

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 28
3: =====
4: Generated: 2025-12-07T06:17:36.486739
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: tests/processing/__init__.py
11: =====
12:
13: """
14: Processing Layer Test Suite
15:
16: Property-based tests using Hypothesis for semantic chunking determinism,
17: SPC ingestion pipeline idempotency, embedding policy consistency, and converter invariants.
18: Plus integration tests for quality gates, policy processor end-to-end execution,
19: and aggregation pipeline correctness.
20:
21: Test Files:
22: - test_semantic_chunking_properties.py: Semantic chunking determinism
23: - test_spc_ingestion_properties.py: SPC ingestion idempotency
24: - test_embedding_policy_properties.py: Embedding consistency
25: - test_converter_properties.py: Converter invariants (to be created)
26: - test_quality_gates_integration.py: Quality gates integration (to be created)
27: - test_policy_processor_integration.py: End-to-end policy processing (to be created)
28: - test_aggregation_integration.py: Aggregation pipeline (to be created)
29: """
30:
31:
32:
33: =====
34: FILE: tests/processing/test_embedding_policy_properties.py
35: =====
36:
37: """
38: Property-Based Tests for Embedding Policy Consistency
39:
40: Verifies that embedding generation is consistent, deterministic, and maintains
41: mathematical properties expected from embeddings.
42: """
43:
44: import numpy as np
45: import pytest
46: from hypothesis import HealthCheck, assume, given, settings
47: from hypothesis import strategies as st
48: from scipy.spatial.distance import cosine
49:
50: from src.farfan_pipeline.processing.embedding_policy import (
51:     AdvancedSemanticChunker,
52:     BayesianNumericalAnalyzer,
53:     ChunkingConfig,
54: )
55:
56:
```

```
57: @st.composite
58: def valid_text(draw):
59:     """Generate valid text for embedding."""
60:     return draw(st.text(
61:         alphabet=st.characters(whitelist_categories=('L', 'N', 'P', 'Z')),
62:         min_size=20,
63:         max_size=500
64:     ))
65:
66:
67: @st.composite
68: def chunking_config(draw):
69:     """Generate valid ChunkingConfig instances."""
70:     chunk_size = draw(st.integers(min_value=128, max_value=1024))
71:     return ChunkingConfig(
72:         chunk_size=chunk_size,
73:         chunk_overlap=draw(st.integers(min_value=16, max_value=chunk_size // 2)),
74:         min_chunk_size=draw(st.integers(min_value=32, max_value=chunk_size // 4)),
75:         respect_boundaries=draw(st.booleans()),
76:         preserve_tables=draw(st.booleans()),
77:         detect_lists=draw(st.booleans()),
78:         section_aware=draw(st.booleans())
79:     )
80:
81:
82: class TestEmbeddingConsistency:
83:     """Property-based tests for embedding consistency."""
84:
85:     @given(text=valid_text())
86:     @settings(max_examples=20, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
87:     def test_embedding_dimensions_are_consistent(self, text):
88:         """Property: All embeddings have consistent dimensions."""
89:         assume(len(text) > 50)
90:
91:         config = ChunkingConfig()
92:         chunker = AdvancedSemanticChunker(config)
93:
94:         chunks = chunker.chunk_document(
95:             text=text,
96:             document_metadata={"doc_id": "test_doc"}
97:         )
98:
99:         if len(chunks) > 0 and len(chunks[0].get("embedding", [])) > 0:
100:             expected_dim = len(chunks[0]["embedding"])
101:
102:             for chunk in chunks:
103:                 if len(chunk.get("embedding", [])) > 0:
104:                     assert len(chunk["embedding"]) == expected_dim, \
105:                         "All embeddings must have same dimension"
106:
107:             @given(
108:                 text=valid_text(),
109:                 seed=st.integers(min_value=0, max_value=1000)
110:             )
111:             @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
112:             def test_embeddings_are_deterministic(self, text, seed):
```

```
113:     """Property: Same text produces same embeddings."""
114:     assume(len(text) > 50)
115:
116:     np.random.seed(seed)
117:     config = ChunkingConfig()
118:     chunker1 = AdvancedSemanticChunker(config)
119:     chunks1 = chunker1.chunk_document(
120:         text=text,
121:         document_metadata={"doc_id": "test_doc"}
122:     )
123:
124:     np.random.seed(seed)
125:     chunker2 = AdvancedSemanticChunker(config)
126:     chunks2 = chunker2.chunk_document(
127:         text=text,
128:         document_metadata={"doc_id": "test_doc"}
129:     )
130:
131:     assert len(chunks1) == len(chunks2)
132:
133:     for c1, c2 in zip(chunks1, chunks2, strict=False):
134:         emb1 = c1.get("embedding", [])
135:         emb2 = c2.get("embedding", [])
136:
137:         if len(emb1) > 0 and len(emb2) > 0:
138:             np.testing.assert_array_almost_equal(emb1, emb2, decimal=4)
139:
140:     @given(text=valid_text())
141:     @settings(max_examples=15, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
142:     def test_embeddings_are_normalized(self, text):
143:         """Property: Embeddings should be unit vectors or close to it."""
144:         assume(len(text) > 50)
145:
146:         config = ChunkingConfig()
147:         chunker = AdvancedSemanticChunker(config)
148:
149:         chunks = chunker.chunk_document(
150:             text=text,
151:             document_metadata={"doc_id": "test_doc"}
152:         )
153:
154:         for chunk in chunks:
155:             emb = chunk.get("embedding", [])
156:             if len(emb) > 0:
157:                 norm = np.linalg.norm(emb)
158:                 assert 0.8 <= norm <= 1.2, \
159:                     f"Embedding norm should be close to 1: {norm}"
160:
161:     @given(
162:         text1=valid_text(),
163:         text2=valid_text()
164:     )
165:     @settings(max_examples=10, deadline=6000, suppress_health_check=[HealthCheck.too_slow])
166:     def test_similar_texts_have_similar_embeddings(self, text1, text2):
167:         """Property: Similar texts have similar embeddings."""
168:         assume(len(text1) > 50 and len(text2) > 50)
```

```
169:
170:     words1 = set(text1.lower().split())
171:     words2 = set(text2.lower().split())
172:
173:     if len(words1) > 0 and len(words2) > 0:
174:         overlap = len(words1 & words2) / max(len(words1), len(words2))
175:         assume(overlap > 0.5)
176:
177:         config = ChunkingConfig()
178:         chunker = AdvancedSemanticChunker(config)
179:
180:         chunks1 = chunker.chunk_document(
181:             text=text1,
182:             document_metadata={"doc_id": "doc1"})
183:         )
184:         chunks2 = chunker.chunk_document(
185:             text=text2,
186:             document_metadata={"doc_id": "doc2"})
187:         )
188:
189:     if len(chunks1) > 0 and len(chunks2) > 0:
190:         emb1 = chunks1[0].get("embedding", [])
191:         emb2 = chunks2[0].get("embedding", [])
192:
193:         if len(emb1) > 0 and len(emb2) > 0:
194:             similarity = 1 - cosine(emb1, emb2)
195:             assert similarity > 0.2, \
196:                 f"Similar texts should have similar embeddings: {similarity}"
197:
198:
199: class TestChunkingConfigurationInvariants:
200:     """Invariant tests for chunking configuration."""
201:
202:     @given(config=chunking_config())
203:     @settings(max_examples=30, deadline=1000)
204:     def test_config_invariants_hold(self, config):
205:         """Property: Configuration parameters maintain valid relationships."""
206:         assert config.chunk_size > 0, "Chunk size must be positive"
207:         assert config.chunk_overlap >= 0, "Overlap must be non-negative"
208:         assert config.chunk_overlap < config.chunk_size, \
209:             "Overlap must be less than chunk size"
210:         assert config.min_chunk_size > 0, "Min chunk size must be positive"
211:         assert config.min_chunk_size <= config.chunk_size, \
212:             "Min chunk size must not exceed max chunk size"
213:
214:     @given(
215:         text=valid_text(),
216:         config=chunking_config()
217:     )
218:     @settings(max_examples=10, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
219:     def test_chunker_respects_config(self, text, config):
220:         """Property: Chunker respects configuration parameters."""
221:         assume(len(text) > config.chunk_size)
222:
223:         chunker = AdvancedSemanticChunker(config)
```

```
225:     chunks = chunker.chunk_document(
226:         text=text,
227:         document_metadata={"doc_id": "test_doc"}
228:     )
229:
230:     for chunk in chunks:
231:         token_count = chunk.get("token_count", 0)
232:
233:         if config.min_chunk_size:
234:             assert token_count >= config.min_chunk_size * 0.5 or len(chunks) == 1, \
235:                 f"Chunk smaller than min size: {token_count} < {config.min_chunk_size}"
236:
237:
238: class TestBayesianNumericalAnalyzerProperties:
239:     """Property-based tests for Bayesian numerical analysis."""
240:
241:     @given(
242:         values=st.lists(st.floats(min_value=0.0, max_value=1.0), min_size=1, max_size=50),
243:         prior_alpha=st.floats(min_value=0.1, max_value=10.0),
244:         prior_beta=st.floats(min_value=0.1, max_value=10.0)
245:     )
246:     @settings(max_examples=50, deadline=1000)
247:     def test_bayesian_evaluation_produces_valid_credible_intervals(self, values, prior_alpha, prior_beta):
248:         """Property: Credible intervals are valid."""
249:         analyzer = BayesianNumericalAnalyzer(prior_alpha=prior_alpha, prior_beta=prior_beta)
250:
251:         result = analyzer.analyze_proportion(
252:             successes=sum(values),
253:             trials=len(values),
254:             context="test"
255:         )
256:
257:         lower, upper = result["credible_interval_95"]
258:
259:         assert 0.0 <= lower <= 1.0, f"Lower bound out of range: {lower}"
260:         assert 0.0 <= upper <= 1.0, f"Upper bound out of range: {upper}"
261:         assert lower <= upper, f"Lower bound exceeds upper bound: {lower} > {upper}"
262:         assert lower <= result["point_estimate"] <= upper, \
263:             "Point estimate must be within credible interval"
264:
265:     @given(
266:         successes=st.integers(min_value=0, max_value=100),
267:         trials=st.integers(min_value=1, max_value=100)
268:     )
269:     @settings(max_examples=50, deadline=1000)
270:     def test_bayesian_evaluation_is_bounded(self, successes, trials):
271:         """Property: Bayesian evaluation produces bounded results."""
272:         assume(successes <= trials)
273:
274:         analyzer = BayesianNumericalAnalyzer()
275:
276:         result = analyzer.analyze_proportion(
277:             successes=successes,
278:             trials=trials,
279:             context="test"
280:         )
```

```
281:
282:     assert 0.0 <= result["point_estimate"] <= 1.0
283:     assert result["evidence_strength"] in ["weak", "moderate", "strong", "very_strong"]
284:     assert 0.0 <= result["numerical_coherence"] <= 1.0
285:
286:     @given(
287:         successes=st.integers(min_value=0, max_value=100),
288:         trials=st.integers(min_value=1, max_value=100)
289:     )
290:     @settings(max_examples=30, deadline=1000)
291:     def test_more_data_reduces_uncertainty(self, successes, trials):
292:         """Property: More data should reduce uncertainty (narrower intervals)."""
293:         assume(successes <= trials and trials >= 10)
294:
295:         analyzer = BayesianNumericalAnalyzer()
296:
297:         result_few = analyzer.analyze_proportion(
298:             successes=successes // 2,
299:             trials=trials // 2,
300:             context="test"
301:         )
302:
303:         result_many = analyzer.analyze_proportion(
304:             successes=successes,
305:             trials=trials,
306:             context="test"
307:         )
308:
309:         width_few = result_few["credible_interval_95"][1] - result_few["credible_interval_95"][0]
310:         width_many = result_many["credible_interval_95"][1] - result_many["credible_interval_95"][0]
311:
312:         assert width_many <= width_few * 1.5, \
313:             "More data should not significantly increase uncertainty"
314:
315:
316: class TestPDQContextInference:
317:     """Property-based tests for PDQ context inference."""
318:
319:     @given(text=valid_text())
320:     @settings(max_examples=15, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
321:     def test_pdq_context_structure_is_valid(self, text):
322:         """Property: PDQ context has valid structure when inferred."""
323:         assume(len(text) > 50)
324:
325:         config = ChunkingConfig()
326:         chunker = AdvancedSemanticChunker(config)
327:
328:         chunks = chunker.chunk_document(
329:             text=text,
330:             document_metadata={"doc_id": "test_doc"}
331:         )
332:
333:         for chunk in chunks:
334:             pdq = chunk.get("pdq_context")
335:             if pdq is not None:
336:                 assert "question_unique_id" in pdq
```

```
337:             assert "policy" in pdq
338:             assert "dimension" in pdq
339:             assert "question" in pdq
340:             assert "rubric_key" in pdq
341:
342:             assert pdq["policy"].startswith("PA")
343:             assert pdq["dimension"].startswith("DIM")
344:             assert isinstance(pdq["question"], int)
345:
346:     @given(text=valid_text())
347:     @settings(max_examples=10, deadline=3000, suppress_health_check=[HealthCheck.too_slow])
348:     def test_pdq_inference_is_stable(self, text):
349:         """Property: PDQ inference is stable across runs."""
350:         assume(len(text) > 50)
351:
352:         config = ChunkingConfig()
353:         chunker1 = AdvancedSemanticChunker(config)
354:         chunker2 = AdvancedSemanticChunker(config)
355:
356:         chunks1 = chunker1.chunk_document(
357:             text=text,
358:             document_metadata={"doc_id": "test_doc"})
359:     )
360:         chunks2 = chunker2.chunk_document(
361:             text=text,
362:             document_metadata={"doc_id": "test_doc"})
363:     )
364:
365:     for c1, c2 in zip(chunks1, chunks2, strict=False):
366:         pdq1 = c1.get("pdq_context")
367:         pdq2 = c2.get("pdq_context")
368:
369:         if pdq1 is not None and pdq2 is not None:
370:             assert pdq1["policy"] == pdq2["policy"]
371:             assert pdq1["dimension"] == pdq2["dimension"]
372:
373:
374: if __name__ == "__main__":
375:     pytest.main([__file__, "-v", "--tb=short"])
376:
377:
378:
379: =====
380: FILE: tests/processing/test_semantic_chunking_properties.py
381: =====
382:
383: """
384: Property-Based Tests for Semantic Chunking Determinism
385:
386: Uses Hypothesis to verify that semantic chunking is deterministic, consistent,
387: and maintains correctness properties across different inputs.
388: """
389:
390: import hashlib
391: import re
392:
```

```
393: import numpy as np
394: import pytest
395: from hypothesis import HealthCheck, assume, given, settings
396: from hypothesis import strategies as st
397:
398: from src.farfan_pipeline.processing.semantic_chunking_policy import (
399:     BayesianEvidenceIntegrator,
400:     SemanticConfig,
401:     SemanticProcessor,
402: )
403:
404:
405: @st.composite
406: def spanish_policy_text(draw):
407:     """Generate realistic Spanish policy text for testing."""
408:     paragraphs = draw(st.lists(
409:         st.text(
410:             alphabet=st.characters(whitelist_categories=('L', 'N', 'P', 'Z'), max_codepoint=255),
411:             min_size=50,
412:             max_size=500
413:         ),
414:             min_size=1,
415:             max_size=20
416:     ))
417:     return "\n\n".join(p.strip() for p in paragraphs if p.strip())
418:
419:
420: @st.composite
421: def chunking_config_strategy(draw):
422:     """Generate valid SemanticConfig instances."""
423:     chunk_size = draw(st.integers(min_value=256, max_value=1024))
424:     chunk_overlap = draw(st.integers(min_value=32, max_value=chunk_size // 2))
425:     return SemanticConfig(
426:         chunk_size=chunk_size,
427:         chunk_overlap=chunk_overlap,
428:         similarity_threshold=draw(st.floats(min_value=0.5, max_value=0.95)),
429:         min_evidence_chunks=draw(st.integers(min_value=1, max_value=5)),
430:         bayesian_prior_strength=draw(st.floats(min_value=0.1, max_value=1.0)),
431:         device="cpu",
432:         batch_size=8,
433:         fp16=False
434:     )
435:
436:
437: class TestSemanticChunkingDeterminism:
438:     """Property-based tests for semantic chunking determinism."""
439:
440:     @given(text=spanish_policy_text())
441:     @settings(max_examples=20, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
442:     def test_chunking_is_deterministic(self, text):
443:         """Property: Chunking the same text twice produces identical results."""
444:         assume(len(text) > 100)
445:
446:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
447:         processor = SemanticProcessor(config)
448:
```

```
449:     chunks1 = processor.chunk_text(text, preserve_structure=False)
450:     chunks2 = processor.chunk_text(text, preserve_structure=False)
451:
452:     assert len(chunks1) == len(chunks2), "Chunk count must be deterministic"
453:
454:     for c1, c2 in zip(chunks1, chunks2, strict=False):
455:         assert c1["content"] == c2["content"], "Chunk content must be identical"
456:         assert c1["token_count"] == c2["token_count"], "Token counts must match"
457:
458:     @given(text=spanish_policy_text())
459:     @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
460:     def test_chunks_preserve_all_text(self, text):
461:         """Property: All input text appears in output chunks (no loss)."""
462:         assume(len(text) > 100)
463:
464:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
465:         processor = SemanticProcessor(config)
466:
467:         chunks = processor.chunk_text(text, preserve_structure=False)
468:
469:         reconstructed = " ".join(c["content"] for c in chunks)
470:
471:         important_words = re.findall(r'\b\w{4,}\b', text.lower())[:20]
472:         for word in important_words:
473:             assert word in reconstructed.lower(), f"Word '{word}' lost during chunking"
474:
475:     @given(text=spanish_policy_text())
476:     @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
477:     def test_chunk_boundaries_are_coherent(self, text):
478:         """Property: Chunks don't split words or create malformed fragments."""
479:         assume(len(text) > 100)
480:
481:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
482:         processor = SemanticProcessor(config)
483:
484:         chunks = processor.chunk_text(text, preserve_structure=False)
485:
486:         for chunk in chunks:
487:             content = chunk["content"]
488:             assert content.strip() == content or len(content) < 10, \
489:                 "Chunks should be trimmed or very short"
490:
491:             assert not content.startswith(' ' * 5), "No excessive leading whitespace"
492:             assert not content.endswith(' ' * 5), "No excessive trailing whitespace"
493:
494:     @given(
495:         text=spanish_policy_text(),
496:         config=chunking_config_strategy()
497:     )
498:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
499:     def test_chunk_sizes_respect_config(self, text, config):
500:         """Property: Chunks respect configured size limits."""
501:         assume(len(text) > config.chunk_size)
502:
503:         processor = SemanticProcessor(config)
504:         chunks = processor.chunk_text(text, preserve_structure=False)
```

