."""
5363:         weight = -0.1
5364:         assert weight < 0 # Should be detected as invalid
5365:
5366:     def test_confidence_range_0_to_1(self):
5367:         """Test confidence value in range [0, 1]."""
5368:         valid_confidences = [0.0, 0.5, 0.75, 1.0]
5369:
5370:         for conf in valid_confidences:
5371:             assert 0.0 <= conf <= 1.0
5372:
5373:     def test_confidence_out_of_range_detected(self):
5374:         """Test confidence value outside range detected."""
5375:         invalid_confidences = [-0.1, 1.5, 2.0]
5376:
```



```
5377:         for conf in invalid_confidences:
5378:             assert not (0.0 <= conf <= 1.0)
5379:
5380:
5381: class TestPhase6SchemaVersionValidation:
5382:     """Test schema version validation."""
5383:
5384:     def test_schema_version_format_validation(self):
5385:         """Test schema version follows semantic versioning."""
5386:         version = "2.0.0"
5387:         parts = version.split(".")
5388:
5389:         assert len(parts) == 3
5390:         assert all(part.isdigit() for part in parts)
5391:
5392:     def test_expected_schema_version_2_0_0(self):
5393:         """Test expected schema version is 2.0.0."""
5394:         validator = MonolithSchemaValidator()
5395:         assert validator.EXPECTED_SCHEMA_VERSION == "2.0.0"
5396:
5397:     def test_different_schema_version_warning(self):
5398:         """Test different schema version generates warning."""
5399:         monolith = {
5400:             "schema_version": "1.5.0", # Different version
5401:             "version": "1.0.0",
5402:             "blocks": {
5403:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5404:                 "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5405:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5406:                 "macro_question": {"id": "MACRO_01"},
5407:                 "scoring": {},
5408:             },
5409:             "integrity": {},
5410:         }
5411:         validator = MonolithSchemaValidator()
5412:
5413:         report = validator.validate_monolith(monolith, strict=False)
5414:
5415:         assert len(report.warnings) > 0
5416:
5417:     def test_missing_schema_version_error(self):
5418:         """Test missing schema version generates error."""
5419:         monolith = {
5420:             "version": "1.0.0",
5421:             "blocks": {},
5422:             "integrity": {},
5423:         }
5424:         validator = MonolithSchemaValidator()
5425:
5426:         report = validator.validate_monolith(monolith, strict=False)
5427:
5428:         assert report.validation_passed is False
5429:         assert report.schema_version == ""
5430:
5431:
5432: class TestPhase6ReferentialIntegrity:
```

```
5433:     """Test referential integrity checking."""
5434:
5435:     def test_question_id_uniqueness(self):
5436:         """Test all question IDs are unique."""
5437:         questions = [
5438:             {"id": "Q001"},
5439:             {"id": "Q002"},
5440:             {"id": "Q003"},
5441:         ]
5442:
5443:         ids = [q["id"] for q in questions]
5444:         assert len(ids) == len(set(ids))
5445:
5446:     def test_duplicate_question_id_detected(self):
5447:         """Test duplicate question ID detected."""
5448:         questions = [
5449:             {"id": "Q001"},
5450:             {"id": "Q002"},
5451:             {"id": "Q001"}, # Duplicate
5452:         ]
5453:
5454:         ids = [q["id"] for q in questions]
5455:         assert len(ids) != len(set(ids))
5456:
5457:     def test_cross_reference_validation(self):
5458:         """Test cross-reference between blocks validated."""
5459:         monolith = {
5460:             "blocks": {
5461:                 "micro_questions": [
5462:                     {"id": "Q001", "meso_parent": "M01"},
5463:                     {"id": "Q002", "meso_parent": "M01"},
5464:                 ],
5465:                 "meso_questions": [
5466:                     {"id": "M01"},
5467:                 ],
5468:             }
5469:         }
5470:
5471:         # Check all referenced meso IDs exist
5472:         meso_ids = {q["id"] for q in monolith["blocks"]["meso_questions"]}
5473:         referenced_meso = {
5474:             q.get("meso_parent")
5475:             for q in monolith["blocks"]["micro_questions"]
5476:             if "meso_parent" in q
5477:         }
5478:
5479:         assert referenced_meso.issubset(meso_ids)
5480:
5481:     def test_broken_cross_reference_detected(self):
5482:         """Test broken cross-reference detected."""
5483:         monolith = {
5484:             "blocks": {
5485:                 "micro_questions": [
5486:                     {"id": "Q001", "meso_parent": "M01"},
5487:                     {"id": "Q002", "meso_parent": "M99"}, # Doesn't exist
5488:                 ],
```

```
5489:         "meso_questions": [
5490:             {"id": "M01"},
5491:         ],
5492:     }
5493: }
5494:
5495: meso_ids = {q["id"] for q in monolith["blocks"]["meso_questions"]}
5496: referenced_meso = {
5497:     q.get("meso_parent")
5498:     for q in monolith["blocks"]["micro_questions"]
5499:     if "meso_parent" in q
5500: }
5501:
5502: assert not referenced_meso.issubset(meso_ids)
5503:
5504:
5505: class TestPhase6FieldCoverageValidation:
5506:     """Test field coverage validation."""
5507:
5508:     def test_all_questions_have_required_fields(self):
5509:         """Test all questions have required fields."""
5510:         required_fields = {"id", "text"}
5511:         questions = [
5512:             {"id": "Q001", "text": "Question 1", "dimension": "D1"},
5513:             {"id": "Q002", "text": "Question 2", "dimension": "D2"},
5514:         ]
5515:
5516:         for q in questions:
5517:             assert required_fields.issubset(set(q.keys()))
5518:
5519:     def test_missing_optional_field_allowed(self):
5520:         """Test missing optional field is allowed."""
5521:         required_fields = {"id", "text"}
5522:         question = {"id": "Q001", "text": "Question 1"}
5523:
5524:         assert required_fields.issubset(set(question.keys()))
5525:         assert "optional_field" not in question
5526:
5527:     def test_field_coverage_percentage(self):
5528:         """Test field coverage percentage calculation."""
5529:         all_possible_fields = {"id", "text", "dimension", "weight", "type"}
5530:         questions = [
5531:             {"id": "Q001", "text": "Q1", "dimension": "D1"},
5532:             {"id": "Q002", "text": "Q2", "dimension": "D2", "weight": 1.0},
5533:         ]
5534:
5535:         total_fields = len(all_possible_fields) * len(questions)
5536:         present_fields = sum(
5537:             len(set(q.keys()) & all_possible_fields) for q in questions
5538:         )
5539:         coverage = present_fields / total_fields
5540:
5541:         assert 0.0 <= coverage <= 1.0
5542:
5543:
5544: class TestPhase6HashCalculation:
```

```
5545:     """Test hash calculation and verification."""
5546:
5547:     def test_schema_hash_calculated(self):
5548:         """Test schema hash is calculated in report."""
5549:         monolith = {
5550:             "schema_version": "2.0.0",
5551:             "version": "1.0.0",
5552:             "blocks": {
5553:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5554:                 "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5555:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5556:                 "macro_question": {"id": "MACRO_01"},
5557:                 "scoring": {},
5558:             },
5559:             "integrity": {},
5560:         }
5561:         validator = MonolithSchemaValidator()
5562:
5563:         report = validator.validate_monolith(monolith, strict=False)
5564:
5565:         assert report.schema_hash
5566:         assert len(report.schema_hash) > 0
5567:
5568:     def test_same_monolith_same_hash(self):
5569:         """Test same monolith produces same hash."""
5570:         monolith = {
5571:             "schema_version": "2.0.0",
5572:             "version": "1.0.0",
5573:             "blocks": {"test": "data"},
5574:             "integrity": {},
5575:         }
5576:         validator = MonolithSchemaValidator()
5577:
5578:         report1 = validator.validate_monolith(monolith, strict=False)
5579:         report2 = validator.validate_monolith(monolith, strict=False)
5580:
5581:         assert report1.schema_hash == report2.schema_hash
5582:
5583:     def test_different_monolith_different_hash(self):
5584:         """Test different monolith produces different hash."""
5585:         monolith1 = {
5586:             "schema_version": "2.0.0",
5587:             "version": "1.0.0",
5588:             "blocks": {"test": "data1"},
5589:             "integrity": {},
5590:         }
5591:         monolith2 = {
5592:             "schema_version": "2.0.0",
5593:             "version": "1.0.0",
5594:             "blocks": {"test": "data2"},
5595:             "integrity": {},
5596:         }
5597:         validator = MonolithSchemaValidator()
5598:
5599:         report1 = validator.validate_monolith(monolith1, strict=False)
5600:         report2 = validator.validate_monolith(monolith2, strict=False)
```

```
5601:
5602:         assert report1.schema_hash != report2.schema_hash
5603:
5604:
5605: class TestPhase6ValidationReport:
5606:     """Test validation report structure and content."""
5607:
5608:     def test_report_contains_timestamp(self):
5609:         """Test validation report contains timestamp."""
5610:         monolith = {
5611:             "schema_version": "2.0.0",
5612:             "version": "1.0.0",
5613:             "blocks": {},
5614:             "integrity": {},
5615:         }
5616:         validator = MonolithSchemaValidator()
5617:
5618:         report = validator.validate_monolith(monolith, strict=False)
5619:
5620:         assert report.timestamp
5621:         assert len(report.timestamp) > 0
5622:
5623:     def test_report_contains_schema_version(self):
5624:         """Test validation report contains schema version."""
5625:         monolith = {
5626:             "schema_version": "2.0.0",
5627:             "version": "1.0.0",
5628:             "blocks": {},
5629:             "integrity": {},
5630:         }
5631:         validator = MonolithSchemaValidator()
5632:
5633:         report = validator.validate_monolith(monolith, strict=False)
5634:
5635:         assert report.schema_version == "2.0.0"
5636:
5637:     def test_report_contains_validation_passed_flag(self):
5638:         """Test validation report contains validation_passed boolean."""
5639:         monolith = {
5640:             "schema_version": "2.0.0",
5641:             "version": "1.0.0",
5642:             "blocks": {
5643:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5644:                 "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5645:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5646:                 "macro_question": {"id": "MACRO_01"},
5647:                 "scoring": {},
5648:             },
5649:             "integrity": {},
5650:         }
5651:         validator = MonolithSchemaValidator()
5652:
5653:         report = validator.validate_monolith(monolith, strict=False)
5654:
5655:         assert isinstance(report.validation_passed, bool)
5656:
```