```
505:
506:     for chunk in chunks:
507:         token_count = chunk["token_count"]
508:         assert token_count <= config.chunk_size * 1.5, \
509:             f"Chunk exceeds size limit: {token_count} > {config.chunk_size * 1.5}"
510:
511:     @given(text=spanish_policy_text())
512:     @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
513:     def test_chunks_have_required_fields(self, text):
514:         """Property: All chunks have required fields with correct types."""
515:         assume(len(text) > 100)
516:
517:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
518:         processor = SemanticProcessor(config)
519:
520:         chunks = processor.chunk_text(text, preserve_structure=False)
521:
522:         required_fields = ["content", "section_type", "token_count", "position",
523:                            "has_table", "has_numerical", "pdq_context"]
524:
525:         for chunk in chunks:
526:             for field in required_fields:
527:                 assert field in chunk, f"Missing required field: {field}"
528:
529:                 assert isinstance(chunk["content"], str)
530:                 assert isinstance(chunk["token_count"], int)
531:                 assert isinstance(chunk["position"], int)
532:                 assert isinstance(chunk["has_table"], bool)
533:                 assert isinstance(chunk["has_numerical"], bool)
534:
535:     @given(text=spanish_policy_text())
536:     @settings(max_examples=10, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
537:     def test_embeddings_are_deterministic(self, text):
538:         """Property: Same text produces identical embeddings."""
539:         assume(len(text) > 50)
540:
541:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
542:         processor = SemanticProcessor(config)
543:
544:         emb1 = processor.embed_single(text)
545:         emb2 = processor.embed_single(text)
546:
547:         np.testing.assert_array_almost_equal(emb1, emb2, decimal=5,
548:                                              err_msg="Embeddings must be deterministic")
549:
550:
551: class TestBayesianEvidenceProperties:
552:     """Property-based tests for Bayesian evidence integration."""
553:
554:     @given(
555:         similarities=st.lists(st.floats(min_value=-1.0, max_value=1.0), min_size=1, max_size=20),
556:         prior_concentration=st.floats(min_value=0.1, max_value=2.0)
557:     )
558:     @settings(max_examples=50, deadline=1000)
559:     def test_evidence_integration_is_bounded(self, similarities, prior_concentration):
560:         """Property: Evidence integration produces scores in [0, 1]."""
```

```

561:     integrator = BayesianEvidenceIntegrator(prior_concentration)
562:
563:     metadata = [{"position": i, "has_table": False, "has_numerical": False,
564:                  "section_type": "diagnostic"}]
565:                 for i in range(len(similarities))]
566:
567:     result = integrator.integrate_evidence(np.array(similarities), metadata)
568:
569:     assert 0.0 <= result["posterior_mean"] <= 1.0, \
570:           f"Posterior mean out of bounds: {result['posterior_mean']}"
571:     assert 0.0 <= result["confidence"] <= 1.0, \
572:           f"Confidence out of bounds: {result['confidence']}"
573:     assert result["information_gain"] >= 0.0, \
574:           f"Information gain must be non-negative: {result['information_gain']}"
575:
576: @given(
577:     similarities=st.lists(st.floats(min_value=0.5, max_value=1.0), min_size=1, max_size=10)
578: )
579: @settings(max_examples=30, deadline=1000)
580: def test_high_similarity_increases_confidence(self, similarities):
581:     """Property: Higher similarities lead to higher confidence."""
582:     integrator = BayesianEvidenceIntegrator(0.5)
583:
584:     metadata = [{"position": i, "has_table": False, "has_numerical": False,
585:                  "section_type": "diagnostic"}]
586:                 for i in range(len(similarities))]
587:
588:     result = integrator.integrate_evidence(np.array(similarities), metadata)
589:
590:     assert result["posterior_mean"] > 0.4, \
591:           "High similarities should produce high posterior mean"
592:     assert result["confidence"] > 0.3, \
593:           "High similarities should produce reasonable confidence"
594:
595: @given(
596:     n_chunks=st.integers(min_value=1, max_value=20),
597:     prior=st.floats(min_value=0.1, max_value=2.0)
598: )
599: @settings(max_examples=30, deadline=1000)
600: def test_evidence_integration_is_symmetric(self, n_chunks, prior):
601:     """Property: Same similarities in different order produce same result."""
602:     integrator = BayesianEvidenceIntegrator(prior)
603:
604:     similarities = np.random.RandomState(42).uniform(0.3, 0.9, n_chunks)
605:     metadata = [{"position": i, "has_table": False, "has_numerical": False,
606:                  "section_type": "diagnostic"}]
607:                 for i in range(n_chunks)]
608:
609:     result1 = integrator.integrate_evidence(similarities, metadata)
610:
611:     shuffled_indices = np.random.RandomState(43).permutation(n_chunks)
612:     similarities_shuffled = similarities[shuffled_indices]
613:     metadata_shuffled = [metadata[i] for i in shuffled_indices]
614:
615:     result2 = integrator.integrate_evidence(similarities_shuffled, metadata_shuffled)
616:

```

```
617:         assert abs(result1["posterior_mean"] - result2["posterior_mean"]) < 0.15, \
618:             "Evidence integration should be approximately order-independent"
619:
620:
621: class TestSemanticChunkingInvariants:
622:     """Invariant tests for semantic chunking pipeline."""
623:
624:     @given(text=spanish_policy_text())
625:     @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
626:     def test_chunk_positions_are_monotonic(self, text):
627:         """Property: Chunk positions increase monotonically."""
628:         assume(len(text) > 100)
629:
630:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
631:         processor = SemanticProcessor(config)
632:
633:         chunks = processor.chunk_text(text, preserve_structure=False)
634:
635:         positions = [c["position"] for c in chunks]
636:         assert positions == sorted(positions), "Chunk positions must be monotonic"
637:
638:     @given(text=spanish_policy_text())
639:     @settings(max_examples=15, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
640:     def test_chunk_hashes_are_unique_for_different_content(self, text):
641:         """Property: Different chunk contents produce different hashes."""
642:         assume(len(text) > 200)
643:
644:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
645:         processor = SemanticProcessor(config)
646:
647:         chunks = processor.chunk_text(text, preserve_structure=False)
648:
649:         if len(chunks) >= 2:
650:             hash1 = hashlib.sha256(chunks[0]["content"].encode()).hexdigest()
651:             hash2 = hashlib.sha256(chunks[1]["content"].encode()).hexdigest()
652:
653:             if chunks[0]["content"] != chunks[1]["content"]:
654:                 assert hash1 != hash2, "Different content must have different hashes"
655:
656:     @given(
657:         text=spanish_policy_text(),
658:         seed=st.integers(min_value=0, max_value=1000)
659:     )
660:     @settings(max_examples=10, deadline=5000, suppress_health_check=[HealthCheck.too_slow])
661:     def test_chunking_is_reproducible_with_seed(self, text, seed):
662:         """Property: Same seed produces same chunking."""
663:         assume(len(text) > 100)
664:
665:         np.random.seed(seed)
666:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
667:         processor1 = SemanticProcessor(config)
668:         chunks1 = processor1.chunk_text(text, preserve_structure=False)
669:
670:         np.random.seed(seed)
671:         config = SemanticConfig(device="cpu", batch_size=8, fp16=False)
672:         processor2 = SemanticProcessor(config)
```

```
673:     chunks2 = processor2.chunk_text(text, preserve_structure=False)
674:
675:     assert len(chunks1) == len(chunks2)
676:     for c1, c2 in zip(chunks1, chunks2, strict=False):
677:         assert c1["content"] == c2["content"]
678:
679:
680: if __name__ == "__main__":
681:     pytest.main([__file__, "-v", "--tb=short"])
682:
683:
684:
685: =====
686: FILE: tests/processing/test_spc_ingestion_properties.py
687: =====
688:
689: """
690: Property-Based Tests for SPC Ingestion Pipeline Idempotency
691:
692: Verifies that the SPC ingestion pipeline is idempotent, meaning running
693: it multiple times on the same input produces identical results.
694: """
695:
696: import hashlib
697: import json
698:
699: import pytest
700: from hypothesis import HealthCheck, assume, given, settings
701: from hypothesis import strategies as st
702:
703: from src.farfan_pipeline.processing.spc_ingestion import StrategicChunkingSystem
704:
705:
706: @st.composite
707: def minimal_policy_document(draw):
708:     """Generate minimal valid policy documents for testing."""
709:     title = draw(st.text(min_size=10, max_size=100))
710:     content_paragraphs = draw(st.lists(
711:         st.text(min_size=50, max_size=300),
712:         min_size=3,
713:         max_size=10
714:     ))
715:     content = "\n\n".join(content_paragraphs)
716:
717:     return {
718:         "doc_id": draw(st.text(alphabet=st.characters(whitelist_categories=('L', 'N'))),
719:                         min_size=5, max_size=20)),
720:         "title": title,
721:         "content": content
722:     }
723:
724:
725: class TestSPCIngestionIdempotency:
726:     """Property-based tests for SPC ingestion idempotency."""
727:
728:     @given(doc=minimal_policy_document())
```

```
729:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
730:     def test_ingestion_is_idempotent(self, doc):
731:         """Property: Running ingestion twice produces identical results."""
732:         assume(len(doc["content"]) > 200)
733:
734:         system = StrategicChunkingSystem()
735:
736:         result1 = system.process_document(doc["content"], {
737:             "doc_id": doc["doc_id"],
738:             "title": doc["title"]
739:         })
740:
741:         result2 = system.process_document(doc["content"], {
742:             "doc_id": doc["doc_id"],
743:             "title": doc["title"]
744:         })
745:
746:         assert len(result1) == len(result2), "Chunk count must be identical"
747:
748:         for i, (c1, c2) in enumerate(zip(result1, result2, strict=False)):
749:             c1_text = system.get_chunk_text(c1)
750:             c2_text = system.get_chunk_text(c2)
751:             assert c1_text == c2_text, f"Chunk {i} content differs"
752:
753:     @given(doc=minimal_policy_document())
754:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
755:     def test_chunk_hashes_are_stable(self, doc):
756:         """Property: Chunk hashes remain stable across runs."""
757:         assume(len(doc["content"]) > 200)
758:
759:         system = StrategicChunkingSystem()
760:
761:         result1 = system.process_document(doc["content"], {
762:             "doc_id": doc["doc_id"],
763:             "title": doc["title"]
764:         })
765:
766:         result2 = system.process_document(doc["content"], {
767:             "doc_id": doc["doc_id"],
768:             "title": doc["title"]
769:         })
770:
771:         for c1, c2 in zip(result1, result2, strict=False):
772:             c1_text = system.get_chunk_text(c1)
773:             c2_text = system.get_chunk_text(c2)
774:
775:             hash1 = hashlib.sha256(c1_text.encode()).hexdigest()
776:             hash2 = hashlib.sha256(c2_text.encode()).hexdigest()
777:
778:             assert hash1 == hash2, "Chunk hashes must be stable"
779:
780:     @given(doc=minimal_policy_document())
781:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
782:     def test_metadata_is_consistent(self, doc):
783:         """Property: Chunk metadata is consistent across runs."""
784:         assume(len(doc["content"]) > 200)
```

```
785:         system = StrategicChunkingSystem()
786:
787:
788:     result1 = system.process_document(doc["content"], {
789:         "doc_id": doc["doc_id"],
790:         "title": doc["title"]
791:     })
792:
793:     result2 = system.process_document(doc["content"], {
794:         "doc_id": doc["doc_id"],
795:         "title": doc["title"]
796:     })
797:
798:     for c1, c2 in zip(result1, result2, strict=False):
799:         meta1 = system.get_chunk_metadata(c1)
800:         meta2 = system.get_chunk_metadata(c2)
801:
802:         assert meta1["document_id"] == meta2["document_id"]
803:         assert meta1["has_table"] == meta2["has_table"]
804:         assert meta1["has_list"] == meta2["has_list"]
805:         assert meta1["has_numbers"] == meta2["has_numbers"]
806:
807:     @given(doc=minimal_policy_document())
808:     @settings(max_examples=8, deadline=15000, suppress_health_check=[HealthCheck.too_slow])
809:     def test_pdq_context_inference_is_stable(self, doc):
810:         """Property: PDQ context inference is stable across runs."""
811:         assume(len(doc["content"]) > 200)
812:
813:         system = StrategicChunkingSystem()
814:
815:         result1 = system.process_document(doc["content"], {
816:             "doc_id": doc["doc_id"],
817:             "title": doc["title"]
818:         })
819:
820:         result2 = system.process_document(doc["content"], {
821:             "doc_id": doc["doc_id"],
822:             "title": doc["title"]
823:         })
824:
825:         for c1, c2 in zip(result1, result2, strict=False):
826:             ctx1 = system.get_chunk_pdq_context(c1)
827:             ctx2 = system.get_chunk_pdq_context(c2)
828:
829:             if ctx1 is not None and ctx2 is not None:
830:                 assert ctx1["policy"] == ctx2["policy"]
831:                 assert ctx1["dimension"] == ctx2["dimension"]
832:
833:
834: class TestSPCIgestionInvariants:
835:     """Invariant tests for SPC ingestion pipeline."""
836:
837:     @given(doc=minimal_policy_document())
838:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
839:     def test_chunks_cover_all_content(self, doc):
840:         """Property: Chunks collectively cover all important content."""
```

```
841:     assume(len(doc["content"]) > 200)
842:
843:     system = StrategicChunkingSystem()
844:
845:     chunks = system.process_document(doc["content"], {
846:         "doc_id": doc["doc_id"],
847:         "title": doc["title"]
848:     })
849:
850:     all_chunk_text = " ".join(system.get_chunk_text(c) for c in chunks)
851:
852:     important_words = set()
853:     for word in doc["content"].split():
854:         if len(word) > 5:
855:             important_words.add(word.lower())
856:
857:     coverage = sum(1 for word in important_words
858:                   if word in all_chunk_text.lower())
859:
860:     coverage_ratio = coverage / max(len(important_words), 1)
861:     assert coverage_ratio > 0.7, \
862:         f"Chunks must cover at least 70% of important words: {coverage_ratio:.2%}"
863:
864:     @given(doc=minimal_policy_document())
865:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
866:     def test_chunks_have_minimum_quality(self, doc):
867:         """Property: All chunks meet minimum quality thresholds."""
868:         assume(len(doc["content"]) > 200)
869:
870:         system = StrategicChunkingSystem()
871:
872:         chunks = system.process_document(doc["content"], {
873:             "doc_id": doc["doc_id"],
874:             "title": doc["title"]
875:         })
876:
877:         for chunk in chunks:
878:             text = system.get_chunk_text(chunk)
879:
880:             assert len(text) >= 20, f"Chunk too short: {len(text)} chars"
881:
882:             assert text.strip(), "Chunk must not be empty"
883:
884:             words = text.split()
885:             assert len(words) >= 5, f"Chunk has too few words: {len(words)}"
886:
887:     @given(doc=minimal_policy_document())
888:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
889:     def test_chunk_boundaries_dont_duplicate_content(self, doc):
890:         """Property: Chunks don't have excessive duplication."""
891:         assume(len(doc["content"]) > 200)
892:
893:         system = StrategicChunkingSystem()
894:
895:         chunks = system.process_document(doc["content"], {
896:             "doc_id": doc["doc_id"],
```

```
897:         "title": doc["title"]
898:     })
899:
900:     if len(chunks) >= 2:
901:         for i in range(len(chunks) - 1):
902:             text1 = system.get_chunk_text(chunks[i])
903:             text2 = system.get_chunk_text(chunks[i + 1])
904:
905:             words1 = set(text1.lower().split())
906:             words2 = set(text2.lower().split())
907:
908:             if words1 and words2:
909:                 overlap = len(words1 & words2) / max(len(words1), len(words2))
910:                 assert overlap < 0.8, \
911:                     f"Excessive overlap between chunks {i} and {i+1}: {overlap:.2%}"
912:
913:
914: class TestSPCIgestionCorrectness:
915:     """Correctness properties for SPC ingestion."""
916:
917:     @given(doc=minimal_policy_document())
918:     @settings(max_examples=10, deadline=10000, suppress_health_check=[HealthCheck.too_slow])
919:     def test_document_id_propagates_to_chunks(self, doc):
920:         """Property: Document ID is preserved in all chunks."""
921:         assume(len(doc["content"]) > 200)
922:
923:         system = StrategicChunkingSystem()
924:
925:         chunks = system.process_document(doc["content"], {
926:             "doc_id": doc["doc_id"],
927:             "title": doc["title"]
928:         })
929:
930:         for chunk in chunks:
931:             meta = system.get_chunk_metadata(chunk)
932:             assert meta["document_id"] == doc["doc_id"], \
933:                 "Document ID must propagate to all chunks"
934:
935:     @given(
936:         doc=minimal_policy_document(),
937:         run_count=st.integers(min_value=2, max_value=4)
938:     )
939:     @settings(max_examples=5, deadline=20000, suppress_health_check=[HealthCheck.too_slow])
940:     def test_multiple_runs_produce_identical_output(self, doc, run_count):
941:         """Property: Multiple runs produce bit-for-bit identical output."""
942:         assume(len(doc["content"]) > 200)
943:
944:         system = StrategicChunkingSystem()
945:
946:         results = []
947:         for _ in range(run_count):
948:             chunks = system.process_document(doc["content"], {
949:                 "doc_id": doc["doc_id"],
950:                 "title": doc["title"]
951:             })
952:
```

```
953:         serialized = json.dumps([
954:             {
955:                 "text": system.get_chunk_text(c),
956:                 "metadata": system.get_chunk_metadata(c)
957:             }
958:             for c in chunks
959:         ], sort_keys=True)
960:
961:     results.append(hashlib.sha256(serialized.encode()).hexdigest())
962:
963:     assert len(set(results)) == 1, \
964:         f"Multiple runs produced different outputs: {results}"
965:
966:
967: if __name__ == "__main__":
968:     pytest.main([__file__, "-v", "--tb=short"])
969:
970:
971:
972: =====
973: FILE: tests/synchronization/__init__.py
974: =====
975:
976:
977:
978:
979: =====
980: FILE: tests/synchronization/test_irrigation_synchronizer.py
981: =====
982:
983: """Unit tests for ChunkMatrix validation and chunk routing."""
984:
985: import sys
986: from datetime import datetime
987: from pathlib import Path
988:
989: import pytest
990:
991: project_root = Path(__file__).parent.parent.parent
992: sys.path.insert(0, str(project_root / "src"))
993:
994: from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
995:     ChunkRoutingResult,
996:     IrrigationSynchronizer,
997: )
998: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
999: from farfan_pipeline.synchronization.irrigation_synchronizer import ChunkMatrix
1000:
1001:
1002: def create_chunk(
1003:     chunk_id: int, policy_area_id: str, dimension_id: str, text: str = "test content"
1004: ) -> ChunkData:
1005:     """Factory for creating test chunks."""
1006:     return ChunkData(
1007:         id=chunk_id,
1008:         text=text,
```