```
5657: def test_report_contains_errors_list(self):
5658:     """Test validation report contains errors list."""
5659:     monolith = {
5660:         "version": "1.0.0", # Missing schema_version
5661:         "blocks": {},
5662:         "integrity": {},
5663:     }
5664:     validator = MonolithSchemaValidator()
5665:
5666:     report = validator.validate_monolith(monolith, strict=False)
5667:
5668:     assert isinstance(report.errors, list)
5669:     assert len(report.errors) > 0
5670:
5671: def test_report_contains_warnings_list(self):
5672:     """Test validation report contains warnings list."""
5673:     monolith = {
5674:         "schema_version": "1.5.0", # Different version triggers warning
5675:         "version": "1.0.0",
5676:         "blocks": {
5677:             "niveles_abstraccion": ["micro", "meso", "macro"],
5678:             "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5679:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5680:             "macro_question": {"id": "MACRO_01"},
5681:             "scoring": {},
5682:         },
5683:         "integrity": {},
5684:     }
5685:     validator = MonolithSchemaValidator()
5686:
5687:     report = validator.validate_monolith(monolith, strict=False)
5688:
5689:     assert isinstance(report.warnings, list)
5690:
5691: def test_report_contains_question_counts(self):
5692:     """Test validation report contains question counts."""
5693:     monolith = {
5694:         "schema_version": "2.0.0",
5695:         "version": "1.0.0",
5696:         "blocks": {
5697:             "niveles_abstraccion": ["micro", "meso", "macro"],
5698:             "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5699:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5700:             "macro_question": {"id": "MACRO_01"},
5701:             "scoring": {},
5702:         },
5703:         "integrity": {},
5704:     }
5705:     validator = MonolithSchemaValidator()
5706:
5707:     report = validator.validate_monolith(monolith, strict=False)
5708:
5709:     assert isinstance(report.question_counts, dict)
5710:     assert "micro_questions" in report.question_counts
5711:
5712: def test_report_contains_referential_integrity(self):
```

```
5713:         """Test validation report contains referential integrity dict."""
5714:         monolith = {
5715:             "schema_version": "2.0.0",
5716:             "version": "1.0.0",
5717:             "blocks": {
5718:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5719:                 "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
5720:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5721:                 "macro_question": {"id": "MACRO_01"},
5722:                 "scoring": {},
5723:             },
5724:             "integrity": {},
5725:         }
5726:         validator = MonolithSchemaValidator()
5727:
5728:         report = validator.validate_monolith(monolith, strict=False)
5729:
5730:         assert isinstance(report.referential_integrity, dict)
5731:
5732:
5733: class TestPhase6StrictMode:
5734:     """Test strict mode validation behavior."""
5735:
5736:     def test_strict_mode_raises_exception_on_error(self):
5737:         """Test strict mode raises SchemaInitializationError on validation failure."""
5738:         monolith = {
5739:             "version": "1.0.0", # Missing schema_version
5740:             "blocks": {},
5741:             "integrity": {},
5742:         }
5743:         validator = MonolithSchemaValidator()
5744:
5745:         with pytest.raises(SchemaInitializationError):
5746:             validator.validate_monolith(monolith, strict=True)
5747:
5748:     def test_non_strict_mode_returns_report(self):
5749:         """Test non-strict mode returns report without raising."""
5750:         monolith = {
5751:             "version": "1.0.0", # Missing schema_version
5752:             "blocks": {},
5753:             "integrity": {},
5754:         }
5755:         validator = MonolithSchemaValidator()
5756:
5757:         report = validator.validate_monolith(monolith, strict=False)
5758:
5759:         assert isinstance(report, MonolithIntegrityReport)
5760:         assert report.validation_passed is False
5761:
5762:     def test_strict_mode_exception_contains_error_details(self):
5763:         """Test strict mode exception contains detailed error messages."""
5764:         monolith = {
5765:             "version": "1.0.0",
5766:             "blocks": {},
5767:             "integrity": {},
5768:         }
```

```
5769:         validator = MonolithSchemaValidator()
5770:
5771:         with pytest.raises(SchemaInitializationError) as exc_info:
5772:             validator.validate_monolith(monolith, strict=True)
5773:
5774:         error_msg = str(exc_info.value)
5775:         assert "Schema initialization failed" in error_msg
5776:
5777:
5778: class TestPhase6EdgeCases:
5779:     """Test edge cases in schema validation."""
5780:
5781:     def test_empty_monolith_dict(self):
5782:         """Test validation with empty monolith dict."""
5783:         monolith = {}
5784:         validator = MonolithSchemaValidator()
5785:
5786:         report = validator.validate_monolith(monolith, strict=False)
5787:
5788:         assert report.validation_passed is False
5789:         assert len(report.errors) > 0
5790:
5791:     def test_none_monolith_handled(self):
5792:         """Test validation handles None monolith gracefully."""
5793:         validator = MonolithSchemaValidator()
5794:
5795:         with pytest.raises((TypeError, AttributeError)):
5796:             validator.validate_monolith(None, strict=False)
5797:
5798:     def test_deeply_nested_structure(self):
5799:         """Test validation handles deeply nested structures."""
5800:         monolith = {
5801:             "schema_version": "2.0.0",
5802:             "version": "1.0.0",
5803:             "blocks": {
5804:                 "niveles_abstraccion": ["micro", "meso", "macro"],
5805:                 "micro_questions": [
5806:                     {
5807:                         "id": f"Q{i:03d}",
5808:                         "nested": {"level1": {"level2": {"level3": "deep_value"}}},
5809:                     }
5810:                     for i in range(1, 301)
5811:                 ],
5812:                 "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5813:                 "macro_question": {"id": "MACRO_01"},
5814:                 "scoring": {},
5815:             },
5816:             "integrity": {},
5817:         }
5818:         validator = MonolithSchemaValidator()
5819:
5820:         report = validator.validate_monolith(monolith, strict=False)
5821:
5822:         # Should not crash
5823:         assert isinstance(report, MonolithIntegrityReport)
5824:
```



```
5825: def test_unicode_in_question_text(self):
5826:     """Test validation handles unicode characters in questions."""
5827:     monolith = {
5828:         "schema_version": "2.0.0",
5829:         "version": "1.0.0",
5830:         "blocks": {
5831:             "niveles_abstraccion": ["micro", "meso", "macro"],
5832:             "micro_questions": [
5833:                 {"id": f"Q{i:03d}", "text": f"Pregunta {i} con Ã;Ã©Ã-Ã³Ã° Ã±"}
5834:                 for i in range(1, 301)
5835:             ],
5836:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5837:             "macro_question": {"id": "MACRO_01"},
5838:             "scoring": {},
5839:         },
5840:         "integrity": {},
5841:     }
5842:     validator = MonolithSchemaValidator()
5843:
5844:     report = validator.validate_monolith(monolith, strict=False)
5845:
5846:     assert isinstance(report, MonolithIntegrityReport)
5847:
5848: def test_large_monolith_performance(self):
5849:     """Test validation performance with large monolith."""
5850:     monolith = {
5851:         "schema_version": "2.0.0",
5852:         "version": "1.0.0",
5853:         "blocks": {
5854:             "niveles_abstraccion": ["micro", "meso", "macro"],
5855:             "micro_questions": [
5856:                 {
5857:                     "id": f"Q{i:03d}",
5858:                     "text": f"Question {i}" * 100, # Large text
5859:                     "dimension": f"D{(i % 6) + 1}",
5860:                     "weight": 1.0,
5861:                 }
5862:                 for i in range(1, 301)
5863:             ],
5864:             "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
5865:             "macro_question": {"id": "MACRO_01"},
5866:             "scoring": {},
5867:         },
5868:         "integrity": {},
5869:     }
5870:     validator = MonolithSchemaValidator()
5871:
5872:     report = validator.validate_monolith(monolith, strict=False)
5873:
5874:     assert isinstance(report, MonolithIntegrityReport)
5875:
5876:
5877:
5878: =====
5879: FILE: tests/phases/test_phase7_task_construction.py
5880: =====
```

```
5881:
5882: """Test Phase 7: Task Construction
5883:
5884: Tests Phase 7 task construction logic including:
5885: - ExecutableTask immutability enforcement
5886: - __post_init__ validation for all mandatory fields
5887: - Type coercion for tuple/MappingProxyType fields
5888: - Integration with _construct_task
5889: - Boundary cases for question_global range
5890: - Task ID generation and uniqueness
5891: - Metadata construction and provenance
5892: - Pattern and signal handling
5893: - Expected elements validation
5894: """
5895:
5896: from dataclasses import FrozenInstanceError
5897: from datetime import datetime
5898: from typing import Any
5899:
5900: import pytest
5901:
5902: from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
5903:     ChunkRoutingResult,
5904: )
5905: from farfan_pipeline.core.orchestrator.task_planner import (
5906:     MAX_QUESTION_GLOBAL,
5907:     ExecutableTask,
5908:     _construct_task,
5909:     _construct_task_legacy,
5910: )
5911: from farfan_pipeline.core.types import ChunkData
5912:
5913:
5914: def create_test_chunk_routing_result(
5915:     policy_area_id: str = "PA01",
5916:     chunk_id: str | None = None,
5917:     dimension_id: str = "DIM01",
5918:     text_content: str = "Test chunk content",
5919:     expected_elements: list[dict[str, Any]] | None = None,
5920:     document_position: tuple[int, int] | None = None,
5921: ) -> ChunkRoutingResult:
5922:     """Helper function to create test ChunkRoutingResult with all required fields."""
5923:     if expected_elements is None:
5924:         expected_elements = []
5925:     if document_position is None:
5926:         document_position = (0, 100)
5927:     if chunk_id is None:
5928:         chunk_id = f"{policy_area_id}-{dimension_id}"
5929:
5930:     target_chunk = ChunkData(
5931:         id=0,
5932:         text=text_content,
5933:         chunk_type="diagnostic",
5934:         sentences=[],
5935:         tables=[],
5936:         start_pos=0,
```

```
5937:         end_pos=len(text_content),
5938:         confidence=0.95,
5939:         chunk_id=chunk_id,
5940:         policy_area_id=policy_area_id,
5941:         dimension_id=dimension_id,
5942:     )
5943:
5944:     return ChunkRoutingResult(
5945:         target_chunk=target_chunk,
5946:         chunk_id=chunk_id,
5947:         policy_area_id=policy_area_id,
5948:         dimension_id=dimension_id,
5949:         text_content=text_content,
5950:         expected_elements=expected_elements,
5951:         document_position=document_position,
5952:     )
5953:
5954:
5955: class MockRoutingResult:
5956:     """Mock routing result for legacy tests."""
5957:
5958:     def __init__(self, policy_area_id: str = "PA01"):
5959:         self.policy_area_id = policy_area_id
5960:
5961:
5962: class TestPhase7ExecutableTaskImmutability:
5963:     """Test ExecutableTask immutability enforcement via frozen dataclass."""
5964:
5965:     def test_task_id_immutable(self):
5966:         """Verify task_id cannot be modified after creation."""
5967:         task = ExecutableTask(
5968:             task_id="MQC-001_PA01",
5969:             question_id="D1-Q1",
5970:             question_global=1,
5971:             policy_area_id="PA01",
5972:             dimension_id="DIM01",
5973:             chunk_id="chunk_001",
5974:             patterns=[],
5975:             signals={},
5976:             creation_timestamp="2024-01-01T00:00:00Z",
5977:             expected_elements=[],
5978:             metadata={},
5979:         )
5980:
5981:         with pytest.raises(FrozenInstanceError):
5982:             task.task_id = "MQC-002_PA01" # type: ignore[misc]
5983:
5984:     def test_question_id_immutable(self):
5985:         """Verify question_id cannot be modified after creation."""
5986:         task = ExecutableTask(
5987:             task_id="MQC-001_PA01",
5988:             question_id="D1-Q1",
5989:             question_global=1,
5990:             policy_area_id="PA01",
5991:             dimension_id="DIM01",
5992:             chunk_id="chunk_001",
```