```
1009:     chunk_type="diagnostic",
1010:     sentences=[],
1011:     tables=[],
1012:     start_pos=0,
1013:     end_pos=len(text),
1014:     confidence=0.95,
1015:     policy_area_id=policy_area_id,
1016:     dimension_id=dimension_id,
1017: )
1018:
1019:
1020: def create_complete_document() -> PreprocessedDocument:
1021:     """Create a valid document with all 60 required chunks."""
1022:     chunks = []
1023:     chunk_id = 0
1024:     for pa_num in range(1, 11):
1025:         for dim_num in range(1, 7):
1026:             pa_id = f"PA{pa_num:02d}"
1027:             dim_id = f"DIM{dim_num:02d}"
1028:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
1029:             chunk_id += 1
1030:
1031:     return PreprocessedDocument(
1032:         document_id="test-doc",
1033:         raw_text="test",
1034:         sentences=[],
1035:         tables=[],
1036:         metadata={},
1037:         chunks=chunks,
1038:         ingested_at=datetime.now(),
1039:     )
1040:
1041:
1042: def test_chunk_matrix_accepts_valid_60_chunks() -> None:
1043:     """ChunkMatrix should accept exactly 60 valid chunks."""
1044:     doc = create_complete_document()
1045:     matrix = ChunkMatrix(doc)
1046:
1047:     chunk = matrix.get_chunk("PA01", "DIM01")
1048:     assert chunk.policy_area_id == "PA01"
1049:     assert chunk.dimension_id == "DIM01"
1050:
1051:
1052: def test_chunk_matrix_rejects_59_chunks() -> None:
1053:     """ChunkMatrix should reject documents with 59 chunks by detecting missing combination."""
1054:     doc = create_complete_document()
1055:     doc.chunks = doc.chunks[:59]
1056:
1057:     with pytest.raises(ValueError) as exc_info:
1058:         ChunkMatrix(doc)
1059:
1060:     error_msg = str(exc_info.value)
1061:     assert "Missing chunk combinations" in error_msg
1062:
1063:
1064: def test_chunk_matrix_rejects_61_chunks() -> None:
```

```
1065:     """ChunkMatrix should reject documents with 61 chunks by detecting duplicate."""
1066:     doc = create_complete_document()
1067:     duplicate_chunk = doc.chunks[0]
1068:     doc.chunks.append(
1069:         create_chunk(
1070:             60,
1071:             duplicate_chunk.policy_area_id,
1072:             duplicate_chunk.dimension_id,
1073:             "duplicate",
1074:         )
1075:     )
1076:
1077:     with pytest.raises(ValueError) as exc_info:
1078:         ChunkMatrix(doc)
1079:
1080:     error_msg = str(exc_info.value)
1081:     assert "Duplicate (PA, DIM) combination detected" in error_msg
1082:
1083:
1084: def test_chunk_matrix_rejects_duplicate_keys() -> None:
1085:     """ChunkMatrix should reject documents with duplicate PA-DIM combinations."""
1086:     chunks = []
1087:     chunks.append(create_chunk(0, "PA01", "DIM01"))
1088:     chunks.append(create_chunk(1, "PA01", "DIM01"))
1089:
1090:     chunk_id = 2
1091:     for pa_num in range(1, 11):
1092:         for dim_num in range(1, 7):
1093:             if pa_num == 1 and dim_num == 1:
1094:                 continue
1095:             pa_id = f"PA{pa_num:02d}"
1096:             dim_id = f"DIM{dim_num:02d}"
1097:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
1098:             chunk_id += 1
1099:
1100:     doc = PreprocessedDocument(
1101:         document_id="test-doc",
1102:         raw_text="test",
1103:         sentences=[],
1104:         tables=[],
1105:         metadata={},
1106:         chunks=chunks,
1107:         ingested_at=datetime.now(),
1108:     )
1109:
1110:     with pytest.raises(ValueError, match=r"Duplicate .*PA01-DIM01"):
1111:         ChunkMatrix(doc)
1112:
1113:
1114: def test_chunk_matrix_detects_missing_combinations() -> None:
1115:     """ChunkMatrix should detect and report missing PA-DIM combinations with 59 chunks."""
1116:     chunks = []
1117:     chunk_id = 0
1118:     for pa_num in range(1, 11):
1119:         for dim_num in range(1, 7):
1120:             if pa_num == 5 and dim_num == 3:
```

```
1121:         continue
1122:         pa_id = f"PA{pa_num:02d}"
1123:         dim_id = f"DIM{dim_num:02d}"
1124:         chunks.append(create_chunk(chunk_id, pa_id, dim_id))
1125:         chunk_id += 1
1126:
1127:     doc = PreprocessedDocument(
1128:         document_id="test-doc",
1129:         raw_text="test",
1130:         sentences=[],
1131:         tables=[],
1132:         metadata={},
1133:         chunks=chunks,
1134:         ingested_at=datetime.now(),
1135:     )
1136:
1137:     with pytest.raises(ValueError) as exc_info:
1138:         ChunkMatrix(doc)
1139:
1140:     error_msg = str(exc_info.value)
1141:     assert "Missing chunk combinations" in error_msg
1142:     assert "PA05", 'DIM03' in error_msg
1143:
1144:
1145: def test_chunk_matrix_detects_missing_combinations_with_exact_error() -> None:
1146:     """ChunkMatrix should report missing PA-DIM combination when 60 chunks but duplicate."""
1147:     doc = create_complete_document()
1148:     original_chunk = doc.chunks[0]
1149:     doc.chunks[0] = create_chunk(
1150:         0, original_chunk.policy_area_id, original_chunk.dimension_id
1151:     )
1152:     doc.chunks[10] = create_chunk(
1153:         10, original_chunk.policy_area_id, original_chunk.dimension_id
1154:     )
1155:
1156:     with pytest.raises(ValueError) as exc_info:
1157:         ChunkMatrix(doc)
1158:
1159:     error_msg = str(exc_info.value)
1160:     assert "Duplicate" in error_msg
1161:     assert f"{original_chunk.policy_area_id}-{original_chunk.dimension_id}" in error_msg
1162:
1163:
1164: def test_chunk_matrix_reports_specific_missing_key() -> None:
1165:     """ChunkMatrix should report specific missing key PA05-DIM03 in error message."""
1166:     chunks = []
1167:     chunk_id = 0
1168:     for pa_num in range(1, 11):
1169:         for dim_num in range(1, 7):
1170:             if pa_num == 5 and dim_num == 3:
1171:                 chunk_id += 1
1172:                 continue
1173:             pa_id = f"PA{pa_num:02d}"
1174:             dim_id = f"DIM{dim_num:02d}"
1175:             chunks.append(create_chunk(chunk_id, pa_id, dim_id))
1176:             chunk_id += 1
```

```
1177:  
1178:     chunks.append(create_chunk(59, "PA05", "DIM03"))  
1179:  
1180:     doc = PreprocessedDocument(  
1181:         document_id="test-doc",  
1182:         raw_text="test",  
1183:         sentences=[],  
1184:         tables=[],  
1185:         metadata={},  
1186:         chunks=chunks,  
1187:         ingested_at=datetime.now(),  
1188:     )  
1189:  
1190:     matrix = ChunkMatrix(doc)  
1191:     chunk = matrix.get_chunk("PA05", "DIM03")  
1192:     assert chunk.policy_area_id == "PA05"  
1193:     assert chunk.dimension_id == "DIM03"  
1194:  
1195:  
1196: def test_chunk_matrix_rejects_null_policy_area_id() -> None:  
1197:     """ChunkData should reject chunks with null policy_area_id at creation time."""  
1198:     with pytest.raises(ValueError, match="chunk_id is required"):  
1199:         ChunkData(  
1200:             id=0,  
1201:             text="test",  
1202:             chunk_type="diagnostic",  
1203:             sentences=[],  
1204:             tables=[],  
1205:             start_pos=0,  
1206:             end_pos=4,  
1207:             confidence=0.95,  
1208:             policy_area_id=None,  
1209:             dimension_id="DIM01",  
1210:         )  
1211:  
1212:  
1213: def test_chunk_matrix_rejects_null_dimension_id() -> None:  
1214:     """ChunkData should reject chunks with null dimension_id at creation time."""  
1215:     with pytest.raises(ValueError, match="chunk_id is required"):  
1216:         ChunkData(  
1217:             id=0,  
1218:             text="test",  
1219:             chunk_type="diagnostic",  
1220:             sentences=[],  
1221:             tables=[],  
1222:             start_pos=0,  
1223:             end_pos=4,  
1224:             confidence=0.95,  
1225:             policy_area_id="PA01",  
1226:             dimension_id=None,  
1227:         )  
1228:  
1229:  
1230: def test_chunk_matrix_rejects_invalid_chunk_id_format_pa() -> None:  
1231:     """ChunkData should reject invalid policy area format at creation time."""  
1232:     with pytest.raises(ValueError, match=r"Invalid chunk_id"):
```

```
1233:         create_chunk(0, "PA11", "DIM01")
1234:
1235:
1236: def test_chunk_matrix_rejects_invalid_chunk_id_format_dim() -> None:
1237:     """ChunkData should reject invalid dimension format at creation time."""
1238:     with pytest.raises(ValueError, match=r"Invalid chunk_id"):
1239:         create_chunk(0, "PA01", "DIM07")
1240:
1241:
1242: def test_chunk_matrix_rejects_malformed_chunk_id() -> None:
1243:     """ChunkData should reject malformed chunk IDs at creation time."""
1244:     with pytest.raises(ValueError, match=r"Invalid chunk_id"):
1245:         create_chunk(0, "P01", "DIM01")
1246:
1247:
1248: def test_chunk_matrix_get_chunk_success() -> None:
1249:     """get_chunk should return correct chunk for valid keys."""
1250:     doc = create_complete_document()
1251:     matrix = ChunkMatrix(doc)
1252:
1253:     chunk = matrix.get_chunk("PA05", "DIM03")
1254:     assert chunk.policy_area_id == "PA05"
1255:     assert chunk.dimension_id == "DIM03"
1256:
1257:
1258: def test_chunk_matrix_get_chunk_all_combinations() -> None:
1259:     """get_chunk should work for all 60 valid combinations."""
1260:     doc = create_complete_document()
1261:     matrix = ChunkMatrix(doc)
1262:
1263:     for pa_num in range(1, 11):
1264:         for dim_num in range(1, 7):
1265:             pa_id = f"PA{pa_num:02d}"
1266:             dim_id = f"DIM{dim_num:02d}"
1267:             chunk = matrix.get_chunk(pa_id, dim_id)
1268:             assert chunk.policy_area_id == pa_id
1269:             assert chunk.dimension_id == dim_id
1270:
1271:
1272: def test_chunk_matrix_get_chunk_missing_key() -> None:
1273:     """get_chunk should raise KeyError for missing combinations."""
1274:     doc = create_complete_document()
1275:     doc.chunks = [
1276:         chunk
1277:         for chunk in doc.chunks
1278:             if not (chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03")
1279:     ]
1280:     doc.chunks.append(create_chunk(59, "PA05", "DIM03"))
1281:
1282:     matrix = ChunkMatrix(doc)
1283:
1284:     with pytest.raises(KeyError, match="Chunk not found for key: PA11-DIM01"):
1285:         matrix.get_chunk("PA11", "DIM01")
1286:
1287:
1288: def test_chunk_matrix_get_chunk_o1_lookup() -> None:
```

```
1289:     """get_chunk should provide O(1) lookup performance."""
1290:     doc = create_complete_document()
1291:     matrix = ChunkMatrix(doc)
1292:
1293:     import time
1294:
1295:     iterations = 1000
1296:     start = time.perf_counter()
1297:     for _ in range(iterations):
1298:         matrix.get_chunk("PA05", "DIM03")
1299:     elapsed = time.perf_counter() - start
1300:
1301:     assert elapsed < 0.1, f"O(1) lookup too slow: {elapsed}s for {iterations} lookups"
1302:
1303:
1304: def test_chunk_matrix_preserves_chunk_content() -> None:
1305:     """ChunkMatrix should preserve original chunk content and metadata."""
1306:     doc = create_complete_document()
1307:     doc.chunks[0] = ChunkData(
1308:         id=42,
1309:         text="specific test content",
1310:         chunk_type="activity",
1311:         sentences=[1, 2, 3],
1312:         tables=[],
1313:         start_pos=10,
1314:         end_pos=31,
1315:         confidence=0.87,
1316:         policy_area_id="PA01",
1317:         dimension_id="DIM01",
1318:     )
1319:
1320:     matrix = ChunkMatrix(doc)
1321:     chunk = matrix.get_chunk("PA01", "DIM01")
1322:
1323:     assert chunk.id == 42
1324:     assert chunk.text == "specific test content"
1325:     assert chunk.chunk_type == "activity"
1326:     assert chunk.sentences == [1, 2, 3]
1327:     assert chunk.confidence == 0.87
1328:
1329:
1330: def test_validate_chunk_routing_success() -> None:
1331:     """validate_chunk_routing should successfully route valid question to chunk."""
1332:     doc = create_complete_document()
1333:     questionnaire = {"blocks": {}}
1334:     synchronizer = IrrigationSynchronizer(
1335:         questionnaire=questionnaire, preprocessed_document=doc
1336:     )
1337:
1338:     question = {
1339:         "question_id": "D1-Q01",
1340:         "policy_area_id": "PA05",
1341:         "dimension_id": "DIM03",
1342:         "question_text": "Test question",
1343:     }
1344:
```

```
1345:     result = synchronizer.validate_chunk_routing(question)
1346:
1347:     assert isinstance(result, ChunkRoutingResult)
1348:     assert result.chunk_id == "PA05-DIM03"
1349:     assert result.policy_area_id == "PA05"
1350:     assert result.dimension_id == "DIM03"
1351:     assert result.text_content == "test content"
1352:     assert result.target_chunk.policy_area_id == "PA05"
1353:     assert result.target_chunk.dimension_id == "DIM03"
1354:
1355:
1356: def test_validate_chunk_routing_missing_question_id() -> None:
1357:     """validate_chunk_routing should raise ValueError if question_id missing."""
1358:     doc = create_complete_document()
1359:     questionnaire = {"blocks": {}}
1360:     synchronizer = IrrigationSynchronizer(
1361:         questionnaire=questionnaire, preprocessed_document=doc
1362:     )
1363:
1364:     question = {"policy_area_id": "PA05", "dimension_id": "DIM03"}
1365:
1366:     with pytest.raises(ValueError, match="missing required 'question_id' field"):
1367:         synchronizer.validate_chunk_routing(question)
1368:
1369:
1370: def test_validate_chunk_routing_missing_policy_area_id() -> None:
1371:     """validate_chunk_routing should raise ValueError if policy_area_id missing."""
1372:     doc = create_complete_document()
1373:     questionnaire = {"blocks": {}}
1374:     synchronizer = IrrigationSynchronizer(
1375:         questionnaire=questionnaire, preprocessed_document=doc
1376:     )
1377:
1378:     question = {"question_id": "D1-Q01", "dimension_id": "DIM03"}
1379:
1380:     with pytest.raises(ValueError, match="missing required 'policy_area_id' field"):
1381:         synchronizer.validate_chunk_routing(question)
1382:
1383:
1384: def test_validate_chunk_routing_missing_dimension_id() -> None:
1385:     """validate_chunk_routing should raise ValueError if dimension_id missing."""
1386:     doc = create_complete_document()
1387:     questionnaire = {"blocks": {}}
1388:     synchronizer = IrrigationSynchronizer(
1389:         questionnaire=questionnaire, preprocessed_document=doc
1390:     )
1391:
1392:     question = {"question_id": "D1-Q01", "policy_area_id": "PA05"}
1393:
1394:     with pytest.raises(ValueError, match="missing required 'dimension_id' field"):
1395:         synchronizer.validate_chunk_routing(question)
1396:
1397:
1398: def test_validate_chunk_routing_chunk_not_found() -> None:
1399:     """validate_chunk_routing should raise ValueError if chunk not in matrix."""
1400:     doc = create_complete_document()
```

```
1401:     doc.chunks = [
1402:         chunk
1403:         for chunk in doc.chunks
1404:         if not (chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03")
1405:     ]
1406:
1407:     questionnaire = {"blocks": {}}
1408:
1409:     with pytest.raises(
1410:         ValueError,
1411:         match="Chunk matrix validation failed during synchronizer initialization",
1412:     ):
1413:         IrrigationSynchronizer(questionnaire=questionnaire, preprocessed_document=doc)
1414:
1415:
1416: def test_validate_chunk_routing_empty_text() -> None:
1417:     """validate_chunk_routing should raise ValueError if chunk text is empty."""
1418:     doc = create_complete_document()
1419:     for chunk in doc.chunks:
1420:         if chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03":
1421:             idx = doc.chunks.index(chunk)
1422:             doc.chunks[idx] = ChunkData(
1423:                 id=chunk.id,
1424:                 text="",
1425:                 chunk_type=chunk.chunk_type,
1426:                 sentences=chunk.sentences,
1427:                 tables=chunk.tables,
1428:                 start_pos=chunk.start_pos,
1429:                 end_pos=chunk.end_pos,
1430:                 confidence=chunk.confidence,
1431:                 policy_area_id=chunk.policy_area_id,
1432:                 dimension_id=chunk.dimension_id,
1433:             )
1434:             break
1435:
1436:     questionnaire = {"blocks": {}}
1437:     synchronizer = IrrigationSynchronizer(
1438:         questionnaire=questionnaire, preprocessed_document=doc
1439:     )
1440:
1441:     question = {
1442:         "question_id": "D1-Q01",
1443:         "policy_area_id": "PA05",
1444:         "dimension_id": "DIM03",
1445:     }
1446:
1447:     with pytest.raises(ValueError, match="empty or whitespace-only text"):
1448:         synchronizer.validate_chunk_routing(question)
1449:
1450:
1451: def test_validate_chunk_routing_whitespace_only_text() -> None:
1452:     """validate_chunk_routing should raise ValueError if chunk text is whitespace."""
1453:     doc = create_complete_document()
1454:     for chunk in doc.chunks:
1455:         if chunk.policy_area_id == "PA05" and chunk.dimension_id == "DIM03":
1456:             idx = doc.chunks.index(chunk)
```

```
1457:         doc.chunks[idx] = ChunkData(
1458:             id=chunk.id,
1459:             text="\n\t",
1460:             chunk_type=chunk.chunk_type,
1461:             sentences=chunk.sentences,
1462:             tables=chunk.tables,
1463:             start_pos=chunk.start_pos,
1464:             end_pos=chunk.end_pos,
1465:             confidence=chunk.confidence,
1466:             policy_area_id=chunk.policy_area_id,
1467:             dimension_id=chunk.dimension_id,
1468:         )
1469:     break
1470:
1471: questionnaire = {"blocks": {}}
1472: synchronizer = IrrigationSynchronizer(
1473:     questionnaire=questionnaire, preprocessed_document=doc
1474: )
1475:
1476: question = {
1477:     "question_id": "D1-Q01",
1478:     "policy_area_id": "PA05",
1479:     "dimension_id": "DIM03",
1480: }
1481:
1482: with pytest.raises(ValueError, match="empty or whitespace-only text"):
1483:     synchronizer.validate_chunk_routing(question)
1484:
1485:
1486: def test_validate_chunk_routing_expected_elements_default() -> None:
1487:     """validate_chunk_routing should handle missing expected_elements gracefully."""
1488:     doc = create_complete_document()
1489:     questionnaire = {"blocks": {}}
1490:     synchronizer = IrrigationSynchronizer(
1491:         questionnaire=questionnaire, preprocessed_document=doc
1492:     )
1493:
1494:     question = {
1495:         "question_id": "D1-Q01",
1496:         "policy_area_id": "PA05",
1497:         "dimension_id": "DIM03",
1498:     }
1499:
1500:     result = synchronizer.validate_chunk_routing(question)
1501:
1502:     assert result.expected_elements == []
1503:
1504:
1505: def test_validate_chunk_routing_immutability() -> None:
1506:     """ChunkRoutingResult should be immutable."""
1507:     doc = create_complete_document()
1508:     questionnaire = {"blocks": {}}
1509:     synchronizer = IrrigationSynchronizer(
1510:         questionnaire=questionnaire, preprocessed_document=doc
1511:     )
1512:
```

```
1513:     question = {
1514:         "question_id": "D1-Q01",
1515:         "policy_area_id": "PA05",
1516:         "dimension_id": "DIM03",
1517:     }
1518:
1519:     result = synchronizer.validate_chunk_routing(question)
1520:
1521:     with pytest.raises(Exception):
1522:         result.chunk_id = "different_id"
1523:
1524:
1525: def test_filter_patterns_with_matching_policy_area() -> None:
1526:     """_filter_patterns should include only patterns with matching policy_area_id."""
1527:     doc = create_complete_document()
1528:     questionnaire = {"blocks": {}}
1529:     synchronizer = IrrigationSynchronizer(
1530:         questionnaire=questionnaire, preprocessed_document=doc
1531:     )
1532:
1533:     patterns = [
1534:         {"id": "PAT-001", "policy_area_id": "PA01", "pattern": "test1"},
1535:         {"id": "PAT-002", "policy_area_id": "PA02", "pattern": "test2"},
1536:         {"id": "PAT-003", "policy_area_id": "PA01", "pattern": "test3"},
1537:         {"id": "PAT-004", "pattern": "test4"},
1538:     ]
1539:
1540:     result = synchronizer._filter_patterns(patterns, "PA01")
1541:
1542:     assert isinstance(result, tuple)
1543:     assert len(result) == 2
1544:     assert result[0]["id"] == "PAT-001"
1545:     assert result[1]["id"] == "PAT-003"
1546:
1547:
1548: def test_filter_patterns_excludes_without_policy_area_id() -> None:
1549:     """_filter_patterns should exclude patterns without policy_area_id attribute."""
1550:     doc = create_complete_document()
1551:     questionnaire = {"blocks": {}}
1552:     synchronizer = IrrigationSynchronizer(
1553:         questionnaire=questionnaire, preprocessed_document=doc
1554:     )
1555:
1556:     patterns = [
1557:         {"id": "PAT-001", "pattern": "test1"},
1558:         {"id": "PAT-002", "pattern": "test2"},
1559:     ]
1560:
1561:     result = synchronizer._filter_patterns(patterns, "PA01")
1562:
1563:     assert isinstance(result, tuple)
1564:     assert len(result) == 0
1565:
1566:
1567: def test_filter_patterns_returns_empty_tuple_when_no_matches() -> None:
1568:     """_filter_patterns should return empty tuple when no patterns match."""
```

```
1569:     doc = create_complete_document()
1570:     questionnaire = {"blocks": {}}
1571:     synchronizer = IrrigationSynchronizer(
1572:         questionnaire=questionnaire, preprocessed_document=doc
1573:     )
1574:
1575:     patterns = [
1576:         {"id": "PAT-001", "policy_area_id": "PA02", "pattern": "test1"},
1577:         {"id": "PAT-002", "policy_area_id": "PA03", "pattern": "test2"},
1578:     ]
1579:
1580:     result = synchronizer._filter_patterns(patterns, "PA01")
1581:
1582:     assert isinstance(result, tuple)
1583:     assert len(result) == 0
1584:
1585:
1586: def test_filter_patterns_immutability() -> None:
1587:     """_filter_patterns should return an immutable tuple."""
1588:     doc = create_complete_document()
1589:     questionnaire = {"blocks": {}}
1590:     synchronizer = IrrigationSynchronizer(
1591:         questionnaire=questionnaire, preprocessed_document=doc
1592:     )
1593:
1594:     patterns = [
1595:         {"id": "PAT-001", "policy_area_id": "PA01", "pattern": "test1"},
1596:     ]
1597:
1598:     result = synchronizer._filter_patterns(patterns, "PA01")
1599:
1600:     assert isinstance(result, tuple)
1601:     with pytest.raises(AttributeError):
1602:         result.append({"id": "PAT-002", "pattern": "test2"})
1603:
1604:
1605: def test_filter_patterns_strict_equality() -> None:
1606:     """_filter_patterns should use strict equality for policy_area_id matching."""
1607:     doc = create_complete_document()
1608:     questionnaire = {"blocks": {}}
1609:     synchronizer = IrrigationSynchronizer(
1610:         questionnaire=questionnaire, preprocessed_document=doc
1611:     )
1612:
1613:     patterns = [
1614:         {"id": "PAT-001", "policy_area_id": "PA01", "pattern": "test1"},
1615:         {"id": "PAT-002", "policy_area_id": "PA010", "pattern": "test2"},
1616:         {"id": "PAT-003", "policy_area_id": "PA1", "pattern": "test3"},
1617:     ]
1618:
1619:     result = synchronizer._filter_patterns(patterns, "PA01")
1620:
1621:     assert isinstance(result, tuple)
1622:     assert len(result) == 1
1623:     assert result[0]["id"] == "PAT-001"
1624:
```

```
1625:  
1626:  
1627: =====  
1628: FILE: tests/test_aggregation_sota.py  
1629: =====  
1630:  
1631: """  
1632: Tests for SOTA aggregation features.  
1633:  
1634: This module tests the three SOTA injection points:  
1635: 1. Choquet integral aggregation (non-linear interactions)  
1636: 2. Bayesian uncertainty quantification (bootstrap + CI)  
1637: 3. DAG provenance tracking (lineage + SHAP attribution)  
1638: """  
1639:  
1640: import pytest  
1641: import numpy as np  
1642:  
1643: from farfan_pipeline.processing.aggregation_provenance import (  
1644:     AggregationDAG,  
1645:     ProvenanceNode,  
1646: )  
1647: from farfan_pipeline.processing.uncertainty_quantification import (  
1648:     BootstrapAggregator,  
1649:     BayesianPropagation,  
1650:     SensitivityAnalysis,  
1651:     aggregate_with_uncertainty,  
1652: )  
1653: from farfan_pipeline.processing.choquet_adapter import (  
1654:     ChoquetProcessingAdapter,  
1655:     create_default_choquet_adapter,  
1656: )  
1657:  
1658:  
1659: class TestChoquetAggregation:  
1660:     """Test Choquet integral aggregation."""  
1661:  
1662:     def test_choquetCreates_non_linear_interactions(self):  
1663:         """Choquet with interactions should differ from naive weighted average."""  
1664:         scores = [7.0, 8.0, 6.5]  
1665:         weights = [0.33, 0.33, 0.34]  
1666:  
1667:         # Naive weighted average  
1668:         naive_score = sum(s * w for s, w in zip(scores, weights))  
1669:  
1670:         # Choquet with interactions  
1671:         interaction_weights = {("0", "1"): 0.1} # Synergy between first two  
1672:         choquet = ChoquetProcessingAdapter(  
1673:             linear_weights={str(i): w for i, w in enumerate(weights)},  
1674:             interaction_weights=interaction_weights,  
1675:         )  
1676:         choquet_score = choquet.aggregate(scores, weights, interaction_pairs=[(0, 1)])  
1677:  
1678:         # Choquet should differ due to min(7.0, 8.0) * 0.1 = 0.7 interaction term  
1679:         assert abs(choquet_score - naive_score) > 0.01  
1680:         assert choquet_score > naive_score # Positive interaction (synergy)
```

```
1681:
1682:     def test_choquet_with_redundancy_penalty(self):
1683:         """Negative interaction weights should penalize redundancy."""
1684:         scores = [8.0, 8.0] # Redundant high scores
1685:         weights = [0.5, 0.5]
1686:
1687:         interaction_weights = {("0", "1"): -0.1} # Redundancy penalty
1688:         choquet = ChoquetProcessingAdapter(
1689:             linear_weights={"0": 0.5, "1": 0.5},
1690:             interaction_weights=interaction_weights,
1691:         )
1692:         choquet_score = choquet.aggregate(scores, weights, interaction_pairs=[(0, 1)])
1693:
1694:         naive_score = 8.0 # Weighted average
1695:         assert choquet_score < naive_score # Penalty applied
1696:
1697:     def test_choquet_default_adapter(self):
1698:         """Default adapter should work for n inputs."""
1699:         adapter = create_default_choquet_adapter(n_inputs=5)
1700:         scores = [7.0, 8.0, 6.5, 7.5, 8.5]
1701:         result = adapter.aggregate(scores, weights=None)
1702:
1703:         assert 0.0 <= result <= 10.0 # Reasonable range
1704:         assert isinstance(result, float)
1705:
1706:
1707: class TestUncertaintyQuantification:
1708:     """Test Bayesian uncertainty quantification."""
1709:
1710:     def test_bootstrap_computes_confidence_intervals(self):
1711:         """Bootstrap should provide 95% CI around mean."""
1712:         scores = [7.0, 7.5, 6.8, 7.2, 6.9]
1713:         weights = [0.2, 0.2, 0.2, 0.2, 0.2]
1714:
1715:         bootstrapper = BootstrapAggregator(n_samples=1000, random_seed=42)
1716:         uncertainty = bootstrapper.bootstrap_weighted_average(scores, weights)
1717:
1718:         # Mean should be close to weighted average
1719:         expected_mean = sum(s * w for s, w in zip(scores, weights))
1720:         assert abs(uncertainty.mean - expected_mean) < 0.1
1721:
1722:         # CI should contain mean
1723:         lower, upper = uncertainty.confidence_interval_95
1724:         assert lower < uncertainty.mean < upper
1725:
1726:         # Std should be positive
1727:         assert uncertainty.std > 0.0
1728:
1729:     def test_bootstrap_is_deterministic(self):
1730:         """Same seed should produce identical results."""
1731:         scores = [7.0, 8.0, 6.5]
1732:
1733:         bootstrap1 = BootstrapAggregator(n_samples=500, random_seed=42)
1734:         result1 = bootstrap1.bootstrap_weighted_average(scores)
1735:
1736:         bootstrap2 = BootstrapAggregator(n_samples=500, random_seed=42)
```

```
1737:         result2 = bootstrap2.bootstrap_weighted_average(scores)
1738:
1739:     assert result1.mean == result2.mean
1740:     assert result1.std == result2.std
1741:     assert result1.confidence_interval_95 == result2.confidence_interval_95
1742:
1743: def test_uncertainty_decomposition(self):
1744:     """Epistemic and aleatoric uncertainty should be non-negative."""
1745:     scores = [7.0, 8.0, 6.5, 7.5, 8.2]
1746:
1747:     _, uncertainty = aggregate_with_uncertainty(scores, n_bootstrap=1000, random_seed=42)
1748:
1749:     assert uncertainty.epistemic_uncertainty >= 0.0
1750:     assert uncertainty.aleatoric_uncertainty >= 0.0
1751:     assert uncertainty.dominant_uncertainty_type() in ["epistemic", "aleatoric", "balanced"]
1752:
1753: def test_bayesian_propagation_analytical(self):
1754:     """Analytical propagation should match expected variance formula."""
1755:     scores = [7.0, 8.0, 6.0]
1756:     uncertainties = [0.5, 0.3, 0.4]
1757:     weights = [0.33, 0.33, 0.34]
1758:
1759:     mean, std = BayesianPropagation.propagate_weighted_average(
1760:         scores, uncertainties, weights
1761:     )
1762:
1763:     # Mean should match weighted average
1764:     expected_mean = sum(s * w for s, w in zip(scores, weights))
1765:     assert abs(mean - expected_mean) < 1e-9
1766:
1767:     # Variance: sum(w_i^2 * var_i)
1768:     expected_var = sum(w**2 * u**2 for w, u in zip(weights, uncertainties))
1769:     expected_std = np.sqrt(expected_var)
1770:     assert abs(std - expected_std) < 1e-9
1771:
1772: def test_coefficient_of_variation_threshold(self):
1773:     """High CV should trigger high uncertainty flag."""
1774:     # Low uncertainty case
1775:     scores_low_var = [7.0, 7.1, 7.05, 6.95, 7.0]
1776:     _, unc_low = aggregate_with_uncertainty(scores_low_var, n_bootstrap=1000, random_seed=42)
1777:     assert not unc_low.is_high_uncertainty(threshold=0.1)
1778:
1779:     # High uncertainty case
1780:     scores_high_var = [3.0, 9.0, 2.0, 8.5, 4.0]
1781:     _, unc_high = aggregate_with_uncertainty(scores_high_var, n_bootstrap=1000, random_seed=42)
1782:     assert unc_high.is_high_uncertainty(threshold=0.1)
1783:
1784:
1785: class TestProvenanceDAG:
1786:     """Test DAG provenance tracking."""
1787:
1788:     def test_dag_creation_and_node_addition(self):
1789:         """DAG should accept nodes and maintain structure."""
1790:         dag = AggregationDAG()
1791:
1792:         node1 = ProvenanceNode(node_id="Q001", level="micro", score=7.0, quality_level="BUENO")
```

```
1793:     node2 = ProvenanceNode(node_id="Q002", level="micro", score=8.0, quality_level="EXCELENTE")
1794:
1795:     dag.add_node(node1)
1796:     dag.add_node(node2)
1797:
1798:     assert "Q001" in dag.nodes
1799:     assert "Q002" in dag.nodes
1800:     assert dag.graph.number_of_nodes() == 2
1801:
1802: def test_dag_aggregation_edges(self):
1803:     """DAG should record aggregation operations."""
1804:     dag = AggregationDAG()
1805:
1806:     # Add source nodes
1807:     for i in range(3):
1808:         node = ProvenanceNode(
1809:             node_id=f"Q{i:03d}",
1810:             level="micro",
1811:             score=7.0 + i,
1812:             quality_level="BUENO",
1813:         )
1814:         dag.add_node(node)
1815:
1816:     # Add target node
1817:     target = ProvenanceNode(
1818:         node_id="DIM01_PA01",
1819:         level="dimension",
1820:         score=7.5,
1821:         quality_level="BUENO",
1822:     )
1823:     dag.add_node(target)
1824:
1825:     # Record aggregation
1826:     dag.add_aggregation_edge(
1827:         source_ids=["Q000", "Q001", "Q002"],
1828:         target_id="DIM01_PA01",
1829:         operation="weighted_average",
1830:         weights=[0.33, 0.33, 0.34],
1831:     )
1832:
1833:     assert dag.graph.number_of_edges() == 3
1834:     assert dag.graph.has_edge("Q000", "DIM01_PA01")
1835:
1836: def test_dag_cycle_detection(self):
1837:     """DAG should reject edges that create cycles."""
1838:     dag = AggregationDAG()
1839:
1840:     dag.add_node(ProvenanceNode(node_id="A", level="micro", score=7.0, quality_level="BUENO"))
1841:     dag.add_node(ProvenanceNode(node_id="B", level="dimension", score=7.5, quality_level="BUENO"))
1842:
1843:     # A \206\222 B is valid
1844:     dag.add_aggregation_edge(["A"], "B", "weighted_average", [1.0])
1845:
1846:     # B \206\222 A would create cycle, should raise
1847:     with pytest.raises(ValueError, match="cycle"):
1848:         dag.add_aggregation_edge(["B"], "A", "weighted_average", [1.0])
```

```
1849:  
1850:     def test_lineage_tracing(self):  
1851:         """Should trace complete lineage of a target node."""  
1852:         dag = AggregationDAG()  
1853:  
1854:         # Build hierarchy: Q001, Q002 → \206\222 DIM01 → \206\222 AREA01  
1855:         for qid in ["Q001", "Q002"]:  
1856:             dag.add_node(ProvenanceNode(node_id=qid, level="micro", score=7.0, quality_level="BUENO"))  
1857:  
1858:             dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.5, quality_level="BUENO"))  
1859:             dag.add_aggregation_edge(["Q001", "Q002"], "DIM01", "weighted_average", [0.5, 0.5])  
1860:  
1861:             dag.add_node(ProvenanceNode(node_id="AREA01", level="area", score=7.5, quality_level="BUENO"))  
1862:             dag.add_aggregation_edge(["DIM01"], "AREA01", "weighted_average", [1.0])  
1863:  
1864:         # Trace AREA01  
1865:         lineage = dag.trace_lineage("AREA01")  
1866:  
1867:         assert lineage["ancestor_count"] == 3 # Q001, Q002, DIM01  
1868:         assert "Q001" in lineage["ancestors"]  
1869:         assert "Q002" in lineage["ancestors"]  
1870:         assert "DIM01" in lineage["ancestors"]  
1871:         assert lineage["micro_question_count"] == 2  
1872:         # depth can be fractional due to average path lengths, just check > 1  
1873:         assert lineage["depth"] >= 1  
1874:  
1875:     def test_shapley_attribution(self):  
1876:         """Shapley values should sum to target score."""  
1877:         dag = AggregationDAG()  
1878:  
1879:         # Add sources  
1880:         scores = [7.0, 8.0, 6.5]  
1881:         weights = [0.3, 0.4, 0.3]  
1882:         for i, score in enumerate(scores):  
1883:             dag.add_node(ProvenanceNode(  
1884:                 node_id=f"Q{i:03d}",  
1885:                 level="micro",  
1886:                 score=score,  
1887:                 quality_level="BUENO",  
1888:             ))  
1889:  
1890:         # Add target  
1891:         target_score = sum(s * w for s, w in zip(scores, weights))  
1892:         dag.add_node(ProvenanceNode(  
1893:             node_id="DIM01",  
1894:             level="dimension",  
1895:             score=target_score,  
1896:             quality_level="BUENO",  
1897:         ))  
1898:  
1899:         dag.add_aggregation_edge(  
1900:             ["Q000", "Q001", "Q002"],  
1901:             "DIM01",  
1902:             "weighted_average",  
1903:             weights,  
1904:         )
```

```
1905:  
1906:     # Compute Shapley  
1907:     attribution = dag.compute_shapley_attribution("DIM01")  
1908:  
1909:     # Shapley values should sum to target score (approximately)  
1910:     total_attribution = sum(attribution.values())  
1911:     assert abs(total_attribution - target_score) < 0.01  
1912:  
1913:     # Highest weight should have highest attribution  
1914:     max_weight_idx = weights.index(max(weights))  
1915:     max_attr_node = max(attribution, key=attribution.get)  
1916:     assert max_attr_node == f"Q{max_weight_idx:03d}"  
1917:  
1918: def test_critical_path_identification(self):  
1919:     """Should identify top-k most important inputs."""  
1920:     dag = AggregationDAG()  
1921:  
1922:     scores = [9.0, 7.0, 8.0, 6.0, 7.5]  
1923:     weights = [0.3, 0.2, 0.25, 0.1, 0.15]  
1924:  
1925:     for i, score in enumerate(scores):  
1926:         dag.add_node(ProvenanceNode(  
1927:             node_id=f"Q{i:03d}",  
1928:             level="micro",  
1929:             score=score,  
1930:             quality_level="BUENO",  
1931:         ))  
1932:  
1933:     target_score = sum(s * w for s, w in zip(scores, weights))  
1934:     dag.add_node(ProvenanceNode(  
1935:         node_id="DIM01",  
1936:         level="dimension",  
1937:         score=target_score,  
1938:         quality_level="BUENO",  
1939:     ))  
1940:  
1941:     dag.add_aggregation_edge(  
1942:         [f"Q{i:03d}" for i in range(5)],  
1943:         "DIM01",  
1944:         "weighted_average",  
1945:         weights,  
1946:     )  
1947:  
1948:     critical_path = dag.get_critical_path("DIM01", top_k=3)  
1949:  
1950:     assert len(critical_path) == 3  
1951:     # Top contributor should be Q000 (highest weight * highest score)  
1952:     assert critical_path[0][0] == "Q000"  
1953:  
1954: def test_graphml_export(self, tmp_path):  
1955:     """Should export DAG to GraphML format."""  
1956:     dag = AggregationDAG()  
1957:  
1958:     dag.add_node(ProvenanceNode(node_id="Q001", level="micro", score=7.0, quality_level="BUENO"))  
1959:     dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.0, quality_level="BUENO"))  
1960:     dag.add_aggregation_edge(["Q001"], "DIM01", "weighted_average", [1.0])
```