```
5993:         patterns=[],
5994:         signals={},
5995:         creation_timestamp="2024-01-01T00:00:00Z",
5996:         expected_elements=[],
5997:         metadata={},
5998:     )
5999:
6000:     with pytest.raises(FrozenInstanceError):
6001:         task.question_id = "D1-Q2" # type: ignore[misc]
6002:
6003:     def test_question_global_immutable(self):
6004:         """Verify question_global cannot be modified after creation."""
6005:         task = ExecutableTask(
6006:             task_id="MQC-001_PA01",
6007:             question_id="D1-Q1",
6008:             question_global=1,
6009:             policy_area_id="PA01",
6010:             dimension_id="DIM01",
6011:             chunk_id="chunk_001",
6012:             patterns=[],
6013:             signals={},
6014:             creation_timestamp="2024-01-01T00:00:00Z",
6015:             expected_elements=[],
6016:             metadata={},
6017:         )
6018:
6019:         with pytest.raises(FrozenInstanceError):
6020:             task.question_global = 2 # type: ignore[misc]
6021:
6022:     def test_policy_area_id_immutable(self):
6023:         """Verify policy_area_id cannot be modified after creation."""
6024:         task = ExecutableTask(
6025:             task_id="MQC-001_PA01",
6026:             question_id="D1-Q1",
6027:             question_global=1,
6028:             policy_area_id="PA01",
6029:             dimension_id="DIM01",
6030:             chunk_id="chunk_001",
6031:             patterns=[],
6032:             signals={},
6033:             creation_timestamp="2024-01-01T00:00:00Z",
6034:             expected_elements=[],
6035:             metadata={},
6036:         )
6037:
6038:         with pytest.raises(FrozenInstanceError):
6039:             task.policy_area_id = "PA02" # type: ignore[misc]
6040:
6041:     def test_dimension_id_immutable(self):
6042:         """Verify dimension_id cannot be modified after creation."""
6043:         task = ExecutableTask(
6044:             task_id="MQC-001_PA01",
6045:             question_id="D1-Q1",
6046:             question_global=1,
6047:             policy_area_id="PA01",
6048:             dimension_id="DIM01",
```

```
6049:         chunk_id="chunk_001",
6050:         patterns=[],
6051:         signals={},
6052:         creation_timestamp="2024-01-01T00:00:00Z",
6053:         expected_elements=[],
6054:         metadata={},
6055:     )
6056:
6057:     with pytest.raises(FrozenInstanceError):
6058:         task.dimension_id = "DIM02" # type: ignore[misc]
6059:
6060:     def test_chunk_id_immutable(self):
6061:         """Verify chunk_id cannot be modified after creation."""
6062:         task = ExecutableTask(
6063:             task_id="MQC-001_PA01",
6064:             question_id="D1-Q1",
6065:             question_global=1,
6066:             policy_area_id="PA01",
6067:             dimension_id="DIM01",
6068:             chunk_id="chunk_001",
6069:             patterns=[],
6070:             signals={},
6071:             creation_timestamp="2024-01-01T00:00:00Z",
6072:             expected_elements=[],
6073:             metadata={},
6074:         )
6075:
6076:         with pytest.raises(FrozenInstanceError):
6077:             task.chunk_id = "chunk_002" # type: ignore[misc]
6078:
6079:     def test_patterns_immutable(self):
6080:         """Verify patterns list cannot be reassigned."""
6081:         task = ExecutableTask(
6082:             task_id="MQC-001_PA01",
6083:             question_id="D1-Q1",
6084:             question_global=1,
6085:             policy_area_id="PA01",
6086:             dimension_id="DIM01",
6087:             chunk_id="chunk_001",
6088:             patterns=[{"type": "pattern1"}],
6089:             signals={},
6090:             creation_timestamp="2024-01-01T00:00:00Z",
6091:             expected_elements=[],
6092:             metadata={},
6093:         )
6094:
6095:         with pytest.raises(FrozenInstanceError):
6096:             task.patterns = [{"type": "pattern2"}] # type: ignore[misc]
6097:
6098:     def test_signals_immutable(self):
6099:         """Verify signals dict cannot be reassigned."""
6100:         task = ExecutableTask(
6101:             task_id="MQC-001_PA01",
6102:             question_id="D1-Q1",
6103:             question_global=1,
6104:             policy_area_id="PA01",
```

```
6105:         dimension_id="DIM01",
6106:         chunk_id="chunk_001",
6107:         patterns=[],
6108:         signals={"signal1": 0.5},
6109:         creation_timestamp="2024-01-01T00:00:00Z",
6110:         expected_elements=[],
6111:         metadata={},
6112:     )
6113:
6114:     with pytest.raises(FrozenInstanceError):
6115:         task.signals = {"signal2": 0.7} # type: ignore[misc]
6116:
6117:     def test_creation_timestamp_immutable(self):
6118:         """Verify creation_timestamp cannot be modified."""
6119:         task = ExecutableTask(
6120:             task_id="MQC-001_PA01",
6121:             question_id="D1-Q1",
6122:             question_global=1,
6123:             policy_area_id="PA01",
6124:             dimension_id="DIM01",
6125:             chunk_id="chunk_001",
6126:             patterns=[],
6127:             signals={},
6128:             creation_timestamp="2024-01-01T00:00:00Z",
6129:             expected_elements=[],
6130:             metadata={},
6131:         )
6132:
6133:         with pytest.raises(FrozenInstanceError):
6134:             task.creation_timestamp = "2024-01-02T00:00:00Z" # type: ignore[misc]
6135:
6136:     def test_expected_elements_immutable(self):
6137:         """Verify expected_elements list cannot be reassigned."""
6138:         task = ExecutableTask(
6139:             task_id="MQC-001_PA01",
6140:             question_id="D1-Q1",
6141:             question_global=1,
6142:             policy_area_id="PA01",
6143:             dimension_id="DIM01",
6144:             chunk_id="chunk_001",
6145:             patterns=[],
6146:             signals={},
6147:             creation_timestamp="2024-01-01T00:00:00Z",
6148:             expected_elements=[{"type": "test"}],
6149:             metadata={},
6150:         )
6151:
6152:         with pytest.raises(FrozenInstanceError):
6153:             task.expected_elements = [] # type: ignore[misc]
6154:
6155:     def test_metadata_immutable(self):
6156:         """Verify metadata dict cannot be reassigned."""
6157:         task = ExecutableTask(
6158:             task_id="MQC-001_PA01",
6159:             question_id="D1-Q1",
6160:             question_global=1,
```

```
6161:         policy_area_id="PA01",
6162:         dimension_id="DIM01",
6163:         chunk_id="chunk_001",
6164:         patterns=[],
6165:         signals={},
6166:         creation_timestamp="2024-01-01T00:00:00Z",
6167:         expected_elements=[],
6168:         metadata={"key": "value"},
6169:     )
6170:
6171:     with pytest.raises(FrozenInstanceError):
6172:         task.metadata = {} # type: ignore[misc]
6173:
6174:
6175: class TestPhase7PostInitValidation:
6176:     """Test __post_init__ validation for all mandatory fields."""
6177:
6178:     def test_empty_task_id_raises_error(self):
6179:         """Test empty task_id raises ValueError."""
6180:         with pytest.raises(ValueError, match="task_id cannot be empty"):
6181:             ExecutableTask(
6182:                 task_id="",
6183:                 question_id="D1-Q1",
6184:                 question_global=1,
6185:                 policy_area_id="PA01",
6186:                 dimension_id="DIM01",
6187:                 chunk_id="chunk_001",
6188:                 patterns=[],
6189:                 signals={},
6190:                 creation_timestamp="2024-01-01T00:00:00Z",
6191:                 expected_elements=[],
6192:                 metadata={},
6193:             )
6194:
6195:     def test_empty_question_id_raises_error(self):
6196:         """Test empty question_id raises ValueError."""
6197:         with pytest.raises(ValueError, match="question_id cannot be empty"):
6198:             ExecutableTask(
6199:                 task_id="MQC-001_PA01",
6200:                 question_id="",
6201:                 question_global=1,
6202:                 policy_area_id="PA01",
6203:                 dimension_id="DIM01",
6204:                 chunk_id="chunk_001",
6205:                 patterns=[],
6206:                 signals={},
6207:                 creation_timestamp="2024-01-01T00:00:00Z",
6208:                 expected_elements=[],
6209:                 metadata={},
6210:             )
6211:
6212:     def test_non_integer_question_global_raises_error(self):
6213:         """Test non-integer question_global raises ValueError."""
6214:         with pytest.raises(
6215:             ValueError, match="question_global must be an integer, got str"
6216:         ):
```

```
6217:         ExecutableTask(
6218:             task_id="MQC-001_PA01",
6219:             question_id="D1-Q1",
6220:             question_global="1", # type: ignore[arg-type]
6221:             policy_area_id="PA01",
6222:             dimension_id="DIM01",
6223:             chunk_id="chunk_001",
6224:             patterns=[],
6225:             signals={},
6226:             creation_timestamp="2024-01-01T00:00:00Z",
6227:             expected_elements=[],
6228:             metadata={},
6229:         )
6230:
6231: def test_question_global_below_zero_raises_error(self):
6232:     """Test question_global below 0 raises ValueError."""
6233:     with pytest.raises(
6234:         ValueError,
6235:         match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}",
6236:     ):
6237:         ExecutableTask(
6238:             task_id="MQC-001_PA01",
6239:             question_id="D1-Q1",
6240:             question_global=-1,
6241:             policy_area_id="PA01",
6242:             dimension_id="DIM01",
6243:             chunk_id="chunk_001",
6244:             patterns=[],
6245:             signals={},
6246:             creation_timestamp="2024-01-01T00:00:00Z",
6247:             expected_elements=[],
6248:             metadata={},
6249:         )
6250:
6251: def test_question_global_above_max_raises_error(self):
6252:     """Test question_global above MAX_QUESTION_GLOBAL raises ValueError."""
6253:     with pytest.raises(
6254:         ValueError,
6255:         match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}",
6256:     ):
6257:         ExecutableTask(
6258:             task_id="MQC-999_PA01",
6259:             question_id="D1-Q1",
6260:             question_global=MAX_QUESTION_GLOBAL + 1,
6261:             policy_area_id="PA01",
6262:             dimension_id="DIM01",
6263:             chunk_id="chunk_001",
6264:             patterns=[],
6265:             signals={},
6266:             creation_timestamp="2024-01-01T00:00:00Z",
6267:             expected_elements=[],
6268:             metadata={},
6269:         )
6270:
6271: def test_empty_policy_area_id_raises_error(self):
6272:     """Test empty policy_area_id raises ValueError."""
```