```
1961:  
1962:     output_path = tmp_path / "test_dag.graphml"  
1963:     dag.export_graphml(str(output_path))  
1964:  
1965:     assert output_path.exists()  
1966:     # Verify it's valid XML  
1967:     with open(output_path) as f:  
1968:         content = f.read()  
1969:         assert "<?xml" in content  
1970:         assert "<graphml" in content  
1971:  
1972:     def test_prov_json_export(self, tmp_path):  
1973:         """Should export to W3C PROV-JSON format."""  
1974:         dag = AggregationDAG()  
1975:  
1976:         dag.add_node(ProvenanceNode(node_id="Q001", level="micro", score=7.0, quality_level="BUENO"))  
1977:         dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.0, quality_level="BUENO"))  
1978:         dag.add_aggregation_edge(["Q001"], "DIM01", "weighted_average", [1.0])  
1979:  
1980:         output_path = tmp_path / "provenance.json"  
1981:         dag.export_prov_json(str(output_path))  
1982:  
1983:         assert output_path.exists()  
1984:  
1985:         import json  
1986:         with open(output_path) as f:  
1987:             prov_doc = json.load(f)  
1988:  
1989:             assert "prefix" in prov_doc  
1990:             assert "entity" in prov_doc  
1991:             assert "activity" in prov_doc  
1992:             assert "Q001" in prov_doc["entity"]  
1993:  
1994:     def test_dag_statistics(self):  
1995:         """Should compute graph statistics."""  
1996:         dag = AggregationDAG()  
1997:  
1998:         for i in range(5):  
1999:             dag.add_node(ProvenanceNode(  
2000:                 node_id=f"Q{i:03d}",  
2001:                 level="micro",  
2002:                 score=7.0,  
2003:                 quality_level="BUENO",  
2004:             ))  
2005:  
2006:         dag.add_node(ProvenanceNode(node_id="DIM01", level="dimension", score=7.5, quality_level="BUENO"))  
2007:         dag.add_aggregation_edge(  
2008:             [f"Q{i:03d}" for i in range(5)],  
2009:             "DIM01",  
2010:             "weighted_average",  
2011:             [0.2] * 5,  
2012:         )  
2013:  
2014:         stats = dag.get_statistics()  
2015:  
2016:         assert stats["node_count"] == 6
```

```
2017:         assert stats["edge_count"] == 5
2018:         assert stats["is_dag"] is True
2019:         assert stats["nodes_by_level"]["micro"] == 5
2020:         assert stats["nodes_by_level"]["dimension"] == 1
2021:
2022:
2023: class TestSensitivityAnalysis:
2024:     """Test Sobol sensitivity analysis."""
2025:
2026:     def test_sobol_indices_sum_to_one(self):
2027:         """First-order Sobol indices should sum to ~1.0."""
2028:         scores = [7.0, 8.0, 6.5, 7.5]
2029:         weights = [0.25, 0.25, 0.25, 0.25]
2030:
2031:         sobol = SensitivityAnalysis.compute_sobol_indices(
2032:             scores, weights, n_samples=500, random_seed=42
2033:         )
2034:
2035:         total = sum(sobol.values())
2036:         assert abs(total - 1.0) < 0.05 # Allow 5% tolerance for MC error
2037:
2038:     def test_sobol_identifies_influential_inputs(self):
2039:         """Inputs with high variance should have high Sobol index."""
2040:         # Input 0 has high weight but low variance
2041:         # Input 1 has medium weight and high variance (will be perturbed more)
2042:         scores = [7.0, 7.0, 7.0]
2043:         weights = [0.5, 0.3, 0.2]
2044:
2045:         sobol = SensitivityAnalysis.compute_sobol_indices(
2046:             scores, weights, n_samples=500, random_seed=42
2047:         )
2048:
2049:         # All have same score, so sensitivity is proportional to weight^2
2050:         # (for weighted average, Sobol ~ w^2 * var, but var is introduced by MC)
2051:         assert len(sobol) == 3
2052:
2053:
2054: if __name__ == "__main__":
2055:     pytest.main([__file__, "-v", "--tb=short"])
2056:
2057:
2058:
2059: =====
2060: FILE: tests/test_batch_executor.py
2061: =====
2062:
2063: """Tests for batch executor infrastructure."""
2064:
2065: from pathlib import Path
2066: from unittest.mock import MagicMock
2067:
2068: import pytest
2069:
2070: from farfan_pipeline.core.orchestrator.batch_executor import (
2071:     COMPLEXITY_BATCH_SIZE_MAP,
2072:     EXECUTOR_COMPLEXITY_MAP,
```

```
2073:     AggregatedBatchResults,
2074:     BatchExecutor,
2075:     BatchExecutorConfig,
2076:     BatchResult,
2077:     BatchStatus,
2078:     ExecutorComplexity,
2079: )
2080: from farfan_pipeline.core.types import PreprocessedDocument
2081:
2082:
2083: @pytest.fixture
2084: def batch_config():
2085:     """Create batch executor configuration."""
2086:     return BatchExecutorConfig(
2087:         default_batch_size=10,
2088:         max_batch_size=50,
2089:         min_batch_size=2,
2090:         enable_streaming=True,
2091:         error_threshold=0.5,
2092:         max_retries=2,
2093:         enable_instrumentation=False,
2094:     )
2095:
2096:
2097: @pytest.fixture
2098: def mock_method_executor():
2099:     """Create mock method executor."""
2100:     executor = MagicMock()
2101:     executor.signal_registry = MagicMock()
2102:     return executor
2103:
2104:
2105: @pytest.fixture
2106: def sample_document():
2107:     """Create sample preprocessed document."""
2108:     return PreprocessedDocument(
2109:         document_id="test_doc_001",
2110:         raw_text="Sample document text for testing batch processing.",
2111:         sentences=["Sample document text for testing batch processing."],
2112:         tables=[],
2113:         metadata={"source": "test"},
2114:         source_path=Path("/tmp/test.pdf"),
2115:     )
2116:
2117:
2118: @pytest.fixture
2119: def sample_question_context():
2120:     """Create sample question context."""
2121:     return {
2122:         "question_id": "D1-Q1",
2123:         "question_global": 1,
2124:         "base_slot": "D1-Q1",
2125:         "dimension_id": "DIM01",
2126:         "policy_area_id": "PA01",
2127:         "cluster_id": "CLU01",
2128:         "scoring_modality": "quantitative",
```

```
2129:         "expected_elements": ["baseline", "metrics"],
2130:     }
2131:
2132:
2133: class MockExecutor:
2134:     """Mock executor for testing."""
2135:
2136:     def __init__(
2137:         self,
2138:         method_executor,
2139:         signal_registry,
2140:         config,
2141:         questionnaire_provider,
2142:         calibration_orchestrator,
2143:     ):
2144:         self.method_executor = method_executor
2145:         self.signal_registry = signal_registry
2146:         self.config = config
2147:         self.questionnaire_provider = questionnaire_provider
2148:         self.calibration_orchestrator = calibration_orchestrator
2149:         self.call_count = 0
2150:
2151:     def execute(self, document, method_executor, question_context):
2152:         """Mock execute method."""
2153:         self.call_count += 1
2154:         return {
2155:             "evidence": {"baseline": "test_value"},
2156:             "metadata": {"executed": True},
2157:             "call_count": self.call_count,
2158:         }
2159:
2160:
2161: class FailingExecutor:
2162:     """Mock executor that fails."""
2163:
2164:     def __init__(
2165:         self,
2166:         method_executor,
2167:         signal_registry,
2168:         config,
2169:         questionnaire_provider,
2170:         calibration_orchestrator,
2171:     ):
2172:         self.method_executor = method_executor
2173:         self.signal_registry = signal_registry
2174:
2175:     def execute(self, document, method_executor, question_context):
2176:         """Mock execute that always fails."""
2177:         raise ValueError("Intentional test failure")
2178:
2179:
2180: class PartialFailExecutor:
2181:     """Mock executor that fails every other call."""
2182:
2183:     def __init__(
2184:         self,
```

```
2185:         method_executor,
2186:         signal_registry,
2187:         config,
2188:         questionnaire_provider,
2189:         calibration_orchestrator,
2190:     ):
2191:         self.method_executor = method_executor
2192:         self.call_count = 0
2193:
2194:     def execute(self, document, method_executor, question_context):
2195:         """Mock execute that fails every other call."""
2196:         self.call_count += 1
2197:         if self.call_count % 2 == 0:
2198:             raise ValueError(f"Intentional failure on call {self.call_count}")
2199:         return {"evidence": {"data": f"success_{self.call_count}"}}
2200:
2201:
2202: class TestBatchExecutorConfig:
2203:     """Test batch executor configuration."""
2204:
2205:     def test_valid_config(self):
2206:         """Test valid configuration."""
2207:         config = BatchExecutorConfig(
2208:             default_batch_size=10,
2209:             max_batch_size=100,
2210:             min_batch_size=1,
2211:             error_threshold=0.3,
2212:         )
2213:         assert config.default_batch_size == 10
2214:         assert config.max_batch_size == 100
2215:         assert config.min_batch_size == 1
2216:         assert config.error_threshold == 0.3
2217:
2218:     def test_invalid_batch_sizes(self):
2219:         """Test invalid batch size configuration."""
2220:         with pytest.raises(ValueError, match="Invalid batch size configuration"):
2221:             BatchExecutorConfig(
2222:                 default_batch_size=5,
2223:                 max_batch_size=3,
2224:                 min_batch_size=1,
2225:             )
2226:
2227:     def test_invalid_error_threshold(self):
2228:         """Test invalid error threshold."""
2229:         with pytest.raises(ValueError, match="error_threshold must be in"):
2230:             BatchExecutorConfig(error_threshold=1.5)
2231:
2232:     def test_default_values(self):
2233:         """Test default configuration values."""
2234:         config = BatchExecutorConfig()
2235:         assert config.default_batch_size == 10
2236:         assert config.max_batch_size == 100
2237:         assert config.min_batch_size == 1
2238:         assert config.error_threshold == 0.5
2239:         assert config.max_retries == 2
2240:         assert config.enable_streaming is True
```

```
2241:  
2242:  
2243: class TestComplexityMapping:  
2244:     """Test executor complexity mapping."""  
2245:  
2246:     def test_complexity_map_coverage(self):  
2247:         """Test that all major executors have complexity mappings."""  
2248:         assert "D1-Q1" in EXECUTOR_COMPLEXITY_MAP  
2249:         assert "D3-Q2" in EXECUTOR_COMPLEXITY_MAP  
2250:         assert "D5-Q2" in EXECUTOR_COMPLEXITY_MAP  
2251:  
2252:     def test_complexity_batch_size_map(self):  
2253:         """Test complexity to batch size mapping."""  
2254:         assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.SIMPLE] == 50  
2255:         assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.MODERATE] == 20  
2256:         assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.COMPLEX] == 10  
2257:         assert COMPLEXITY_BATCH_SIZE_MAP[ExecutorComplexity.VERY_COMPLEX] == 5  
2258:  
2259:     def test_very_complex_executors(self):  
2260:         """Test that intensive executors are marked as very complex."""  
2261:         very_complex = [  
2262:             slot  
2263:             for slot, complexity in EXECUTOR_COMPLEXITY_MAP.items()  
2264:             if complexity == ExecutorComplexity.VERY_COMPLEX  
2265:         ]  
2266:         assert "D2-Q2" in very_complex  
2267:         assert "D3-Q2" in very_complex  
2268:         assert "D3-Q3" in very_complex  
2269:  
2270:  
2271: class TestBatchExecutorBasics:  
2272:     """Test basic batch executor functionality."""  
2273:  
2274:     def test_initialization(self, batch_config, mock_method_executor):  
2275:         """Test batch executor initialization."""  
2276:         executor = BatchExecutor(  
2277:             config=batch_config,  
2278:             method_executor=mock_method_executor,  
2279:         )  
2280:         assert executor.config == batch_config  
2281:         assert executor.method_executor == mock_method_executor  
2282:         assert executor._batch_counter == 0  
2283:  
2284:     def test_get_batch_size_for_executor(self, batch_config, mock_method_executor):  
2285:         """Test batch size determination based on complexity."""  
2286:         executor = BatchExecutor(  
2287:             config=batch_config, method_executor=mock_method_executor  
2288:         )  
2289:  
2290:         batch_size_moderate = executor.get_batch_size_for_executor("D1-Q1")  
2291:         assert batch_size_moderate == 20  
2292:  
2293:         batch_size_complex = executor.get_batch_size_for_executor("D3-Q2")  
2294:         assert batch_size_complex == 5  
2295:  
2296:         batch_size_unknown = executor.get_batch_size_for_executor("UNKNOWN")
```

```
2297:         assert batch_size_unknown == 20
2298:
2299:     def test_batch_size_clamping(self, mock_method_executor):
2300:         """Test that batch sizes respect min/max constraints."""
2301:         config = BatchExecutorConfig(
2302:             default_batch_size=10, max_batch_size=15, min_batch_size=8
2303:         )
2304:         executor = BatchExecutor(config=config, method_executor=mock_method_executor)
2305:
2306:         batch_size = executor.get_batch_size_for_executor("D1-Q1")
2307:         assert 8 <= batch_size <= 15
2308:
2309:     def test_create_batches(self, batch_config, mock_method_executor):
2310:         """Test batch creation from items."""
2311:         executor = BatchExecutor(
2312:             config=batch_config, method_executor=mock_method_executor
2313:         )
2314:
2315:         items = list(range(25))
2316:         batches = executor._create_batches(items, batch_size=10)
2317:
2318:         assert len(batches) == 3
2319:         assert batches[0] == (0, list(range(0, 10)))
2320:         assert batches[1] == (1, list(range(10, 20)))
2321:         assert batches[2] == (2, list(range(20, 25)))
2322:
2323:     def test_create_batches_exact_fit(self, batch_config, mock_method_executor):
2324:         """Test batch creation with exact fit."""
2325:         executor = BatchExecutor(
2326:             config=batch_config, method_executor=mock_method_executor
2327:         )
2328:
2329:         items = list(range(20))
2330:         batches = executor._create_batches(items, batch_size=10)
2331:
2332:         assert len(batches) == 2
2333:         assert all(len(batch[1]) == 10 for batch in batches)
2334:
2335:     def test_create_batches_single_batch(self, batch_config, mock_method_executor):
2336:         """Test batch creation with items fitting in single batch."""
2337:         executor = BatchExecutor(
2338:             config=batch_config, method_executor=mock_method_executor
2339:         )
2340:
2341:         items = list(range(5))
2342:         batches = executor._create_batches(items, batch_size=10)
2343:
2344:         assert len(batches) == 1
2345:         assert batches[0] == (0, items)
2346:
2347:
2348: class TestBatchExecution:
2349:     """Test batch execution."""
2350:
2351:     @pytest.mark.asyncio
2352:     async def test_execute_single_batch_success(self):
```

```
2353:         self,
2354:         batch_config,
2355:         mock_method_executor,
2356:         sample_document,
2357:         sample_question_context,
2358:     ):
2359:         """Test successful execution of a single batch."""
2360:         executor = BatchExecutor(
2361:             config=batch_config, method_executor=mock_method_executor
2362:         )
2363:         mock_executor_instance = MockExecutor(
2364:             mock_method_executor, None, batch_config, None, None
2365:         )
2366:
2367:         batch_items = [1, 2, 3]
2368:         result = await executor._execute_single_batch(
2369:             "test_batch_001",
2370:             0,
2371:             batch_items,
2372:             mock_executor_instance,
2373:             sample_document,
2374:             sample_question_context,
2375:         )
2376:
2377:         assert result.batch_id == "test_batch_001"
2378:         assert result.batch_index == 0
2379:         assert result.status == BatchStatus.COMPLETED
2380:         assert result.metrics["successful_items"] == 3
2381:         assert result.metrics["failed_items"] == 0
2382:         assert len(result.results) == 3
2383:         assert all(r is not None for r in result.results)
2384:
2385:     @pytest.mark.asyncio
2386:     async def test_execute_single_batch_all_failures(
2387:         self,
2388:         batch_config,
2389:         mock_method_executor,
2390:         sample_document,
2391:         sample_question_context,
2392:     ):
2393:         """Test batch execution with all failures."""
2394:         executor = BatchExecutor(
2395:             config=batch_config, method_executor=mock_method_executor
2396:         )
2397:         mock_executor_instance = FailingExecutor(
2398:             mock_method_executor, None, batch_config, None, None
2399:         )
2400:
2401:         batch_items = [1, 2, 3]
2402:         result = await executor._execute_single_batch(
2403:             "test_batch_002",
2404:             0,
2405:             batch_items,
2406:             mock_executor_instance,
2407:             sample_document,
2408:             sample_question_context,
```

```
2409:         )
2410:
2411:     assert result.status == BatchStatus.FAILED
2412:     assert result.metrics["successful_items"] == 0
2413:     assert result.metrics["failed_items"] == 3
2414:     assert len(result.errors) == 3
2415:     assert all(r is None for r in result.results)
2416:
2417:     @pytest.mark.asyncio
2418:     async def test_execute_single_batch_partial_success(
2419:         self, mock_method_executor, sample_document, sample_question_context
2420):
2421:         """Test batch execution with partial success."""
2422:         config = BatchExecutorConfig(error_threshold=0.6, enable_instrumentation=False)
2423:         executor = BatchExecutor(config=config, method_executor=mock_method_executor)
2424:         mock_executor_instance = PartialFailExecutor(
2425:             mock_method_executor, None, config, None, None
2426:         )
2427:
2428:         batch_items = [1, 2, 3, 4]
2429:         result = await executor._execute_single_batch(
2430:             "test_batch_003",
2431:             0,
2432:             batch_items,
2433:             mock_executor_instance,
2434:             sample_document,
2435:             sample_question_context,
2436:         )
2437:
2438:         assert result.status == BatchStatus.PARTIAL_SUCCESS
2439:         assert result.metrics["successful_items"] == 2
2440:         assert result.metrics["failed_items"] == 2
2441:         assert len(result.get_successful_results()) == 2
2442:         assert len(result.get_failed_items()) == 2
2443:
2444:     @pytest.mark.asyncio
2445:     async def test_execute_batch_with_retry_success_first_try(
2446:         self,
2447:         batch_config,
2448:         mock_method_executor,
2449:         sample_document,
2450:         sample_question_context,
2451):
2452:         """Test batch retry logic with success on first try."""
2453:         executor = BatchExecutor(
2454:             config=batch_config, method_executor=mock_method_executor
2455:         )
2456:         mock_executor_instance = MockExecutor(
2457:             mock_method_executor, None, batch_config, None, None
2458:         )
2459:
2460:         batch_items = [1, 2]
2461:         result = await executor._execute_batch_with_retry(
2462:             "test_batch_004",
2463:             0,
2464:             batch_items,
```

```
2465:             mock_executor_instance,
2466:             sample_document,
2467:             sample_question_context,
2468:         )
2469:
2470:         assert result.status == BatchStatus.COMPLETED
2471:         assert result.metrics["retries_used"] == 0
2472:
2473:     @pytest.mark.asyncio
2474:     async def test_execute_batch_with_retry_exhausted(
2475:         self, mock_method_executor, sample_document, sample_question_context
2476:     ):
2477:         """Test batch retry logic with retries exhausted."""
2478:         config = BatchExecutorConfig(max_retries=1, enable_instrumentation=False)
2479:         executor = BatchExecutor(config=config, method_executor=mock_method_executor)
2480:         mock_executor_instance = FailingExecutor(
2481:             mock_method_executor, None, config, None, None
2482:         )
2483:
2484:         batch_items = [1, 2]
2485:         result = await executor._execute_batch_with_retry(
2486:             "test_batch_005",
2487:             0,
2488:             batch_items,
2489:             mock_executor_instance,
2490:             sample_document,
2491:             sample_question_context,
2492:         )
2493:
2494:         assert result.status == BatchStatus.FAILED
2495:         assert result.metrics["retries_used"] == 1
2496:
2497:
2498: class TestStreamingBatchExecution:
2499:     """Test streaming batch execution."""
2500:
2501:     @pytest.mark.asyncio
2502:     async def test_execute_batches_streaming(
2503:         self,
2504:         batch_config,
2505:         mock_method_executor,
2506:         sample_document,
2507:         sample_question_context,
2508:     ):
2509:         """Test streaming batch execution."""
2510:         executor = BatchExecutor(
2511:             config=batch_config, method_executor=mock_method_executor
2512:         )
2513:
2514:         items = list(range(25))
2515:         batch_results = []
2516:
2517:         async for batch_result in executor.execute_batches(
2518:             items, MockExecutor, sample_document, sample_question_context, batch_size=10
2519:         ):
2520:             batch_results.append(batch_result)
```

```
2521:  
2522:     assert len(batch_results) == 3  
2523:     assert batch_results[0].batch_index == 0  
2524:     assert batch_results[1].batch_index == 1  
2525:     assert batch_results[2].batch_index == 2  
2526:     assert all(br.status == BatchStatus.COMPLETED for br in batch_results)  
2527:  
2528:     @pytest.mark.asyncio  
2529:     async def test_execute_batches_with_base_slot(  
2530:         self,  
2531:         batch_config,  
2532:         mock_method_executor,  
2533:         sample_document,  
2534:         sample_question_context,  
2535:     ):  
2536:         """Test batch execution with base slot for complexity-based sizing."""  
2537:         executor = BatchExecutor(  
2538:             config=batch_config, method_executor=mock_method_executor  
2539:         )  
2540:  
2541:         items = list(range(30))  
2542:         batch_results = []  
2543:  
2544:         async for batch_result in executor.execute_batches(  
2545:             items,  
2546:             MockExecutor,  
2547:             sample_document,  
2548:             sample_question_context,  
2549:             base_slot="D3-Q2",  
2550:         ):  
2551:             batch_results.append(batch_result)  
2552:  
2553:         assert len(batch_results) == 6  
2554:  
2555:     @pytest.mark.asyncio  
2556:     async def test_execute_batches_empty_items(  
2557:         self,  
2558:         batch_config,  
2559:         mock_method_executor,  
2560:         sample_document,  
2561:         sample_question_context,  
2562:     ):  
2563:         """Test batch execution with empty items list."""  
2564:         executor = BatchExecutor(  
2565:             config=batch_config, method_executor=mock_method_executor  
2566:         )  
2567:  
2568:         batch_results = []  
2569:         async for batch_result in executor.execute_batches(  
2570:             [], MockExecutor, sample_document, sample_question_context  
2571:         ):  
2572:             batch_results.append(batch_result)  
2573:  
2574:         assert len(batch_results) == 0  
2575:  
2576:
```

```
2577: class TestParallelBatchExecution:
2578:     """Test parallel batch execution."""
2579:
2580:     @pytest.mark.asyncio
2581:     async def test_execute_batches_parallel(
2582:         self,
2583:         batch_config,
2584:         mock_method_executor,
2585:         sample_document,
2586:         sample_question_context,
2587:     ):
2588:         """Test parallel batch execution."""
2589:         executor = BatchExecutor(
2590:             config=batch_config, method_executor=mock_method_executor
2591:         )
2592:
2593:         items = list(range(40))
2594:         batch_results = await executor.execute_batches_parallel(
2595:             items,
2596:             MockExecutor,
2597:             sample_document,
2598:             sample_question_context,
2599:             batch_size=10,
2600:             max_concurrent_batches=2,
2601:         )
2602:
2603:         assert len(batch_results) == 4
2604:         assert all(br.status == BatchStatus.COMPLETED for br in batch_results)
2605:         assert sum(br.metrics["successful_items"] for br in batch_results) == 40
2606:
2607:     @pytest.mark.asyncio
2608:     async def test_execute_batches_parallel_with_failures(
2609:         self,
2610:         batch_config,
2611:         mock_method_executor,
2612:         sample_document,
2613:         sample_question_context,
2614:     ):
2615:         """Test parallel batch execution with some failures."""
2616:         executor = BatchExecutor(
2617:             config=batch_config, method_executor=mock_method_executor
2618:         )
2619:
2620:         items = list(range(20))
2621:         batch_results = await executor.execute_batches_parallel(
2622:             items,
2623:             PartialFailExecutor,
2624:             sample_document,
2625:             sample_question_context,
2626:             batch_size=10,
2627:             max_concurrent_batches=2,
2628:         )
2629:
2630:         assert len(batch_results) == 2
2631:         successful_total = sum(br.metrics["successful_items"] for br in batch_results)
2632:         failed_total = sum(br.metrics["failed_items"] for br in batch_results)
```

```
2633:         assert successful_total == 10
2634:         assert failed_total == 10
2635:
2636:
2637: class TestBatchAggregation:
2638:     """Test batch result aggregation."""
2639:
2640:     @pytest.mark.asyncio
2641:     async def test_aggregate_batch_results(self, batch_config, mock_method_executor):
2642:         """Test aggregation of batch results."""
2643:         executor = BatchExecutor(
2644:             config=batch_config, method_executor=mock_method_executor
2645:         )
2646:
2647:         batch_results = [
2648:             BatchResult(
2649:                 batch_id="batch_001",
2650:                 batch_index=0,
2651:                 items=[1, 2, 3],
2652:                 results=[{"a": 1}, {"a": 2}, {"a": 3}],
2653:                 status=BatchStatus.COMPLETED,
2654:                 metrics={
2655:                     "batch_id": "batch_001",
2656:                     "batch_index": 0,
2657:                     "batch_size": 3,
2658:                     "status": BatchStatus.COMPLETED,
2659:                     "start_time": 0.0,
2660:                     "end_time": 1.0,
2661:                     "execution_time_ms": 1000.0,
2662:                     "successful_items": 3,
2663:                     "failed_items": 0,
2664:                     "retries_used": 0,
2665:                     "error_messages": [],
2666:                 },
2667:             ),
2668:             BatchResult(
2669:                 batch_id="batch_002",
2670:                 batch_index=1,
2671:                 items=[4, 5],
2672:                 results=[{"a": 4}, None],
2673:                 status=BatchStatus.PARTIAL_SUCCESS,
2674:                 metrics={
2675:                     "batch_id": "batch_002",
2676:                     "batch_index": 1,
2677:                     "batch_size": 2,
2678:                     "status": BatchStatus.PARTIAL_SUCCESS,
2679:                     "start_time": 1.0,
2680:                     "end_time": 2.0,
2681:                     "execution_time_ms": 1000.0,
2682:                     "successful_items": 1,
2683:                     "failed_items": 1,
2684:                     "retries_used": 1,
2685:                     "error_messages": ["error"],
2686:                 },
2687:                 errors=[{"error": "test"}],
2688:             ),

```

```
2689:         ]
2690:
2691:     aggregated = await executor.aggregate_batch_results(batch_results)
2692:
2693:     assert aggregated.total_batches == 2
2694:     assert aggregated.total_items == 5
2695:     assert aggregated.successful_items == 4
2696:     assert aggregated.failed_items == 1
2697:     assert len(aggregated.results) == 4
2698:     assert len(aggregated.errors) == 1
2699:     assert aggregated.success_rate() == 0.8
2700:
2701:     @pytest.mark.asyncio
2702:     async def test_aggregate_empty_results(self, batch_config, mock_method_executor):
2703:         """Test aggregation with no results."""
2704:         executor = BatchExecutor(
2705:             config=batch_config, method_executor=mock_method_executor
2706:         )
2707:
2708:         aggregated = await executor.aggregate_batch_results([])
2709:
2710:         assert aggregated.total_batches == 0
2711:         assert aggregated.total_items == 0
2712:         assert aggregated.success_rate() == 0.0
2713:
2714:     @pytest.mark.asyncio
2715:     async def test_stream_aggregate_batches(
2716:         self,
2717:         batch_config,
2718:         mock_method_executor,
2719:         sample_document,
2720:         sample_question_context,
2721:     ):
2722:         """Test streaming aggregation of batch results."""
2723:         executor = BatchExecutor(
2724:             config=batch_config, method_executor=mock_method_executor
2725:         )
2726:
2727:         items = list(range(15))
2728:
2729:         def accumulator_fn(acc, result):
2730:             """Accumulate call counts."""
2731:             call_count = result.get("call_count", 0)
2732:             if "total_calls" not in acc:
2733:                 acc["total_calls"] = 0
2734:             acc["total_calls"] += call_count
2735:
2736:             return acc
2737:
2738:         batch_stream = executor.execute_batches(
2739:             items, MockExecutor, sample_document, sample_question_context, batch_size=5
2740:         )
2741:
2742:         final = await executor.stream_aggregate_batches(batch_stream, accumulator_fn)
2743:
2744:         assert final is not None
2745:         assert "total_calls" in final
```

```
2745:  
2746:  
2747: class TestBatchResultHelpers:  
2748:     """Test BatchResult helper methods."""  
2749:  
2750:     def test_is_successful(self):  
2751:         """Test is_successful helper."""  
2752:         result = BatchResult(  
2753:             batch_id="test",  
2754:             batch_index=0,  
2755:             items=[1, 2],  
2756:             results=[{"a": 1}, {"a": 2}],  
2757:             status=BatchStatus.COMPLETED,  
2758:             metrics={  
2759:                 "batch_id": "test",  
2760:                 "batch_index": 0,  
2761:                 "batch_size": 2,  
2762:                 "status": BatchStatus.COMPLETED,  
2763:                 "start_time": 0.0,  
2764:                 "end_time": 1.0,  
2765:                 "execution_time_ms": 1000.0,  
2766:                 "successful_items": 2,  
2767:                 "failed_items": 0,  
2768:                 "retries_used": 0,  
2769:                 "error_messages": [],  
2770:             },  
2771:         )  
2772:         assert result.is_successful() is True  
2773:  
2774:     def test_has_partial_success(self):  
2775:         """Test has_partial_success helper."""  
2776:         result = BatchResult(  
2777:             batch_id="test",  
2778:             batch_index=0,  
2779:             items=[1, 2],  
2780:             results=[{"a": 1}, None],  
2781:             status=BatchStatus.PARTIAL_SUCCESS,  
2782:             metrics={  
2783:                 "batch_id": "test",  
2784:                 "batch_index": 0,  
2785:                 "batch_size": 2,  
2786:                 "status": BatchStatus.PARTIAL_SUCCESS,  
2787:                 "start_time": 0.0,  
2788:                 "end_time": 1.0,  
2789:                 "execution_time_ms": 1000.0,  
2790:                 "successful_items": 1,  
2791:                 "failed_items": 1,  
2792:                 "retries_used": 0,  
2793:                 "error_messages": ["error"],  
2794:             },  
2795:         )  
2796:         assert result.has_partial_success() is True  
2797:  
2798:     def test_get_successful_results(self):  
2799:         """Test get_successful_results helper."""  
2800:         result = BatchResult(
```

```
2801:         batch_id="test",
2802:         batch_index=0,
2803:         items=[1, 2, 3],
2804:         results=[{"a": 1}, None, {"a": 3}],
2805:         status=BatchStatus.PARTIAL_SUCCESS,
2806:         metrics={
2807:             "batch_id": "test",
2808:             "batch_index": 0,
2809:             "batch_size": 3,
2810:             "status": BatchStatus.PARTIAL_SUCCESS,
2811:             "start_time": 0.0,
2812:             "end_time": 1.0,
2813:             "execution_time_ms": 1000.0,
2814:             "successful_items": 2,
2815:             "failed_items": 1,
2816:             "retries_used": 0,
2817:             "error_messages": ["error"],
2818:         },
2819:     )
2820:     successful = result.get_successful_results()
2821:     assert len(successful) == 2
2822:     assert successful[0] == (1, {"a": 1})
2823:     assert successful[1] == (3, {"a": 3})
2824:
2825: def test_get_failed_items(self):
2826:     """Test get_failed_items helper."""
2827:     result = BatchResult(
2828:         batch_id="test",
2829:         batch_index=0,
2830:         items=[1, 2, 3],
2831:         results=[{"a": 1}, None, {"a": 3}],
2832:         status=BatchStatus.PARTIAL_SUCCESS,
2833:         metrics={
2834:             "batch_id": "test",
2835:             "batch_index": 0,
2836:             "batch_size": 3,
2837:             "status": BatchStatus.PARTIAL_SUCCESS,
2838:             "start_time": 0.0,
2839:             "end_time": 1.0,
2840:             "execution_time_ms": 1000.0,
2841:             "successful_items": 2,
2842:             "failed_items": 1,
2843:             "retries_used": 0,
2844:             "error_messages": ["error"],
2845:         },
2846:     )
2847:     failed = result.get_failed_items()
2848:     assert len(failed) == 1
2849:     assert failed[0] == 2
2850:
2851:
2852: class TestEdgeCases:
2853:     """Test edge cases and error conditions."""
2854:
2855:     @pytest.mark.asyncio
2856:     async def test_execute_batches_single_item(self):
```

```
2857:         self,
2858:         batch_config,
2859:         mock_method_executor,
2860:         sample_document,
2861:         sample_question_context,
2862:     ):
2863:         """Test batch execution with single item."""
2864:         executor = BatchExecutor(
2865:             config=batch_config, method_executor=mock_method_executor
2866:         )
2867:
2868:         items = [1]
2869:         batch_results = []
2870:
2871:         async for batch_result in executor.execute_batches(
2872:             items, MockExecutor, sample_document, sample_question_context, batch_size=10
2873:         ):
2874:             batch_results.append(batch_result)
2875:
2876:         assert len(batch_results) == 1
2877:         assert batch_results[0].metrics["batch_size"] == 1
2878:
2879:     @pytest.mark.asyncio
2880:     async def test_batch_counter_increments(
2881:         self,
2882:         batch_config,
2883:         mock_method_executor,
2884:         sample_document,
2885:         sample_question_context,
2886:     ):
2887:         """Test that batch counter increments properly."""
2888:         executor = BatchExecutor(
2889:             config=batch_config, method_executor=mock_method_executor
2890:         )
2891:
2892:         items = list(range(15))
2893:         batch_results = []
2894:
2895:         async for batch_result in executor.execute_batches(
2896:             items, MockExecutor, sample_document, sample_question_context, batch_size=5
2897:         ):
2898:             batch_results.append(batch_result)
2899:
2900:         assert executor._batch_counter == 3
2901:         assert batch_results[0].batch_id == "batch_000001"
2902:         assert batch_results[1].batch_id == "batch_000002"
2903:         assert batch_results[2].batch_id == "batch_000003"
2904:
2905:     def test_aggregated_results_success_rate_zero_items(self):
2906:         """Test success rate calculation with zero items."""
2907:         aggregated = AggregatedBatchResults(
2908:             total_batches=0,
2909:             total_items=0,
2910:             successful_items=0,
2911:             failed_items=0,
2912:             results=[],
```

```
2913:         errors=[],
2914:         execution_time_ms=0.0,
2915:         batch_metrics=[],
2916:     )
2917:     assert aggregated.success_rate() == 0.0
2918:
2919:
2920:
2921: =====
2922: FILE: tests/test_chunk_semantic_auditor.py
2923: =====
2924:
2925: import json
2926: import tempfile
2927: from pathlib import Path
2928:
2929: import pytest
2930:
2931: from tools.chunk_semantic_auditor import (
2932:     ChunkMetadata,
2933:     ChunkSemanticAuditor,
2934:     SemanticAuditResult,
2935: )
2936:
2937:
2938: @pytest.fixture
2939: def temp_artifacts_dir():
2940:     with tempfile.TemporaryDirectory() as tmpdir:
2941:         yield Path(tmpdir)
2942:
2943:
2944: @pytest.fixture
2945: def sample_chunks_file(temp_artifacts_dir):
2946:     chunks_data = {
2947:         "chunks": [
2948:             {
2949:                 "chunk_id": "chunk_001",
2950:                 "policy_area_id": "PA01",
2951:                 "dimension_id": "DIM01",
2952:                 "text": "The budget allocation for economic development includes funding for small business support programs, entrepreneurship training, and innovation infrastructure with a total investment of $5 million.",
2953:             },
2954:             {
2955:                 "chunk_id": "chunk_002",
2956:                 "policy_area_id": "PA02",
2957:                 "dimension_id": "DIM03",
2958:                 "text": "The health program will deliver 50 community clinics, vaccinate 100,000 children, and provide free medical consultations to vulnerable populations.",
2959:             },
2960:             {
2961:                 "chunk_id": "chunk_003",
2962:                 "policy_area_id": "PA04",
2963:                 "dimension_id": "DIM06",
2964:                 "text": "The causal link between reforestation activities and improved air quality will be measured through PM2.5 monitoring stations and longitudinal health studies.",
2965:             },

```

```
2966:         ]
2967:     }
2968:
2969:     chunks_file = temp_artifacts_dir / "test_chunks.json"
2970:     with open(chunks_file, "w") as f:
2971:         json.dump(chunks_data, f)
2972:
2973:     return chunks_file
2974:
2975:
2976: @pytest.fixture
2977: def mismatched_chunks_file(temp_artifacts_dir):
2978:     chunks_data = {
2979:         "chunks": [
2980:             {
2981:                 "chunk_id": "chunk_mismatch",
2982:                 "policy_area_id": "PA01",
2983:                 "dimension_id": "DIM01",
2984:                 "text": "This text discusses cultural heritage preservation and traditional music festivals, which has nothing to do with economic development or inputs.",
2985:             }
2986:         ]
2987:     }
2988:
2989:     chunks_file = temp_artifacts_dir / "mismatched_chunks.json"
2990:     with open(chunks_file, "w") as f:
2991:         json.dump(chunks_data, f)
2992:
2993:     return chunks_file
2994:
2995:
2996: def test_chunk_metadata_creation():
2997:     chunk = ChunkMetadata(
2998:         chunk_id="test_001",
2999:         file_path="test.json",
3000:         policy_area_id="PA01",
3001:         dimension_id="DIM01",
3002:         text_content="Sample text",
3003:     )
3004:
3005:     assert chunk.chunk_id == "test_001"
3006:     assert chunk.policy_area_id == "PA01"
3007:     assert chunk.dimension_id == "DIM01"
3008:
3009:
3010: def test_auditor_initialization(temp_artifacts_dir):
3011:     auditor = ChunkSemanticAuditor(
3012:         artifacts_dir=temp_artifacts_dir, threshold=0.7, verbose=False
3013:     )
3014:
3015:     assert auditor.artifacts_dir == temp_artifacts_dir
3016:     assert auditor.threshold == 0.7
3017:     assert auditor.model is None
3018:     assert len(auditor.chunks) == 0
3019:
3020:
```