```
6273:         with pytest.raises(ValueError, match="policy_area_id cannot be empty"):
6274:             ExecutableTask(
6275:                 task_id="MQC-001_PA01",
6276:                 question_id="D1-Q1",
6277:                 question_global=1,
6278:                 policy_area_id="",
6279:                 dimension_id="DIM01",
6280:                 chunk_id="chunk_001",
6281:                 patterns=[],
6282:                 signals={},
6283:                 creation_timestamp="2024-01-01T00:00:00Z",
6284:                 expected_elements=[],
6285:                 metadata={},
6286:             )
6287:
6288: def test_empty_dimension_id_raises_error(self):
6289:     """Test empty dimension_id raises ValueError."""
6290:     with pytest.raises(ValueError, match="dimension_id cannot be empty"):
6291:         ExecutableTask(
6292:             task_id="MQC-001_PA01",
6293:             question_id="D1-Q1",
6294:             question_global=1,
6295:             policy_area_id="PA01",
6296:             dimension_id="",
6297:             chunk_id="chunk_001",
6298:             patterns=[],
6299:             signals={},
6300:             creation_timestamp="2024-01-01T00:00:00Z",
6301:             expected_elements=[],
6302:             metadata={},
6303:         )
6304:
6305: def test_empty_chunk_id_raises_error(self):
6306:     """Test empty chunk_id raises ValueError."""
6307:     with pytest.raises(ValueError, match="chunk_id cannot be empty"):
6308:         ExecutableTask(
6309:             task_id="MQC-001_PA01",
6310:             question_id="D1-Q1",
6311:             question_global=1,
6312:             policy_area_id="PA01",
6313:             dimension_id="DIM01",
6314:             chunk_id="",
6315:             patterns=[],
6316:             signals={},
6317:             creation_timestamp="2024-01-01T00:00:00Z",
6318:             expected_elements=[],
6319:             metadata={},
6320:         )
6321:
6322: def test_empty_creation_timestamp_raises_error(self):
6323:     """Test empty creation_timestamp raises ValueError."""
6324:     with pytest.raises(ValueError, match="creation_timestamp cannot be empty"):
6325:         ExecutableTask(
6326:             task_id="MQC-001_PA01",
6327:             question_id="D1-Q1",
6328:             question_global=1,
```

```
6329:         policy_area_id="PA01",
6330:         dimension_id="DIM01",
6331:         chunk_id="chunk_001",
6332:         patterns=[],
6333:         signals={},
6334:         creation_timestamp="",
6335:         expected_elements=[],
6336:         metadata={},
6337:     )
6338:
6339: def test_valid_minimum_question_global(self):
6340:     """Test valid minimum question_global value (0)."""
6341:     task = ExecutableTask(
6342:         task_id="MQC-000_PA01",
6343:         question_id="D1-Q0",
6344:         question_global=0,
6345:         policy_area_id="PA01",
6346:         dimension_id="DIM01",
6347:         chunk_id="chunk_001",
6348:         patterns=[],
6349:         signals={},
6350:         creation_timestamp="2024-01-01T00:00:00Z",
6351:         expected_elements=[],
6352:         metadata={},
6353:     )
6354:     assert task.question_global == 0
6355:
6356: def test_valid_maximum_question_global(self):
6357:     """Test valid maximum question_global value (MAX_QUESTION_GLOBAL)."""
6358:     task = ExecutableTask(
6359:         task_id=f"MQC-{MAX_QUESTION_GLOBAL:03d}_PA01",
6360:         question_id="D1-Q999",
6361:         question_global=MAX_QUESTION_GLOBAL,
6362:         policy_area_id="PA01",
6363:         dimension_id="DIM01",
6364:         chunk_id="chunk_001",
6365:         patterns=[],
6366:         signals={},
6367:         creation_timestamp="2024-01-01T00:00:00Z",
6368:         expected_elements=[],
6369:         metadata={},
6370:     )
6371:     assert task.question_global == MAX_QUESTION_GLOBAL
6372:
6373: def test_all_valid_fields_create_task(self):
6374:     """Test task creation with all valid fields."""
6375:     task = ExecutableTask(
6376:         task_id="MQC-042_PA05",
6377:         question_id="D2-Q12",
6378:         question_global=42,
6379:         policy_area_id="PA05",
6380:         dimension_id="DIM02",
6381:         chunk_id="chunk_010",
6382:         patterns=[{"type": "pattern1"}, {"type": "pattern2"}],
6383:         signals={"signal1": 0.8, "signal2": 0.9},
6384:         creation_timestamp="2024-01-01T12:30:45.123456Z",
```

```
6385:         expected_elements=[{"type": "test", "minimum": 2}],
6386:         metadata={"key1": "value1", "key2": "value2"},
6387:     )
6388:     assert task.task_id == "MQC-042_PA05"
6389:     assert task.question_id == "D2-Q12"
6390:     assert task.question_global == 42
6391:     assert task.policy_area_id == "PA05"
6392:     assert task.dimension_id == "DIM02"
6393:     assert task.chunk_id == "chunk_010"
6394:
6395:
6396: class TestPhase7ConstructTaskIntegration:
6397:     """Test integration with _construct_task function."""
6398:
6399:     def test_construct_task_generates_correct_task_id(self):
6400:         """Test _construct_task generates correct task_id format."""
6401:         question = {
6402:             "question_id": "D1-Q1",
6403:             "question_global": 1,
6404:             "dimension_id": "DIM01",
6405:             "base_slot": "D1-Q1",
6406:             "cluster_id": "CL01",
6407:             "expected_elements": [],
6408:         }
6409:         routing_result = create_test_chunk_routing_result(policy_area_id="PA01")
6410:         generated_task_ids: set[str] = set()
6411:         correlation_id = "corr-123"
6412:
6413:         task = _construct_task(
6414:             question,
6415:             routing_result,
6416:             (),
6417:             (),
6418:             generated_task_ids,
6419:             correlation_id,
6420:         )
6421:
6422:         assert task.task_id == "MQC-001_PA01"
6423:         assert "MQC-001_PA01" in generated_task_ids
6424:
6425:     def test_construct_task_with_various_question_globals(self):
6426:         """Test _construct_task with various question_global values."""
6427:         test_cases = [
6428:             (0, "MQC-000_PA01"),
6429:             (1, "MQC-001_PA01"),
6430:             (50, "MQC-050_PA01"),
6431:             (150, "MQC-150_PA01"),
6432:             (300, "MQC-300_PA01"),
6433:             (999, "MQC-999_PA01"),
6434:         ]
6435:
6436:         for question_global, expected_task_id in test_cases:
6437:             question = {
6438:                 "question_id": f"Q{question_global}",
6439:                 "question_global": question_global,
6440:                 "dimension_id": "DIM01",
```

```
6441:         "base_slot": f"Q{question_global}",
6442:         "cluster_id": "CL01",
6443:         "expected_elements": [],
6444:     }
6445:     routing_result = create_test_chunk_routing_result(policy_area_id="PA01")
6446:     generated_task_ids: set[str] = set()
6447:
6448:     task = _construct_task(
6449:         question,
6450:         routing_result,
6451:         (),
6452:         (),
6453:         generated_task_ids,
6454:         "corr-id",
6455:     )
6456:
6457:     assert task.task_id == expected_task_id
6458:     assert task.question_global == question_global
6459:
6460: def test_construct_task_detects_duplicate_task_ids(self):
6461:     """Test _construct_task detects duplicate task_ids."""
6462:     question = {
6463:         "question_id": "D1-Q1",
6464:         "question_global": 1,
6465:         "dimension_id": "DIM01",
6466:         "base_slot": "D1-Q1",
6467:         "cluster_id": "CL01",
6468:         "expected_elements": [],
6469:     }
6470:     routing_result = create_test_chunk_routing_result(policy_area_id="PA01")
6471:     generated_task_ids = {"MQC-001_PA01"} # Pre-populate with duplicate
6472:
6473:     with pytest.raises(
6474:         ValueError, match="Duplicate task_id detected: MQC-001_PA01"
6475:     ):
6476:         _construct_task(
6477:             question,
6478:             routing_result,
6479:             (),
6480:             (),
6481:             generated_task_ids,
6482:             "corr-id",
6483:         )
6484:
6485: def test_construct_task_missing_question_global(self):
6486:     """Test _construct_task raises error when question_global is missing."""
6487:     question = {
6488:         "question_id": "D1-Q1",
6489:         # question_global is missing
6490:         "dimension_id": "DIM01",
6491:         "expected_elements": [],
6492:     }
6493:     routing_result = create_test_chunk_routing_result()
6494:     generated_task_ids: set[str] = set()
6495:
6496:     with pytest.raises(ValueError, match="question_global field missing or None"):
```

```
6497:         _construct_task(
6498:             question,
6499:             routing_result,
6500:             (),
6501:             (),
6502:             generated_task_ids,
6503:             "corr-id",
6504:         )
6505:
6506: def test_construct_task_question_global_none(self):
6507:     """Test _construct_task raises error when question_global is None."""
6508:     question = {
6509:         "question_id": "D1-Q1",
6510:         "question_global": None,
6511:         "dimension_id": "DIM01",
6512:         "expected_elements": [],
6513:     }
6514:     routing_result = create_test_chunk_routing_result()
6515:     generated_task_ids: set[str] = set()
6516:
6517:     with pytest.raises(ValueError, match="question_global field missing or None"):
6518:         _construct_task(
6519:             question,
6520:             routing_result,
6521:             (),
6522:             (),
6523:             generated_task_ids,
6524:             "corr-id",
6525:         )
6526:
6527: def test_construct_task_question_global_not_integer(self):
6528:     """Test _construct_task raises error when question_global is not integer."""
6529:     question = {
6530:         "question_id": "D1-Q1",
6531:         "question_global": "1", # String instead of int
6532:         "dimension_id": "DIM01",
6533:         "expected_elements": [],
6534:     }
6535:     routing_result = create_test_chunk_routing_result()
6536:     generated_task_ids: set[str] = set()
6537:
6538:     with pytest.raises(
6539:         ValueError, match="question_global must be an integer, got str"
6540:     ):
6541:         _construct_task(
6542:             question,
6543:             routing_result,
6544:             (),
6545:             (),
6546:             generated_task_ids,
6547:             "corr-id",
6548:         )
6549:
6550: def test_construct_task_question_global_below_range(self):
6551:     """Test _construct_task raises error when question_global is below 0."""
6552:     question = {
```