```
3021: def test_load_model(temp_artifacts_dir):
3022:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3023:     auditor.load_model()
3024:
3025:     assert auditor.model is not None
3026:
3027:
3028: def test_discover_chunk_artifacts(sample_chunks_file, temp_artifacts_dir):
3029:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3030:     chunk_files = auditor.discover_chunk_artifacts()
3031:
3032:     assert len(chunk_files) >= 1
3033:     assert sample_chunks_file in chunk_files
3034:
3035:
3036: def test_load_chunk_metadata(sample_chunks_file, temp_artifacts_dir):
3037:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3038:     chunks = auditor.load_chunk_metadata(sample_chunks_file)
3039:
3040:     assert len(chunks) == 3
3041:     assert chunks[0].chunk_id == "chunk_001"
3042:     assert chunks[0].policy_area_id == "PA01"
3043:     assert chunks[0].dimension_id == "DIM01"
3044:     assert "economic development" in chunks[0].text_content.lower()
3045:
3046:
3047: def test_load_all_chunks(sample_chunks_file, temp_artifacts_dir):
3048:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3049:     auditor.load_all_chunks()
3050:
3051:     assert len(auditor.chunks) == 3
3052:
3053:
3054: def test_compute_semantic_coherence(sample_chunks_file, temp_artifacts_dir):
3055:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3056:     auditor.load_model()
3057:     auditor.load_all_chunks()
3058:
3059:     chunk = auditor.chunks[0]
3060:     score = auditor.compute_semantic_coherence(chunk)
3061:
3062:     assert isinstance(score, float)
3063:     assert 0.0 <= score <= 1.0
3064:
3065:
3066: def test_audit_chunk_pass(sample_chunks_file, temp_artifacts_dir):
3067:     auditor = ChunkSemanticAuditor(
3068:         artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
3069:     )
3070:     auditor.load_model()
3071:     auditor.load_all_chunks()
3072:
3073:     chunk = auditor.chunks[0]
3074:     result = auditor.audit_chunk(chunk)
3075:
3076:     assert isinstance(result, SemanticAuditResult)
```

```
3077:     assert result.chunk_id == chunk.chunk_id
3078:     assert result.passed is True
3079:
3080:
3081: def test_audit_chunk_fail(mismatched_chunks_file, temp_artifacts_dir):
3082:     auditor = ChunkSemanticAuditor(
3083:         artifacts_dir=temp_artifacts_dir, threshold=0.7, verbose=False
3084:     )
3085:     auditor.load_model()
3086:     auditor.load_all_chunks()
3087:
3088:     chunk = auditor.chunks[0]
3089:     result = auditor.audit_chunk(chunk)
3090:
3091:     assert isinstance(result, SemanticAuditResult)
3092:     assert result.passed is False
3093:     assert result.coherence_score < 0.7
3094:
3095:
3096: def test_audit_all_chunks(sample_chunks_file, temp_artifacts_dir):
3097:     auditor = ChunkSemanticAuditor(
3098:         artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
3099:     )
3100:     auditor.load_model()
3101:     auditor.load_all_chunks()
3102:     auditor.audit_all_chunks()
3103:
3104:     assert len(auditor.audit_results) == 3
3105:     assert all(isinstance(r, SemanticAuditResult) for r in auditor.audit_results)
3106:
3107:
3108: def test_generate_report(sample_chunks_file, temp_artifacts_dir):
3109:     auditor = ChunkSemanticAuditor(
3110:         artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
3111:     )
3112:     auditor.load_model()
3113:     auditor.load_all_chunks()
3114:     auditor.audit_all_chunks()
3115:
3116:     report = auditor.generate_report()
3117:
3118:     assert "metadata" in report
3119:     assert "summary" in report
3120:     assert "failures" in report
3121:     assert report["metadata"]["total_chunks_audited"] == 3
3122:     assert report["summary"]["passed"] >= 0
3123:     assert report["summary"]["failed"] >= 0
3124:     assert report["summary"]["pass_rate"] >= 0.0
3125:
3126:
3127: def test_save_report(sample_chunks_file, temp_artifacts_dir):
3128:     auditor = ChunkSemanticAuditor(
3129:         artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
3130:     )
3131:     auditor.load_model()
3132:     auditor.load_all_chunks()
```

```
3133:     auditor.audit_all_chunks()
3134:
3135:     report = auditor.generate_report()
3136:     output_file = temp_artifacts_dir / "test_report.json"
3137:     auditor.save_report(report, output_file)
3138:
3139:     assert output_file.exists()
3140:
3141:     with open(output_file) as f:
3142:         loaded_report = json.load(f)
3143:
3144:     assert loaded_report == report
3145:
3146:
3147: def test_full_audit_workflow(sample_chunks_file, temp_artifacts_dir):
3148:     auditor = ChunkSemanticAuditor(
3149:         artifacts_dir=temp_artifacts_dir, threshold=0.3, verbose=False
3150:     )
3151:
3152:     exit_code = auditor.run()
3153:
3154:     assert exit_code in [0, 1]
3155:     assert (temp_artifacts_dir / "semantic_audit_report.json").exists()
3156:
3157:
3158: def test_empty_artifacts_dir():
3159:     with tempfile.TemporaryDirectory() as tmpdir:
3160:         auditor = ChunkSemanticAuditor(artifacts_dir=Path(tmpdir), verbose=False)
3161:         auditor.load_model()
3162:         auditor.load_all_chunks()
3163:
3164:         assert len(auditor.chunks) == 0
3165:
3166:
3167: def test_invalid_json_file(temp_artifacts_dir):
3168:     invalid_file = temp_artifacts_dir / "invalid_chunks.json"
3169:     with open(invalid_file, "w") as f:
3170:         f.write("{ invalid json }")
3171:
3172:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3173:     chunks = auditor.load_chunk_metadata(invalid_file)
3174:
3175:     assert len(chunks) == 0
3176:
3177:
3178: def test_missing_metadata_fields(temp_artifacts_dir):
3179:     incomplete_chunks = {
3180:         "chunks": [
3181:             {"chunk_id": "c1", "text": "text only"},
3182:             {"policy_area_id": "PA01", "text": "missing dimension"},
3183:             {"dimension_id": "DIM01", "text": "missing policy area"},
3184:         ]
3185:     }
3186:
3187:     chunks_file = temp_artifacts_dir / "incomplete.json"
3188:     with open(chunks_file, "w") as f:
```

```
3189:         json.dump(incomplete_chunks, f)
3190:
3191:     auditor = ChunkSemanticAuditor(artifacts_dir=temp_artifacts_dir, verbose=False)
3192:     chunks = auditor.load_chunk_metadata(chunks_file)
3193:
3194:     assert len(chunks) == 0
3195:
3196:
3197:
3198: =====
3199: FILE: tests/test_config_manager.py
3200: =====
3201:
3202: """Tests for configuration management system."""
3203:
3204: import json
3205: import shutil
3206: import tempfile
3207: import time
3208: from datetime import datetime
3209: from pathlib import Path
3210:
3211: import pytest
3212:
3213: from system.config.config_manager import ConfigManager
3214:
3215:
3216: @pytest.fixture
3217: def temp_config_dir():
3218:     """Create temporary config directory for testing."""
3219:     temp_dir = tempfile.mkdtemp()
3220:     yield Path(temp_dir)
3221:     shutil.rmtree(temp_dir)
3222:
3223:
3224: @pytest.fixture
3225: def config_manager(temp_config_dir):
3226:     """Create ConfigManager instance for testing."""
3227:     return ConfigManager(config_root=temp_config_dir)
3228:
3229:
3230: class TestConfigManager:
3231:     """Test suite for ConfigManager."""
3232:
3233:     def test_init_creates_directories(self, temp_config_dir):
3234:         """Test that initialization creates required directories."""
3235:         ConfigManager(config_root=temp_config_dir)
3236:
3237:         assert (temp_config_dir / ".backup").exists()
3238:         assert (temp_config_dir / "calibration").exists()
3239:         assert (temp_config_dir / "questionnaire").exists()
3240:         assert (temp_config_dir / "environments").exists()
3241:
3242:     def test_save_configCreates_file(self, config_manager, temp_config_dir):
3243:         """Test saving configuration file."""
3244:         content = "test content"
```

```
3245:     file_path = config_manager.save_config("test/config.txt", content)
3246:
3247:     assert file_path.exists()
3248:     assert file_path.read_text() == content
3249:
3250: def test_save_config_json(self, config_manager, temp_config_dir):
3251:     """Test saving JSON configuration."""
3252:     data = {"key": "value", "number": 42, "nested": {"a": 1}}
3253:     file_path = config_manager.save_config_json("test/config.json", data)
3254:
3255:     assert file_path.exists()
3256:     loaded_data = json.loads(file_path.read_text())
3257:     assert loaded_data == data
3258:
3259: def test_save_updates_registry(self, config_manager, temp_config_dir):
3260:     """Test that saving a file updates the hash registry."""
3261:     config_manager.save_config("test/file.txt", "content")
3262:
3263:     registry = config_manager.get_registry()
3264:     assert "test/file.txt" in registry
3265:     assert "hash" in registry["test/file.txt"]
3266:     assert "last_modified" in registry["test/file.txt"]
3267:     assert "size_bytes" in registry["test/file.txt"]
3268:
3269: def test_hash_computation_deterministic(self, config_manager, temp_config_dir):
3270:     """Test that SHA256 hash computation is deterministic."""
3271:     content = "test content for hashing"
3272:     config_manager.save_config("test/hash.txt", content)
3273:
3274:     info1 = config_manager.get_file_info("test/hash.txt")
3275:     config_manager.save_config("test/hash.txt", content, create_backup=False)
3276:     info2 = config_manager.get_file_info("test/hash.txt")
3277:
3278:     assert info1["hash"] == info2["hash"]
3279:
3280: def test_hash_changes_on_modification(self, config_manager, temp_config_dir):
3281:     """Test that hash changes when file is modified."""
3282:     config_manager.save_config("test/file.txt", "original content")
3283:     original_hash = config_manager.get_file_info("test/file.txt")["hash"]
3284:
3285:     config_manager.save_config("test/file.txt", "modified content")
3286:     modified_hash = config_manager.get_file_info("test/file.txt")["hash"]
3287:
3288:     assert original_hash != modified_hash
3289:
3290: def test_backup_created_on_save(self, config_manager, temp_config_dir):
3291:     """Test that backup is created before modification."""
3292:     config_manager.save_config("test/file.txt", "original")
3293:     time.sleep(0.001)
3294:     config_manager.save_config("test/file.txt", "modified")
3295:
3296:     backups = config_manager.list_backups("test/file.txt")
3297:     assert len(backups) >= 1
3298:
3299:     backup_content = backups[0].read_text()
3300:     assert backup_content == "original"
```

```
3301:
3302:     def test_backup_filename_format(self, config_manager, temp_config_dir):
3303:         """Test that backup filenames follow YYYYMMDD_HHMMSS_microseconds format."""
3304:         config_manager.save_config("calibration/model.json", {'test': true})
3305:         time.sleep(0.001)
3306:         config_manager.save_config("calibration/model.json", {'test': false})
3307:
3308:         backups = config_manager.list_backups("calibration/model.json")
3309:         assert len(backups) >= 1
3310:
3311:         backup_name = backups[0].name
3312:         parts = backup_name.split("_")
3313:         timestamp_part = parts[0] + "_" + parts[1]
3314:
3315:         try:
3316:             datetime.strptime(timestamp_part, "%Y%m%d_%H%M%S")
3317:         except ValueError:
3318:             pytest.fail(
3319:                 f"Backup filename doesn't match YYYYMMDD_HHMMSS_microseconds format: {backup_name}"
3320:             )
3321:
3322:     def test_no_backup_on_first_save(self, config_manager, temp_config_dir):
3323:         """Test that no backup is created for new files."""
3324:         config_manager.save_config("test/new.txt", "first save")
3325:
3326:         backups = config_manager.list_backups("test/new.txt")
3327:         assert len(backups) == 0
3328:
3329:     def test_load_config(self, config_manager, temp_config_dir):
3330:         """Test loading configuration file."""
3331:         content = "test content"
3332:         config_manager.save_config("test/file.txt", content)
3333:
3334:         loaded_content = config_manager.load_config("test/file.txt")
3335:         assert loaded_content == content
3336:
3337:     def test_load_config_json(self, config_manager, temp_config_dir):
3338:         """Test loading JSON configuration."""
3339:         data = {"key": "value", "list": [1, 2, 3]}
3340:         config_manager.save_config_json("test/data.json", data)
3341:
3342:         loaded_data = config_manager.load_config_json("test/data.json")
3343:         assert loaded_data == data
3344:
3345:     def test_verify_hash_valid(self, config_manager, temp_config_dir):
3346:         """Test hash verification for unchanged file."""
3347:         config_manager.save_config("test/file.txt", "content")
3348:         assert config_manager.verify_hash("test/file.txt") is True
3349:
3350:     def test_verify_hash_invalid_after_external_modification(
3351:         self, config_manager, temp_config_dir
3352:     ):
3353:         """Test hash verification fails after external modification."""
3354:         file_path = config_manager.save_config("test/file.txt", "original")
3355:
3356:         file_path.write_text("externally modified")
```

```
3357:  
3358:     assert config_manager.verify_hash("test/file.txt") is False  
3359:  
3360: def test_verify_hash_nonexistent_file(self, config_manager, temp_config_dir):  
3361:     """Test hash verification for nonexistent file."""  
3362:     assert config_manager.verify_hash("nonexistent.txt") is False  
3363:  
3364: def test_list_backups_sorted_by_timestamp(self, config_manager, temp_config_dir):  
3365:     """Test that backups are sorted with newest first."""  
3366:     config_manager.save_config("test/file.txt", "version 1")  
3367:     time.sleep(0.001)  
3368:     config_manager.save_config("test/file.txt", "version 2")  
3369:     time.sleep(0.001)  
3370:     config_manager.save_config("test/file.txt", "version 3")  
3371:  
3372:     backups = config_manager.list_backups("test/file.txt")  
3373:     assert len(backups) >= 2  
3374:  
3375:     for i in range(len(backups) - 1):  
3376:         assert backups[i].name > backups[i + 1].name  
3377:  
3378: def test_list_backups_all(self, config_manager, temp_config_dir):  
3379:     """Test listing all backups without filter."""  
3380:     config_manager.save_config("file1.txt", "content")  
3381:     config_manager.save_config("file1.txt", "modified")  
3382:  
3383:     config_manager.save_config("file2.txt", "content")  
3384:     config_manager.save_config("file2.txt", "modified")  
3385:  
3386:     all_backups = config_manager.list_backups()  
3387:     assert len(all_backups) >= 2  
3388:  
3389: def test_restore_backup(self, config_manager, temp_config_dir):  
3390:     """Test restoring from backup."""  
3391:     original_content = "original version"  
3392:     config_manager.save_config("test/file.txt", original_content)  
3393:     time.sleep(0.001)  
3394:  
3395:     config_manager.save_config("test/file.txt", "modified version")  
3396:  
3397:     backups = config_manager.list_backups("test/file.txt")  
3398:     config_manager.restore_backup(backups[0])  
3399:  
3400:     restored_content = config_manager.load_config("test/file.txt")  
3401:     assert restored_content == original_content  
3402:  
3403: def test_restoreCreatesBackup(self, config_manager, temp_config_dir):  
3404:     """Test that restore creates backup of current version."""  
3405:     config_manager.save_config("test/file.txt", "version 1")  
3406:     time.sleep(0.001)  
3407:     config_manager.save_config("test/file.txt", "version 2")  
3408:  
3409:     backups_before = len(config_manager.list_backups("test/file.txt"))  
3410:  
3411:     first_backup = config_manager.list_backups("test/file.txt")[-1]  
3412:     config_manager.restore_backup(first_backup)
```

```
3413:  
3414:     backups_after = len(config_manager.list_backups("test/file.txt"))  
3415:     assert backups_after == backups_before + 1  
3416:  
3417:     def test_rebuild_registry(self, config_manager, temp_config_dir):  
3418:         """Test rebuilding hash registry from disk."""  
3419:         config_manager.save_config("file1.json", '{"a": 1}')  
3420:         config_manager.save_config("file2.yaml", "key: value")  
3421:  
3422:         registry_file = temp_config_dir / "config_hash_registry.json"  
3423:         registry_file.unlink()  
3424:  
3425:         rebuilt_registry = config_manager.rebuild_registry()  
3426:  
3427:         assert "file1.json" in rebuilt_registry  
3428:         assert "file2.yaml" in rebuilt_registry  
3429:  
3430:     def test_get_file_info_nonexistent(self, config_manager, temp_config_dir):  
3431:         """Test getting info for nonexistent file."""  
3432:         info = config_manager.get_file_info("nonexistent.txt")  
3433:         assert info is None  
3434:  
3435:     def test_registry_persistence(self, temp_config_dir):  
3436:         """Test that registry persists across manager instances."""  
3437:         manager1 = ConfigManager(config_root=temp_config_dir)  
3438:         manager1.save_config("test/file.txt", "content")  
3439:         original_hash = manager1.get_file_info("test/file.txt")["hash"]  
3440:  
3441:         manager2 = ConfigManager(config_root=temp_config_dir)  
3442:         loaded_hash = manager2.get_file_info("test/file.txt")["hash"]  
3443:  
3444:         assert original_hash == loaded_hash  
3445:  
3446:     def test_save_without_backup(self, config_manager, temp_config_dir):  
3447:         """Test saving without creating backup."""  
3448:         config_manager.save_config("test/file.txt", "original")  
3449:         config_manager.save_config("test/file.txt", "modified", create_backup=False)  
3450:  
3451:         backups = config_manager.list_backups("test/file.txt")  
3452:         assert len(backups) == 0  
3453:  
3454:     def test_nested_directory_creation(self, config_manager, temp_config_dir):  
3455:         """Test that nested directories are created automatically."""  
3456:         config_manager.save_config("a/b/c/deep.json", {"nested": true})  
3457:  
3458:         file_path = temp_config_dir / "a/b/c/deep.json"  
3459:         assert file_path.exists()  
3460:  
3461:     def test_json_sorted_keys(self, config_manager, temp_config_dir):  
3462:         """Test that JSON is saved with sorted keys."""  
3463:         data = {"z": 1, "a": 2, "m": 3}  
3464:         file_path = config_manager.save_config_json("test/sorted.json", data)  
3465:  
3466:         content = file_path.read_text()  
3467:         assert content.index('a') < content.index('m') < content.index('z')  
3468:
```