```
6553:         "question_id": "D1-Q1",
6554:         "question_global": -1,
6555:         "dimension_id": "DIM01",
6556:         "expected_elements": [],
6557:     }
6558:     routing_result = create_test_chunk_routing_result()
6559:     generated_task_ids: set[str] = set()
6560:
6561:     with pytest.raises(
6562:         ValueError,
6563:         match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}, got -1",
6564:     ):
6565:         _construct_task(
6566:             question,
6567:             routing_result,
6568:             (),
6569:             (),
6570:             generated_task_ids,
6571:             "corr-id",
6572:         )
6573:
6574: def test_construct_task_question_global_above_range(self):
6575:     """Test _construct_task raises error when question_global exceeds MAX."""
6576:     question = {
6577:         "question_id": "D1-Q1000",
6578:         "question_global": 1000,
6579:         "dimension_id": "DIM01",
6580:         "expected_elements": [],
6581:     }
6582:     routing_result = create_test_chunk_routing_result()
6583:     generated_task_ids: set[str] = set()
6584:
6585:     with pytest.raises(
6586:         ValueError,
6587:         match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}, got 1000",
6588:     ):
6589:         _construct_task(
6590:             question,
6591:             routing_result,
6592:             (),
6593:             (),
6594:             generated_task_ids,
6595:             "corr-id",
6596:         )
6597:
6598: def test_construct_task_with_patterns_tuple(self):
6599:     """Test _construct_task handles patterns as tuple."""
6600:     question = {
6601:         "question_id": "D1-Q1",
6602:         "question_global": 1,
6603:         "dimension_id": "DIM01",
6604:         "base_slot": "D1-Q1",
6605:         "cluster_id": "CL01",
6606:         "expected_elements": [],
6607:     }
6608:     routing_result = create_test_chunk_routing_result()
```

```
6609:     applicable_patterns = (
6610:         {"type": "pattern1", "value": 0.8},
6611:         {"type": "pattern2", "value": 0.9},
6612:     )
6613:     generated_task_ids: set[str] = set()
6614:
6615:     task = _construct_task(
6616:         question,
6617:         routing_result,
6618:         applicable_patterns,
6619:         (),
6620:         generated_task_ids,
6621:         "corr-id",
6622:     )
6623:
6624:     assert len(task.patterns) == 2
6625:     assert isinstance(task.patterns, list)
6626:     assert task.patterns[0]["type"] == "pattern1"
6627:     assert task.patterns[1]["type"] == "pattern2"
6628:
6629: def test_construct_task_with_signals_tuple(self):
6630:     """Test _construct_task handles signals as tuple and converts to dict."""
6631:     question = {
6632:         "question_id": "D1-Q1",
6633:         "question_global": 1,
6634:         "dimension_id": "DIM01",
6635:         "base_slot": "D1-Q1",
6636:         "cluster_id": "CL01",
6637:         "expected_elements": [],
6638:     }
6639:     routing_result = create_test_chunk_routing_result()
6640:
6641:     # Signals as tuple of dicts with signal_type
6642:     resolved_signals = (
6643:         {"signal_type": "signal1", "value": 0.8},
6644:         {"signal_type": "signal2", "value": 0.9},
6645:     )
6646:     generated_task_ids: set[str] = set()
6647:
6648:     task = _construct_task(
6649:         question,
6650:         routing_result,
6651:         (),
6652:         resolved_signals,
6653:         generated_task_ids,
6654:         "corr-id",
6655:     )
6656:
6657:     assert isinstance(task.signals, dict)
6658:     assert "signal1" in task.signals
6659:     assert "signal2" in task.signals
6660:     assert task.signals["signal1"]["value"] == 0.8
6661:     assert task.signals["signal2"]["value"] == 0.9
6662:
6663: def test_construct_task_with_expected_elements(self):
6664:     """Test _construct_task includes expected_elements in task."""
```

```
6665:         expected_elements = [
6666:             {"type": "fuentes_oficiales", "minimum": 2, "required": True},
6667:             {"type": "indicadores_cuantitativos", "minimum": 3, "required": False},
6668:         ]
6669:         question = {
6670:             "question_id": "D1-Q1",
6671:             "question_global": 1,
6672:             "dimension_id": "DIM01",
6673:             "base_slot": "D1-Q1",
6674:             "cluster_id": "CL01",
6675:             "expected_elements": expected_elements,
6676:         }
6677:         routing_result = create_test_chunk_routing_result()
6678:         generated_task_ids: set[str] = set()
6679:
6680:         task = _construct_task(
6681:             question,
6682:             routing_result,
6683:             (),
6684:             (),
6685:             generated_task_ids,
6686:             "corr-id",
6687:         )
6688:
6689:         assert len(task.expected_elements) == 2
6690:         assert isinstance(task.expected_elements, list)
6691:         assert task.expected_elements[0]["type"] == "fuentes_oficiales"
6692:         assert task.expected_elements[1]["type"] == "indicadores_cuantitativos"
6693:
6694:     def test_construct_task_includes_metadata(self):
6695:         """Test _construct_task includes comprehensive metadata."""
6696:         question = {
6697:             "question_id": "D1-Q1",
6698:             "question_global": 1,
6699:             "dimension_id": "DIM01",
6700:             "base_slot": "D1-Q1",
6701:             "cluster_id": "CL01",
6702:             "expected_elements": [{"type": "test"}],
6703:         }
6704:         routing_result = create_test_chunk_routing_result(document_position=(100, 200))
6705:         applicable_patterns = ({ "pattern": "p1"}, { "pattern": "p2"})
6706:         resolved_signals = ({ "signal_type": "s1", "value": 0.5},)
6707:         generated_task_ids: set[str] = set()
6708:         correlation_id = "corr-abc-123"
6709:
6710:         task = _construct_task(
6711:             question,
6712:             routing_result,
6713:             applicable_patterns,
6714:             resolved_signals,
6715:             generated_task_ids,
6716:             correlation_id,
6717:         )
6718:
6719:         assert "base_slot" in task.metadata
6720:         assert "cluster_id" in task.metadata
```



```
6721:         assert "document_position" in task.metadata
6722:         assert "synchronizer_version" in task.metadata
6723:         assert "correlation_id" in task.metadata
6724:         assert "original_pattern_count" in task.metadata
6725:         assert "original_signal_count" in task.metadata
6726:         assert "filtered_pattern_count" in task.metadata
6727:         assert "resolved_signal_count" in task.metadata
6728:         assert "schema_element_count" in task.metadata
6729:
6730:         assert task.metadata["base_slot"] == "D1-Q1"
6731:         assert task.metadata["cluster_id"] == "CL01"
6732:         assert task.metadata["document_position"] == (100, 200)
6733:         assert task.metadata["correlation_id"] == "corr-abc-123"
6734:         assert task.metadata["original_pattern_count"] == 2
6735:         assert task.metadata["original_signal_count"] == 1
6736:         assert task.metadata["filtered_pattern_count"] == 2
6737:         assert task.metadata["resolved_signal_count"] == 1
6738:         assert task.metadata["schema_element_count"] == 1
6739:
6740:     def test_construct_task_timestamp_format(self):
6741:         """Test _construct_task creates ISO 8601 timestamp."""
6742:         question = {
6743:             "question_id": "D1-Q1",
6744:             "question_global": 1,
6745:             "dimension_id": "DIM01",
6746:             "base_slot": "D1-Q1",
6747:             "cluster_id": "CL01",
6748:             "expected_elements": [],
6749:         }
6750:         routing_result = create_test_chunk_routing_result()
6751:         generated_task_ids: set[str] = set()
6752:
6753:         task = _construct_task(
6754:             question,
6755:             routing_result,
6756:             (),
6757:             (),
6758:             generated_task_ids,
6759:             "corr-id",
6760:         )
6761:
6762:         # Timestamp should be ISO 8601 format with timezone
6763:         assert "T" in task.creation_timestamp
6764:         # Should be parseable by datetime
6765:         parsed = datetime.fromisoformat(task.creation_timestamp)
6766:         assert parsed.tzinfo is not None
6767:
6768:     def test_construct_task_uses_routing_result_dimension(self):
6769:         """Test _construct_task uses dimension_id from routing_result."""
6770:         question = {
6771:             "question_id": "D1-Q1",
6772:             "question_global": 1,
6773:             "dimension_id": "DIM01", # Different from routing_result
6774:             "base_slot": "D1-Q1",
6775:             "cluster_id": "CL01",
6776:             "expected_elements": [],
```

```
6777:     }
6778:     routing_result = create_test_chunk_routing_result(dimension_id="DIM02")
6779:     generated_task_ids: set[str] = set()
6780:
6781:     task = _construct_task(
6782:         question,
6783:         routing_result,
6784:         (),
6785:         (),
6786:         generated_task_ids,
6787:         "corr-id",
6788:     )
6789:
6790:     # Should use routing_result dimension_id, not question dimension_id
6791:     assert task.dimension_id == "DIM02"
6792:
6793:
6794: class TestPhase7ConstructTaskLegacy:
6795:     """Test legacy _construct_task_legacy function."""
6796:
6797:     def test_construct_task_legacy_generates_correct_task_id(self):
6798:         """Test _construct_task_legacy generates correct task_id format."""
6799:         question = {
6800:             "question_id": "D1-Q1",
6801:             "question_global": 1,
6802:             "policy_area_id": "PA01",
6803:             "dimension_id": "DIM01",
6804:             "expected_elements": [],
6805:         }
6806:         chunk = {"id": "chunk_001", "expected_elements": []}
6807:         patterns: list[dict[str, Any]] = []
6808:         signals: dict[str, Any] = {}
6809:         generated_task_ids: set[str] = set()
6810:         routing_result = MockRoutingResult(policy_area_id="PA01")
6811:
6812:         task = _construct_task_legacy(
6813:             question, chunk, patterns, signals, generated_task_ids, routing_result
6814:         )
6815:
6816:         assert task.task_id == "MQC-001_PA01"
6817:         assert "MQC-001_PA01" in generated_task_ids
6818:
6819:     def test_construct_task_legacy_detects_duplicate_task_ids(self):
6820:         """Test _construct_task_legacy detects duplicate task_ids."""
6821:         question = {
6822:             "question_id": "D1-Q1",
6823:             "question_global": 1,
6824:             "dimension_id": "DIM01",
6825:             "expected_elements": [],
6826:         }
6827:         chunk = {"id": "chunk_001", "expected_elements": []}
6828:         generated_task_ids = {"MQC-001_PA01"}
6829:         routing_result = MockRoutingResult(policy_area_id="PA01")
6830:
6831:         with pytest.raises(
6832:             ValueError,
```

```
6833:         match="Duplicate task_id detected: MQC-001_PA01 for question D1-Q1",
6834:     ):
6835:         _construct_task_legacy(
6836:             question, chunk, [], {}, generated_task_ids, routing_result
6837:         )
6838:
6839: def test_construct_task_legacy_invalid_question_global_type(self):
6840:     """Test _construct_task_legacy raises error for non-integer question_global."""
6841:     question = {
6842:         "question_id": "D1-Q1",
6843:         "question_global": "not_an_int",
6844:         "dimension_id": "DIM01",
6845:         "expected_elements": [],
6846:     }
6847:     chunk = {"id": "chunk_001"}
6848:     generated_task_ids: set[str] = set()
6849:     routing_result = MockRoutingResult()
6850:
6851:     with pytest.raises(
6852:         ValueError, match="Invalid question_global.*Must be an integer in range"
6853:     ):
6854:         _construct_task_legacy(
6855:             question, chunk, [], {}, generated_task_ids, routing_result
6856:         )
6857:
6858: def test_construct_task_legacy_question_global_below_range(self):
6859:     """Test _construct_task_legacy raises error for negative question_global."""
6860:     question = {
6861:         "question_id": "D1-Q1",
6862:         "question_global": -1,
6863:         "dimension_id": "DIM01",
6864:         "expected_elements": [],
6865:     }
6866:     chunk = {"id": "chunk_001"}
6867:     generated_task_ids: set[str] = set()
6868:     routing_result = MockRoutingResult()
6869:
6870:     with pytest.raises(
6871:         ValueError, match="Invalid question_global.*Must be an integer in range"
6872:     ):
6873:         _construct_task_legacy(
6874:             question, chunk, [], {}, generated_task_ids, routing_result
6875:         )
6876:
6877: def test_construct_task_legacy_question_global_above_range(self):
6878:     """Test _construct_task_legacy raises error for question_global > MAX."""
6879:     question = {
6880:         "question_id": "D1-Q1",
6881:         "question_global": 1000,
6882:         "dimension_id": "DIM01",
6883:         "expected_elements": [],
6884:     }
6885:     chunk = {"id": "chunk_001"}
6886:     generated_task_ids: set[str] = set()
6887:     routing_result = MockRoutingResult()
6888:
```

```
6889:         with pytest.raises(
6890:             ValueError, match="Invalid question_global.*Must be an integer in range"
6891:         ):
6892:             _construct_task_legacy(
6893:                 question, chunk, [], {}, generated_task_ids, routing_result
6894:             )
6895:
6896: def test_construct_task_legacy_coerces_patterns_to_list(self):
6897:     """Test _construct_task_legacy coerces patterns tuple to list."""
6898:     question = {
6899:         "question_id": "D1-Q1",
6900:         "question_global": 1,
6901:         "dimension_id": "DIM01",
6902:         "expected_elements": [],
6903:     }
6904:     chunk = {"id": "chunk_001"}
6905:     patterns = ({ "pattern": "p1" }, { "pattern": "p2" }) # Tuple
6906:     generated_task_ids: set[str] = set()
6907:     routing_result = MockRoutingResult()
6908:
6909:     task = _construct_task_legacy(
6910:         question, chunk, patterns, {}, generated_task_ids, routing_result # type: ignore[arg-type]
6911:     )
6912:
6913:     assert isinstance(task.patterns, list)
6914:     assert len(task.patterns) == 2
6915:
6916: def test_construct_task_legacy_coerces_signals_to_dict(self):
6917:     """Test _construct_task_legacy coerces signals to dict if needed."""
6918:     question = {
6919:         "question_id": "D1-Q1",
6920:         "question_global": 1,
6921:         "dimension_id": "DIM01",
6922:         "expected_elements": [],
6923:     }
6924:     chunk = {"id": "chunk_001"}
6925:     signals = {"signal1": 0.8, "signal2": 0.9}
6926:     generated_task_ids: set[str] = set()
6927:     routing_result = MockRoutingResult()
6928:
6929:     task = _construct_task_legacy(
6930:         question, chunk, [], signals, generated_task_ids, routing_result
6931:     )
6932:
6933:     assert isinstance(task.signals, dict)
6934:     assert task.signals == signals
6935:
6936: def test_construct_task_legacy_timestamp_is_iso8601(self):
6937:     """Test _construct_task_legacy creates ISO 8601 timestamp."""
6938:     question = {
6939:         "question_id": "D1-Q1",
6940:         "question_global": 1,
6941:         "dimension_id": "DIM01",
6942:         "expected_elements": [],
6943:     }
6944:     chunk = {"id": "chunk_001"}
```

```
6945:     generated_task_ids: set[str] = set()
6946:     routing_result = MockRoutingResult()
6947:
6948:     task = _construct_task_legacy(
6949:         question, chunk, [], {}, generated_task_ids, routing_result
6950:     )
6951:
6952:     # Should have "T" separator for ISO 8601
6953:     assert "T" in task.creation_timestamp
6954:     # Should be parseable
6955:     parsed = datetime.fromisoformat(task.creation_timestamp)
6956:     assert parsed is not None
6957:
6958:
6959: class TestPhase7BoundaryConditions:
6960:     """Test boundary conditions for question_global range."""
6961:
6962:     def test_question_global_boundary_zero(self):
6963:         """Test question_global=0 is valid."""
6964:         task = ExecutableTask(
6965:             task_id="MQC-000_PA01",
6966:             question_id="Q0",
6967:             question_global=0,
6968:             policy_area_id="PA01",
6969:             dimension_id="DIM01",
6970:             chunk_id="chunk_001",
6971:             patterns=[],
6972:             signals={},
6973:             creation_timestamp="2024-01-01T00:00:00Z",
6974:             expected_elements=[],
6975:             metadata={},
6976:         )
6977:         assert task.question_global == 0
6978:
6979:     def test_question_global_boundary_max(self):
6980:         """Test question_global=MAX_QUESTION_GLOBAL is valid."""
6981:         task = ExecutableTask(
6982:             task_id=f"MQC-{{MAX_QUESTION_GLOBAL:03d}}_PA01",
6983:             question_id=f"Q{{MAX_QUESTION_GLOBAL}}",
6984:             question_global=MAX_QUESTION_GLOBAL,
6985:             policy_area_id="PA01",
6986:             dimension_id="DIM01",
6987:             chunk_id="chunk_001",
6988:             patterns=[],
6989:             signals={},
6990:             creation_timestamp="2024-01-01T00:00:00Z",
6991:             expected_elements=[],
6992:             metadata={},
6993:         )
6994:         assert task.question_global == MAX_QUESTION_GLOBAL
6995:
6996:     def test_question_global_just_below_zero_invalid(self):
6997:         """Test question_global=-1 is invalid."""
6998:         with pytest.raises(ValueError):
6999:             ExecutableTask(
7000:                 task_id="MQC-001_PA01",
```

```
7001:         question_id="Q-1",
7002:         question_global=-1,
7003:         policy_area_id="PA01",
7004:         dimension_id="DIM01",
7005:         chunk_id="chunk_001",
7006:         patterns=[],
7007:         signals={},
7008:         creation_timestamp="2024-01-01T00:00:00Z",
7009:         expected_elements=[],
7010:         metadata={},
7011:     )
7012:
7013: def test_question_global_just_above_max_invalid(self):
7014:     """Test question_global=MAX_QUESTION_GLOBAL+1 is invalid."""
7015:     with pytest.raises(ValueError):
7016:         ExecutableTask(
7017:             task_id=f"MQC-{MAX_QUESTION_GLOBAL+1:03d}_PA01",
7018:             question_id=f"Q{MAX_QUESTION_GLOBAL+1}",
7019:             question_global=MAX_QUESTION_GLOBAL + 1,
7020:             policy_area_id="PA01",
7021:             dimension_id="DIM01",
7022:             chunk_id="chunk_001",
7023:             patterns=[],
7024:             signals={},
7025:             creation_timestamp="2024-01-01T00:00:00Z",
7026:             expected_elements=[],
7027:             metadata={},
7028:         )
7029:
7030: def test_construct_task_boundaries_validated(self):
7031:     """Test _construct_task validates question_global boundaries."""
7032:     # Test minimum valid
7033:     question_min = {
7034:         "question_id": "Q0",
7035:         "question_global": 0,
7036:         "dimension_id": "DIM01",
7037:         "base_slot": "Q0",
7038:         "cluster_id": "CL01",
7039:         "expected_elements": [],
7040:     }
7041:     routing_result = create_test_chunk_routing_result()
7042:     generated_task_ids: set[str] = set()
7043:
7044:     task_min = _construct_task(
7045:         question_min,
7046:         routing_result,
7047:         (),
7048:         (),
7049:         generated_task_ids,
7050:         "corr-id",
7051:     )
7052:     assert task_min.question_global == 0
7053:
7054:     # Test maximum valid
7055:     question_max = {
7056:         "question_id": f"Q{MAX_QUESTION_GLOBAL}",
```

```
7057:         "question_global": MAX_QUESTION_GLOBAL,
7058:         "dimension_id": "DIM01",
7059:         "base_slot": f"Q{MAX_QUESTION_GLOBAL}",
7060:         "cluster_id": "CL01",
7061:         "expected_elements": [],
7062:     }
7063:     generated_task_ids_max: set[str] = set()
7064:
7065:     task_max = _construct_task(
7066:         question_max,
7067:         routing_result,
7068:         (),
7069:         (),
7070:         generated_task_ids_max,
7071:         "corr-id",
7072:     )
7073:     assert task_max.question_global == MAX_QUESTION_GLOBAL
7074:
7075:     # Test below minimum invalid
7076:     question_below = {
7077:         "question_id": "Q-1",
7078:         "question_global": -1,
7079:         "dimension_id": "DIM01",
7080:         "expected_elements": [],
7081:     }
7082:     generated_task_ids_below: set[str] = set()
7083:
7084:     with pytest.raises(ValueError, match="question_global must be in range"):
7085:         _construct_task(
7086:             question_below,
7087:             routing_result,
7088:             (),
7089:             (),
7090:             generated_task_ids_below,
7091:             "corr-id",
7092:         )
7093:
7094:     # Test above maximum invalid
7095:     question_above = {
7096:         "question_id": f"Q{MAX_QUESTION_GLOBAL+1}",
7097:         "question_global": MAX_QUESTION_GLOBAL + 1,
7098:         "dimension_id": "DIM01",
7099:         "expected_elements": [],
7100:     }
7101:     generated_task_ids_above: set[str] = set()
7102:
7103:     with pytest.raises(ValueError, match="question_global must be in range"):
7104:         _construct_task(
7105:             question_above,
7106:             routing_result,
7107:             (),
7108:             (),
7109:             generated_task_ids_above,
7110:             "corr-id",
7111:         )
7112:
```

```
7113:
7114: class TestPhase7TaskIDGeneration:
7115:     """Test task ID generation patterns and consistency."""
7116:
7117:     def test_task_id_format_mqc_prefix(self):
7118:         """Test task_id starts with MQC- prefix."""
7119:         task = ExecutableTask(
7120:             task_id="MQC-001_PA01",
7121:             question_id="Q1",
7122:             question_global=1,
7123:             policy_area_id="PA01",
7124:             dimension_id="DIM01",
7125:             chunk_id="chunk_001",
7126:             patterns=[],
7127:             signals={},
7128:             creation_timestamp="2024-01-01T00:00:00Z",
7129:             expected_elements=[],
7130:             metadata={},
7131:         )
7132:         assert task.task_id.startswith("MQC-")
7133:
7134:     def test_task_id_format_zero_padded(self):
7135:         """Test task_id has zero-padded question number."""
7136:         test_cases = [
7137:             (1, "MQC-001_PA01"),
7138:             (10, "MQC-010_PA01"),
7139:             (100, "MQC-100_PA01"),
7140:             (999, "MQC-999_PA01"),
7141:         ]
7142:
7143:         for question_global, expected_prefix in test_cases:
7144:             task = ExecutableTask(
7145:                 task_id=expected_prefix,
7146:                 question_id=f"Q{question_global}",
7147:                 question_global=question_global,
7148:                 policy_area_id="PA01",
7149:                 dimension_id="DIM01",
7150:                 chunk_id="chunk_001",
7151:                 patterns=[],
7152:                 signals={},
7153:                 creation_timestamp="2024-01-01T00:00:00Z",
7154:                 expected_elements=[],
7155:                 metadata={},
7156:             )
7157:             assert task.task_id == expected_prefix
7158:
7159:     def test_task_id_format_includes_policy_area(self):
7160:         """Test task_id includes policy area ID."""
7161:         test_cases = [
7162:             ("PA01", "MQC-001_PA01"),
7163:             ("PA05", "MQC-001_PA05"),
7164:             ("PA10", "MQC-001_PA10"),
7165:         ]
7166:
7167:         for policy_area_id, expected_task_id in test_cases:
7168:             task = ExecutableTask(
```



```
7169:         task_id=expected_task_id,
7170:         question_id="Q1",
7171:         question_global=1,
7172:         policy_area_id=policy_area_id,
7173:         dimension_id="DIM01",
7174:         chunk_id="chunk_001",
7175:         patterns=[],
7176:         signals={},
7177:         creation_timestamp="2024-01-01T00:00:00Z",
7178:         expected_elements=[],
7179:         metadata={},
7180:     )
7181:     assert task.task_id == expected_task_id
7182:     assert policy_area_id in task.task_id
7183:
7184: def test_construct_task_generates_consistent_task_id(self):
7185:     """Test _construct_task generates consistent task_id from inputs."""
7186:     question = {
7187:         "question_id": "D2-Q25",
7188:         "question_global": 75,
7189:         "dimension_id": "DIM02",
7190:         "base_slot": "D2-Q25",
7191:         "cluster_id": "CL02",
7192:         "expected_elements": [],
7193:     }
7194:     routing_result = create_test_chunk_routing_result(policy_area_id="PA07")
7195:     generated_task_ids: set[str] = set()
7196:
7197:     task1 = _construct_task(
7198:         question,
7199:         routing_result,
7200:         (),
7201:         (),
7202:         generated_task_ids,
7203:         "corr-id",
7204:     )
7205:
7206:     # Reset and create again
7207:     generated_task_ids2: set[str] = set()
7208:     task2 = _construct_task(
7209:         question,
7210:         routing_result,
7211:         (),
7212:         (),
7213:         generated_task_ids2,
7214:         "corr-id",
7215:     )
7216:
7217:     assert task1.task_id == task2.task_id == "MQC-075_PA07"
7218:
7219:
7220: class TestPhase7ProvenanceTracking:
7221:     """Test provenance and metadata tracking in task construction."""
7222:
7223:     def test_metadata_includes_base_slot(self):
7224:         """Test metadata includes base_slot from question."""
```

```
7225:         question = {
7226:             "question_id": "D1-Q1",
7227:             "question_global": 1,
7228:             "dimension_id": "DIM01",
7229:             "base_slot": "D1-Q1-SLOT",
7230:             "cluster_id": "CL01",
7231:             "expected_elements": [],
7232:         }
7233:         routing_result = create_test_chunk_routing_result()
7234:         generated_task_ids: set[str] = set()
7235:
7236:         task = _construct_task(
7237:             question,
7238:             routing_result,
7239:             (),
7240:             (),
7241:             generated_task_ids,
7242:             "corr-id",
7243:         )
7244:
7245:         assert task.metadata["base_slot"] == "D1-Q1-SLOT"
7246:
7247:     def test_metadata_includes_cluster_id(self):
7248:         """Test metadata includes cluster_id from question."""
7249:         question = {
7250:             "question_id": "D1-Q1",
7251:             "question_global": 1,
7252:             "dimension_id": "DIM01",
7253:             "base_slot": "D1-Q1",
7254:             "cluster_id": "CLUSTER-ABC",
7255:             "expected_elements": [],
7256:         }
7257:         routing_result = create_test_chunk_routing_result()
7258:         generated_task_ids: set[str] = set()
7259:
7260:         task = _construct_task(
7261:             question,
7262:             routing_result,
7263:             (),
7264:             (),
7265:             generated_task_ids,
7266:             "corr-id",
7267:         )
7268:
7269:         assert task.metadata["cluster_id"] == "CLUSTER-ABC"
7270:
7271:     def test_metadata_includes_correlation_id(self):
7272:         """Test metadata includes correlation_id for tracing."""
7273:         question = {
7274:             "question_id": "D1-Q1",
7275:             "question_global": 1,
7276:             "dimension_id": "DIM01",
7277:             "base_slot": "D1-Q1",
7278:             "cluster_id": "CL01",
7279:             "expected_elements": [],
7280:         }
```

```
7281:         routing_result = create_test_chunk_routing_result()
7282:         generated_task_ids: set[str] = set()
7283:         correlation_id = "CORRELATION-XYZ-789"
7284:
7285:         task = _construct_task(
7286:             question,
7287:             routing_result,
7288:             (),
7289:             (),
7290:             generated_task_ids,
7291:             correlation_id,
7292:         )
7293:
7294:         assert task.metadata["correlation_id"] == "CORRELATION-XYZ-789"
7295:
7296:     def test_metadata_includes_document_position(self):
7297:         """Test metadata includes document_position from routing_result."""
7298:         question = {
7299:             "question_id": "D1-Q1",
7300:             "question_global": 1,
7301:             "dimension_id": "DIM01",
7302:             "base_slot": "D1-Q1",
7303:             "cluster_id": "CL01",
7304:             "expected_elements": [],
7305:         }
7306:         routing_result = create_test_chunk_routing_result(document_position=(500, 750))
7307:         generated_task_ids: set[str] = set()
7308:
7309:         task = _construct_task(
7310:             question,
7311:             routing_result,
7312:             (),
7313:             (),
7314:             generated_task_ids,
7315:             "corr-id",
7316:         )
7317:
7318:         assert task.metadata["document_position"] == (500, 750)
7319:
7320:     def test_metadata_tracks_pattern_counts(self):
7321:         """Test metadata tracks original and filtered pattern counts."""
7322:         question = {
7323:             "question_id": "D1-Q1",
7324:             "question_global": 1,
7325:             "dimension_id": "DIM01",
7326:             "base_slot": "D1-Q1",
7327:             "cluster_id": "CL01",
7328:             "expected_elements": [],
7329:         }
7330:         routing_result = create_test_chunk_routing_result()
7331:         applicable_patterns = ({ "p": "1" }, { "p": "2" }, { "p": "3" })
7332:         generated_task_ids: set[str] = set()
7333:
7334:         task = _construct_task(
7335:             question,
7336:             routing_result,
```

```
7337:         applicable_patterns,
7338:         (),
7339:         generated_task_ids,
7340:         "corr-id",
7341:     )
7342:
7343:     assert task.metadata["original_pattern_count"] == 3
7344:     assert task.metadata["filtered_pattern_count"] == 3
7345:
7346:     def test_metadata_tracks_signal_counts(self):
7347:         """Test metadata tracks original and resolved signal counts."""
7348:         question = {
7349:             "question_id": "D1-Q1",
7350:             "question_global": 1,
7351:             "dimension_id": "DIM01",
7352:             "base_slot": "D1-Q1",
7353:             "cluster_id": "CL01",
7354:             "expected_elements": [],
7355:         }
7356:         routing_result = create_test_chunk_routing_result()
7357:         resolved_signals = (
7358:             {"signal_type": "s1", "v": 0.5},
7359:             {"signal_type": "s2", "v": 0.7},
7360:         )
7361:         generated_task_ids: set[str] = set()
7362:
7363:         task = _construct_task(
7364:             question,
7365:             routing_result,
7366:             (),
7367:             resolved_signals,
7368:             generated_task_ids,
7369:             "corr-id",
7370:         )
7371:
7372:         assert task.metadata["original_signal_count"] == 2
7373:         assert task.metadata["resolved_signal_count"] == 2
7374:
7375:     def test_metadata_tracks_schema_element_count(self):
7376:         """Test metadata tracks expected_elements count."""
7377:         expected_elements = [
7378:             {"type": "elem1"},
7379:             {"type": "elem2"},
7380:             {"type": "elem3"},
7381:         ]
7382:         question = {
7383:             "question_id": "D1-Q1",
7384:             "question_global": 1,
7385:             "dimension_id": "DIM01",
7386:             "base_slot": "D1-Q1",
7387:             "cluster_id": "CL01",
7388:             "expected_elements": expected_elements,
7389:         }
7390:         routing_result = create_test_chunk_routing_result()
7391:         generated_task_ids: set[str] = set()
7392:
```

```
7393:         task = _construct_task(
7394:             question,
7395:             routing_result,
7396:             (),
7397:             (),
7398:             generated_task_ids,
7399:             "corr-id",
7400:         )
7401:
7402:         assert task.metadata["schema_element_count"] == 3
7403:
7404:     def test_metadata_includes_synchronizer_version(self):
7405:         """Test metadata includes synchronizer version."""
7406:         question = {
7407:             "question_id": "D1-Q1",
7408:             "question_global": 1,
7409:             "dimension_id": "DIM01",
7410:             "base_slot": "D1-Q1",
7411:             "cluster_id": "CL01",
7412:             "expected_elements": [],
7413:         }
7414:         routing_result = create_test_chunk_routing_result()
7415:         generated_task_ids: set[str] = set()
7416:
7417:         task = _construct_task(
7418:             question,
7419:             routing_result,
7420:             (),
7421:             (),
7422:             generated_task_ids,
7423:             "corr-id",
7424:         )
7425:
7426:         assert "synchronizer_version" in task.metadata
7427:         assert isinstance(task.metadata["synchronizer_version"], str)
7428:
7429:
7430:
7431: =====
7432: FILE: tests/phases/test_phase_boundaries.py
7433: =====
7434:
7435: """Test Phase Boundary Contracts
7436:
7437: Tests that phase N output becomes phase N+1 input with no data loss.
7438: """
7439: import pytest
7440: from unittest.mock import AsyncMock, patch
7441:
7442:
7443: class TestPhaseBoundaries:
7444:     """Test phase boundary contract enforcement."""
7445:
7446:     def test_phase0_output_is_phase1_input(self):
7447:         """Test Phase 0 output (CanonicalInput) is Phase 1 input."""
7448:         from farfan_pipeline.core.phases.phase0_input_validation import Phase0ValidationContract
```

```
7449:         from farfan_pipeline.core.phases.phase1_spc_ingestion import Phase1SPCIngestionContract
7450:
7451:         phase0 = Phase0ValidationContract()
7452:         phase1 = Phase1SPCIngestionContract()
7453:
7454:         assert phase0.phase_name == "phase0_input_validation"
7455:         assert phase1.phase_name == "phase1_spc_ingestion"
7456:
7457:     def test_phase1_output_is_adapter_input(self):
7458:         """Test Phase 1 output (CanonPolicyPackage) is adapter input."""
7459:         from farfan_pipeline.core.phases.phase1_spc_ingestion import Phase1SPCIngestionContract
7460:         from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract
7461:
7462:         phase1 = Phase1SPCIngestionContract()
7463:         adapter = AdapterContract()
7464:
7465:         assert adapter.phase_name == "phase1_to_phase2_adapter"
7466:
7467:     def test_adapter_output_is_phase2_input(self):
7468:         """Test adapter output (PreprocessedDocument) is Phase 2 input."""
7469:         from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract
7470:
7471:         adapter = AdapterContract()
7472:         assert "phase2" in adapter.phase_name.lower() or "adapter" in adapter.phase_name.lower()
7473:
7474:     def test_no_phase_can_be_skipped(self):
7475:         """Test orchestrator enforces sequential execution."""
7476:         from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
7477:
7478:         orchestrator = PhaseOrchestrator()
7479:         assert hasattr(orchestrator, 'phase0')
7480:         assert hasattr(orchestrator, 'phase1')
7481:         assert hasattr(orchestrator, 'adapter')
7482:
7483:     @pytest.mark.asyncio
7484:     async def test_phase_contract_validates_input_output(self):
7485:         """Test PhaseContract.run() validates input and output."""
7486:         from farfan_pipeline.core.phases.phase_protocol import PhaseContract
7487:
7488:         class TestContract(PhaseContract):
7489:             def validate_input(self, data):
7490:                 from farfan_pipeline.core.phases.phase_protocol import ContractValidationResult
7491:                 return ContractValidationResult(True, "input", "test")
7492:
7493:             def validate_output(self, data):
7494:                 from farfan_pipeline.core.phases.phase_protocol import ContractValidationResult
7495:                 return ContractValidationResult(True, "output", "test")
7496:
7497:             async def execute(self, data):
7498:                 return data
7499:
7500:         contract = TestContract("test")
7501:         output, metadata = await contract.run("test_input")
7502:         assert output == "test_input"
7503:         assert metadata.success is True
7504:
```

```
7505:
7506:
7507: =====
7508: FILE: tests/phases/test_provenance_completeness.py
7509: =====
7510:
7511: """Test Provenance Completeness Verification
7512:
7513: Tests that all chunks have provenance_completeness=1.0 or near 1.0.
7514: """
7515: import pytest
7516: import hashlib
7517:
7518:
7519: class TestProvenanceCompleteness:
7520:     """Test provenance completeness for all chunks."""
7521:
7522:     def test_provenance_threshold_in_quality_metrics(self):
7523:         """Test QualityMetrics enforces provenance_completeness >= 0.8."""
7524:         from farfan_pipeline.processing.models import QualityMetrics
7525:
7526:         metrics = QualityMetrics(
7527:             provenance_completeness=0.9,
7528:             structural_consistency=0.9
7529:         )
7530:         assert metrics.provenance_completeness >= 0.8
7531:
7532:     def test_phase1_invariant_checks_provenance(self):
7533:         """Test Phase 1 has provenance threshold invariant."""
7534:         from farfan_pipeline.core.phases.phase1_spc_ingestion import Phase1SPCIngestionContract
7535:
7536:         contract = Phase1SPCIngestionContract()
7537:         invariant_names = [inv.name for inv in contract.invariants]
7538:         assert "provenance_threshold" in invariant_names
7539:
7540:     def test_chunk_has_provenance_field(self):
7541:         """Test Chunk model has provenance field."""
7542:         from farfan_pipeline.processing.models import Chunk, ChunkResolution, TextSpan, ProvenanceMap
7543:
7544:         chunk = Chunk(
7545:             id="test", text="test", text_span=TextSpan(0, 4),
7546:             resolution=ChunkResolution.MESO,
7547:             bytes_hash=hashlib.blake2b(b"test").hexdigest(),
7548:             provenance=ProvenanceMap(source_page=1, source_section="intro")
7549:         )
7550:         assert chunk.provenance is not None
7551:         assert chunk.provenance.source_page == 1
7552:
7553:     def test_provenance_map_structure(self):
7554:         """Test ProvenanceMap has required fields."""
7555:         from farfan_pipeline.processing.models import ProvenanceMap
7556:
7557:         prov = ProvenanceMap(
7558:             source_page=5,
7559:             source_section="Section 2.1",
7560:             extraction_method="semantic_chunking"
```

```
7561:     )
7562:     assert prov.source_page == 5
7563:     assert prov.source_section == "Section 2.1"
7564:     assert prov.extraction_method == "semantic_chunking"
7565:
7566: def test_cpp_quality_metrics_has_provenance_completeness(self):
7567:     """Test CanonPolicyPackage tracks provenance_completeness."""
7568:     from farfan_pipeline.processing.models import (
7569:         CanonPolicyPackage, ChunkGraph, QualityMetrics,
7570:         IntegrityIndex, PolicyManifest
7571:     )
7572:
7573:     cpp = CanonPolicyPackage(
7574:         schema_version="SPC-2025.1",
7575:         chunk_graph=ChunkGraph(),
7576:         quality_metrics=QualityMetrics(
7577:             provenance_completeness=1.0,
7578:             structural_consistency=0.95
7579:         ),
7580:         integrity_index=IntegrityIndex(blake2b_root="a"*64),
7581:         metadata={}
7582:     )
7583:     assert cpp.quality_metrics.provenance_completeness == 1.0
7584:
7585: def test_all_chunks_should_have_provenance(self):
7586:     """Test that chunks in a complete CPP have provenance data."""
7587:     from farfan_pipeline.processing.models import (
7588:         CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
7589:         TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest,
7590:         ProvenanceMap
7591:     )
7592:     from farfan_pipeline.core.phases.phase1_spc_ingestion import POLICY_AREAS, DIMENSIONS
7593:
7594:     chunk_graph = ChunkGraph()
7595:     chunks_with_provenance = 0
7596:     total_chunks = 0
7597:
7598:     for i, pa in enumerate(POLICY_AREAS):
7599:         for j, dim in enumerate(DIMENSIONS):
7600:             chunk = Chunk(
7601:                 id=f"c_{pa}_{dim}",
7602:                 text=f"Test chunk {i*6+j}",
7603:                 text_span=TextSpan(i*100, i*100+50),
7604:                 resolution=ChunkResolution.MESO,
7605:                 bytes_hash=hashlib.blake2b(f"t{i}{j}".encode()).hexdigest(),
7606:                 policy_area_id=pa,
7607:                 dimension_id=dim,
7608:                 provenance=ProvenanceMap(
7609:                     source_page=i+1,
7610:                     source_section=f"Section {i+1}"
7611:                 )
7612:             )
7613:             chunk_graph.add_chunk(chunk)
7614:             total_chunks += 1
7615:             if chunk.provenance is not None:
7616:                 chunks_with_provenance += 1
```



```
7617:
7618:     provenance_completeness = chunks_with_provenance / total_chunks
7619:     assert provenance_completeness == 1.0
7620:     assert total_chunks == 60
7621:
7622: def test_provenance_completeness_calculation(self):
7623:     """Test provenance_completeness metric calculation."""
7624:     total_chunks = 60
7625:     chunks_with_provenance = 60
7626:
7627:     provenance_completeness = chunks_with_provenance / total_chunks
7628:     assert provenance_completeness == 1.0
7629:
7630:     chunks_with_provenance = 48
7631:     provenance_completeness = chunks_with_provenance / total_chunks
7632:     assert provenance_completeness == 0.8
7633:
7634: def test_phase1_validates_provenance_completeness(self):
7635:     """Test Phase 1 validates provenance_completeness >= 0.8."""
7636:     from farfan_pipeline.processing.models import (
7637:         CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
7638:         TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest
7639:     )
7640:     from farfan_pipeline.core.phases.phase1_spc_ingestion import (
7641:         Phase1SPCIngestionContract, POLICY_AREAS, DIMENSIONS
7642:     )
7643:
7644:     contract = Phase1SPCIngestionContract()
7645:     chunk_graph = ChunkGraph()
7646:
7647:     for i, pa in enumerate(POLICY_AREAS):
7648:         for j, dim in enumerate(DIMENSIONS):
7649:             chunk = Chunk(
7650:                 id=f"c_{pa}_{dim}",
7651:                 text="t",
7652:                 text_span=TextSpan(0, 1),
7653:                 resolution=ChunkResolution.MESO,
7654:                 bytes_hash=hashlib.blake2b(f"t{i}{j}".encode()).hexdigest(),
7655:                 policy_area_id=pa,
7656:                 dimension_id=dim
7657:             )
7658:             chunk_graph.add_chunk(chunk)
7659:
7660:     cpp_low = CanonPolicyPackage(
7661:         schema_version="SPC-2025.1",
7662:         chunk_graph=chunk_graph,
7663:         quality_metrics=QualityMetrics(
7664:             provenance_completeness=0.5,
7665:             structural_consistency=0.9
7666:         ),
7667:         integrity_index=IntegrityIndex(blake2b_root="a"*64),
7668:         metadata={"document_id": "test"}
7669:     )
7670:
7671:     result = contract.validate_output(cpp_low)
7672:     assert not result.passed
```

```
7673:
7674:     cpp_high = CanonPolicyPackage(
7675:         schema_version="SPC-2025.1",
7676:         chunk_graph=chunk_graph,
7677:         quality_metrics=QualityMetrics(
7678:             provenance_completeness=0.95,
7679:             structural_consistency=0.9
7680:         ),
7681:         integrity_index=IntegrityIndex(blake2b_root="a"*64),
7682:         metadata={"document_id": "test"}
7683:     )
7684:
7685:     result = contract.validate_output(cpp_high)
7686:     assert result.passed
7687:
7688:
```