```
3469:     def test_unicode_content_handling(self, config_manager, temp_config_dir):
3470:         """Test handling of Unicode content."""
3471:         content = "Hello ä,\u2264\225\214 ö\237\214\215"
3472:         config_manager.save_config("test/unicode.txt", content)
3473:
3474:         loaded = config_manager.load_config("test/unicode.txt")
3475:         assert loaded == content
3476:
3477:
3478:
3479: =====
3480: FILE: tests/test_intrinsic_pipeline_behavior.py
3481: =====
3482:
3483: """
3484: test_intrinsic_pipeline_behavior.py - Test pipeline execution with intrinsic calibration
3485:
3486: Tests fallback behavior:
3487: 1. All computed: Normal execution with actual @b values
3488: 2. Mix of pending/excluded: Verify fallback behavior (pending\206\222@b=0.5, excluded\206\222skip, none\206\222@b=0.3 with warning)
3489:
3490: DEPRECATED: This test relies on outdated IntrinsicCalibrationLoader interface.
3491: See tests/DEPRECATED_TESTS.md for details.
3492: """
3493: import json
3494: import tempfile
3495: from pathlib import Path
3496:
3497: import pytest
3498:
3499: from src.farfan_pipeline.core.calibration.intrinsic_calibration_loader import (
3500:     IntrinsicCalibrationLoader,
3501: )
3502:
3503: pytestmark = pytest.mark.obsolete
3504:
3505:
3506: @pytest.fixture
3507: def temp_calibration_file():
3508:     """Create a temporary calibration file for testing."""
3509:     data = {
3510:         "_metadata": {
3511:             "version": "1.0.0",
3512:             "generated": "2025-01-15T00:00:00Z",
3513:             "description": "Test calibration data",
3514:             "total_methods": 5,
3515:             "computed_methods": 4,
3516:             "coverage_percent": 80.0
3517:         },
3518:         "TestClass.computed_method": {
3519:             "intrinsic_score": [0.75, 0.85],
3520:             "b_theory": 0.80,
3521:             "b_impl": 0.78,
3522:             "b_deploy": 0.83,
3523:             "calibration_status": "computed",
3524:             "layer": "engine",
```

```
3525:         "last_updated": "2025-01-15T00:00:00Z"
3526:     },
3527:     "TestClass.pending_method": {
3528:         "intrinsic_score": [0.0, 0.0],
3529:         "b_theory": 0.0,
3530:         "b_impl": 0.0,
3531:         "b_deploy": 0.0,
3532:         "calibration_status": "pending",
3533:         "layer": "processor",
3534:         "last_updated": "2025-01-15T00:00:00Z"
3535:     },
3536:     "TestClass.excluded_method": {
3537:         "intrinsic_score": [0.0, 0.0],
3538:         "b_theory": 0.0,
3539:         "b_impl": 0.0,
3540:         "b_deploy": 0.0,
3541:         "calibration_status": "excluded",
3542:         "layer": "utility",
3543:         "last_updated": "2025-01-15T00:00:00Z"
3544:     },
3545:     "TestClass.none_method": {
3546:         "intrinsic_score": [0.0, 0.0],
3547:         "b_theory": 0.0,
3548:         "b_impl": 0.0,
3549:         "b_deploy": 0.0,
3550:         "calibration_status": "none",
3551:         "layer": "engine",
3552:         "last_updated": "2025-01-15T00:00:00Z"
3553:     },
3554:     "TestClass.another_computed": {
3555:         "intrinsic_score": [0.68, 0.78],
3556:         "b_theory": 0.73,
3557:         "b_impl": 0.70,
3558:         "b_deploy": 0.76,
3559:         "calibration_status": "computed",
3560:         "layer": "processor",
3561:         "last_updated": "2025-01-15T00:00:00Z"
3562:     }
3563: }
3564:
3565: with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.json') as f:
3566:     json.dump(data, f, indent=2)
3567:     temp_path = f.name
3568:
3569:     yield temp_path
3570:     Path(temp_path).unlink()
3571:
3572:
3573: def test_loader_initialization(temp_calibration_file):
3574:     """Test loader initializes correctly with valid data."""
3575:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3576:     metadata = loader.get_metadata()
3577:
3578:     assert metadata["coverage_percent"] >= 80.0
3579:     assert metadata["computed_methods"] == 4
3580:
```

```
3581:
3582: def test_loader_rejects_low_coverage():
3583:     """Test loader rejects data with <80% coverage."""
3584:     data = {
3585:         "_metadata": {
3586:             "version": "1.0.0",
3587:             "generated": "2025-01-15T00:00:00Z",
3588:             "description": "Low coverage test",
3589:             "total_methods": 100,
3590:             "computed_methods": 70,
3591:             "coverage_percent": 70.0
3592:         }
3593:     }
3594:
3595:     with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.json') as f:
3596:         json.dump(data, f, indent=2)
3597:         temp_path = f.name
3598:
3599:     try:
3600:         with pytest.raises(ValueError, match="coverage.*80"):
3601:             IntrinsicCalibrationLoader(temp_path)
3602:     finally:
3603:         Path(temp_path).unlink()
3604:
3605:
3606: def test_computed_method_returns_actual_values(temp_calibration_file):
3607:     """Test computed method returns actual @b values from JSON."""
3608:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3609:     cal = loader.get_calibration("TestClass.computed_method")
3610:
3611:     assert cal is not None
3612:     assert cal.calibration_status == "computed"
3613:     assert cal.b_theory == 0.80
3614:     assert cal.b_impl == 0.78
3615:     assert cal.b_deploy == 0.83
3616:     assert cal.intrinsic_score == (0.75, 0.85)
3617:     assert cal.layer == "engine"
3618:
3619:
3620: def test_pending_method_returns_fallback_05(temp_calibration_file):
3621:     """Test pending method returns @b=0.5 fallback."""
3622:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3623:     cal = loader.get_calibration("TestClass.pending_method")
3624:
3625:     assert cal is not None
3626:     assert cal.calibration_status == "pending"
3627:     assert cal.b_theory == 0.5
3628:     assert cal.b_impl == 0.5
3629:     assert cal.b_deploy == 0.5
3630:     assert cal.intrinsic_score == (0.48, 0.52)
3631:
3632:
3633: def test_excluded_method_returns_none(temp_calibration_file):
3634:     """Test excluded method returns None (signals skip)."""
3635:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3636:     cal = loader.get_calibration("TestClass.excluded_method")
```

```
3637:  
3638:     assert cal is None  
3639:  
3640:  
3641: def test_none_method_returns_fallback_03(temp_calibration_file):  
3642:     """Test none status method returns @b=0.3 fallback."""  
3643:     loader = IntrinsicCalibrationLoader(temp_calibration_file)  
3644:     cal = loader.get_calibration("TestClass.none_method")  
3645:  
3646:     assert cal is not None  
3647:     assert cal.calibration_status == "none"  
3648:     assert cal.b_theory == 0.3  
3649:     assert cal.b_impl == 0.3  
3650:     assert cal.b_deploy == 0.3  
3651:     assert cal.intrinsic_score == (0.28, 0.32)  
3652:  
3653:  
3654: def test_missing_method_returns_fallback_03(temp_calibration_file):  
3655:     """Test method not in registry returns @b=0.3 fallback."""  
3656:     loader = IntrinsicCalibrationLoader(temp_calibration_file)  
3657:     cal = loader.get_calibration("TestClass.unknown_method")  
3658:  
3659:     assert cal is not None  
3660:     assert cal.calibration_status == "none"  
3661:     assert cal.b_theory == 0.3  
3662:     assert cal.b_impl == 0.3  
3663:     assert cal.b_deploy == 0.3  
3664:  
3665:  
3666: def test_contamination_detection():  
3667:     """Test loader detects contaminated data."""  
3668:     data = {  
3669:         "_metadata": {  
3670:             "version": "1.0.0",  
3671:             "generated": "2025-01-15T00:00:00Z",  
3672:             "description": "Contaminated test",  
3673:             "total_methods": 1,  
3674:             "computed_methods": 1,  
3675:             "coverage_percent": 100.0  
3676:         },  
3677:         "TestClass.contaminated_method": {  
3678:             "intrinsic_score": [0.75, 0.85],  
3679:             "b_theory": 0.80,  
3680:             "b_impl": 0.78,  
3681:             "b_deploy": 0.83,  
3682:             "final_score": 0.85, # CONTAMINATION  
3683:             "calibration_status": "computed",  
3684:             "layer": "engine",  
3685:             "last_updated": "2025-01-15T00:00:00Z"  
3686:         }  
3687:     }  
3688:  
3689:     with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.json') as f:  
3690:         json.dump(data, f, indent=2)  
3691:         temp_path = f.name  
3692:
```

```
3693:     try:
3694:         loader = IntrinsicCalibrationLoader(temp_path)
3695:         with pytest.raises(ValueError, match="CONTAMINATION"):
3696:             loader.verify_purity()
3697:     finally:
3698:         Path(temp_path).unlink()
3699:
3700:
3701: def test_composite_b_calculation(temp_calibration_file):
3702:     """Test composite @b score calculation."""
3703:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3704:     cal = loader.get_calibration("TestClass.computed_method")
3705:
3706:     expected_composite = (0.80 + 0.78 + 0.83) / 3.0
3707:     assert abs(cal.get_composite_b() - expected_composite) < 0.001
3708:
3709:
3710: def test_pipeline_execution_all_computed(temp_calibration_file):
3711:     """Simulate pipeline execution with all computed methods."""
3712:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3713:
3714:     methods_to_execute = [
3715:         "TestClass.computed_method",
3716:         "TestClass.another_computed"
3717:     ]
3718:
3719:     results = {}
3720:     for method_id in methods_to_execute:
3721:         cal = loader.get_calibration(method_id)
3722:         if cal is not None: # Not excluded
3723:             results[method_id] = {
3724:                 "executed": True,
3725:                 "composite_b": cal.get_composite_b(),
3726:                 "status": cal.calibration_status
3727:             }
3728:
3729:     assert len(results) == 2
3730:     assert all(r["executed"] for r in results.values())
3731:     assert all(r["status"] == "computed" for r in results.values())
3732:
3733:
3734: def test_pipeline_execution_mixed_statuses(temp_calibration_file):
3735:     """Simulate pipeline execution with mixed calibration statuses."""
3736:     loader = IntrinsicCalibrationLoader(temp_calibration_file)
3737:
3738:     methods_to_execute = [
3739:         "TestClass.computed_method",
3740:         "TestClass.pending_method",
3741:         "TestClass.excluded_method",
3742:         "TestClass.none_method"
3743:     ]
3744:
3745:     results = {}
3746:     skipped = []
3747:
3748:     for method_id in methods_to_execute:
```

```
3749:     cal = loader.get_calibration(method_id)
3750:     if cal is None: # Excluded
3751:         skipped.append(method_id)
3752:     else:
3753:         results[method_id] = {
3754:             "executed": True,
3755:             "composite_b": cal.get_composite_b(),
3756:             "status": cal.calibration_status,
3757:             "is_fallback": cal.calibration_status != "computed"
3758:         }
3759:
3760:     assert len(results) == 3 # computed, pending, none
3761:     assert len(skipped) == 1 # excluded
3762:     assert "TestClass.excluded_method" in skipped
3763:
3764:     # Verify fallback behavior
3765:     assert results["TestClass.pending_method"]["composite_b"] == 0.5
3766:     assert results["TestClass.none_method"]["composite_b"] == 0.3
3767:     assert results["TestClass.computed_method"]["composite_b"] > 0.7
3768:
3769:
3770: def test_real_calibration_file_exists():
3771:     """Test that the real calibration file exists and is valid."""
3772:     cal_path = Path("config/intrinsic_calibration.json")
3773:     assert cal_path.exists(), "Real calibration file must exist"
3774:
3775:     loader = IntrinsicCalibrationLoader(str(cal_path))
3776:     metadata = loader.get_metadata()
3777:
3778:     assert metadata["coverage_percent"] >= 80.0
3779:     assert metadata["computed_methods"] >= 30
3780:
3781:     # Verify purity
3782:     assert loader.verify_purity()
3783:
3784:
3785: if __name__ == "__main__":
3786:     pytest.main([__file__, "-v"])
3787:
3788:
3789:
3790: =====
3791: FILE: tests/test_intrinsic_purity.py
3792: =====
3793:
3794: """
3795: test_intrinsic_purity.py - Verification of @b-only enforcement in intrinsic calibration
3796:
3797: This test ensures strict @b-only enforcement:
3798: - NO keys matching @chain, @q, @d, @p, @C, @u, @m, final_score, layer_scores
3799: - ONLY keys: intrinsic_score, b_theory, b_impl, b_deploy, calibration_status, layer, last_updated
3800: - Coverage >= 80% (computed methods / total methods)
3801: - At least 30 executors with status='computed'
3802: """
3803: import json
3804: from pathlib import Path
```

```
3805:  
3806: import pytest  
3807:  
3808:  
3809: def test_intrinsic_calibration_exists():  
3810:     """Verify intrinsic_calibration.json exists."""  
3811:     cal_path = Path("config/intrinsic_calibration.json")  
3812:     assert cal_path.exists(), "config/intrinsic_calibration.json must exist"  
3813:  
3814:  
3815: def test_no_contaminated_keys():  
3816:     """CRITICAL: Fail if ANY non-@b layer data exists in JSON."""  
3817:     cal_path = Path("config/intrinsic_calibration.json")  
3818:     with open(cal_path) as f:  
3819:         data = json.load(f)  
3820:  
3821:     forbidden_patterns = ["@chain", "@q", "@d", "@p", "@C", "@u", "@m",  
3822:                            "final_score", "layer_scores", "chain_", "queue_",  
3823:                            "downstream_", "upstream_", "composite_"]  
3824:  
3825:     for method_id, method_data in data.items():  
3826:         if method_id == "_metadata":  
3827:             continue  
3828:  
3829:         for key in method_data:  
3830:             for pattern in forbidden_patterns:  
3831:                 assert pattern not in key.lower(), (  
3832:                     f"CONTAMINATION DETECTED: method '{method_id}' contains "  
3833:                     f"forbidden key '{key}' matching pattern '{pattern}'. "  
3834:                     f"Intrinsic calibration incomplete or contaminated."  
3835:                 )  
3836:  
3837:  
3838: def test_only_b_layer_keys():  
3839:     """Verify each method entry contains ONLY @b-layer keys."""  
3840:     cal_path = Path("config/intrinsic_calibration.json")  
3841:     with open(cal_path) as f:  
3842:         data = json.load(f)  
3843:  
3844:     allowed_keys = {"intrinsic_score", "b_theory", "b_impl", "b_deploy",  
3845:                      "calibration_status", "layer", "last_updated"}  
3846:  
3847:     for method_id, method_data in data.items():  
3848:         if method_id == "_metadata":  
3849:             continue  
3850:  
3851:         actual_keys = set(method_data.keys())  
3852:         extra_keys = actual_keys - allowed_keys  
3853:         missing_keys = allowed_keys - actual_keys  
3854:  
3855:         assert not extra_keys, (  
3856:             f"Method '{method_id}' has unauthorized keys: {extra_keys}. "  
3857:             f"Only {allowed_keys} are allowed."  
3858:         )  
3859:         assert not missing_keys, (  
3860:             f"Method '{method_id}' is missing required keys: {missing_keys}"
```

```
3861:         )
3862:
3863:
3864: def test_coverage_requirement():
3865:     """Verify >=80% coverage (computed methods / total methods)."""
3866:     cal_path = Path("config/intrinsic_calibration.json")
3867:     with open(cal_path) as f:
3868:         data = json.load(f)
3869:
3870:     metadata = data["_metadata"]
3871:     total = metadata["total_methods"]
3872:     computed = metadata["computed_methods"]
3873:     coverage = metadata["coverage_percent"]
3874:
3875:     assert coverage >= 80.0, (
3876:         f"Coverage requirement FAILED: {coverage}% < 80%."
3877:         f"Need at least {int(total * 0.8)} computed methods, have {computed}. "
3878:         f"Intrinsic calibration incomplete or contaminated."
3879:     )
3880:
3881:
3882: def test_minimum_computed_executors():
3883:     """Verify at least 30 executors have status='computed'."""
3884:     cal_path = Path("config/intrinsic_calibration.json")
3885:     with open(cal_path) as f:
3886:         data = json.load(f)
3887:
3888:     computed_count = sum(
3889:         1 for method_id, method_data in data.items()
3890:         if method_id != "_metadata" and method_data.get("calibration_status") == "computed"
3891:     )
3892:
3893:     assert computed_count >= 30, (
3894:         f"Minimum executor requirement FAILED: {computed_count} < 30 computed methods. "
3895:         f"Intrinsic calibration incomplete or contaminated."
3896:     )
3897:
3898:
3899: def test_valid_intrinsic_scores():
3900:     """Verify intrinsic_score is [low, high] with valid values."""
3901:     cal_path = Path("config/intrinsic_calibration.json")
3902:     with open(cal_path) as f:
3903:         data = json.load(f)
3904:
3905:     for method_id, method_data in data.items():
3906:         if method_id == "_metadata":
3907:             continue
3908:
3909:         if method_data.get("calibration_status") == "computed":
3910:             score = method_data.get("intrinsic_score")
3911:             assert isinstance(score, list) and len(score) == 2, (
3912:                 f"Method '{method_id}' has invalid intrinsic_score format: {score}"
3913:             )
3914:             assert 0.0 <= score[0] <= 1.0 and 0.0 <= score[1] <= 1.0, (
3915:                 f"Method '{method_id}' has intrinsic_score out of range [0,1]: {score}"
3916:             )
```

```
3917:         assert score[0] <= score[1], (
3918:             f"Method '{method_id}' has invalid intrinsic_score (low > high): {score}"
3919:         )
3920:
3921:
3922: def test_valid_b_scores():
3923:     """Verify b_theory, b_impl, b_deploy are valid floats in [0,1]."""
3924:     cal_path = Path("config/intrinsic_calibration.json")
3925:     with open(cal_path) as f:
3926:         data = json.load(f)
3927:
3928:     for method_id, method_data in data.items():
3929:         if method_id == "_metadata":
3930:             continue
3931:
3932:         if method_data.get("calibration_status") == "computed":
3933:             for b_key in ["b_theory", "b_impl", "b_deploy"]:
3934:                 b_val = method_data.get(b_key)
3935:                 assert isinstance(b_val, int | float),
3936:                     f"Method '{method_id}' has invalid {b_key} type: {type(b_val)}"
3937:             )
3938:             assert 0.0 <= b_val <= 1.0,
3939:                 f"Method '{method_id}' has {b_key} out of range [0,1]: {b_val}"
3940:             )
3941:
3942:
3943: def test_calibration_status_values():
3944:     """Verify calibration_status is one of: computed, pending, excluded, none."""
3945:     cal_path = Path("config/intrinsic_calibration.json")
3946:     with open(cal_path) as f:
3947:         data = json.load(f)
3948:
3949:     valid_statuses = {"computed", "pending", "excluded", "none"}
3950:
3951:     for method_id, method_data in data.items():
3952:         if method_id == "_metadata":
3953:             continue
3954:
3955:         status = method_data.get("calibration_status")
3956:         assert status in valid_statuses,
3957:             f"Method '{method_id}' has invalid calibration_status: '{status}'. "
3958:             f"Must be one of {valid_statuses}"
3959:         )
3960:
3961:
3962: def test_layer_values():
3963:     """Verify layer is one of: engine, processor, utility."""
3964:     cal_path = Path("config/intrinsic_calibration.json")
3965:     with open(cal_path) as f:
3966:         data = json.load(f)
3967:
3968:     valid_layers = {"engine", "processor", "utility"}
3969:
3970:     for method_id, method_data in data.items():
3971:         if method_id == "_metadata":
3972:             continue
```

```
3973:         layer = method_data.get("layer")
3974:         assert layer in valid_layers, (
3975:             f"Method '{method_id}' has invalid layer: '{layer}'. "
3976:             f"Must be one of {valid_layers}"
3977:         )
3978:     )
3979:
3980:
3981: def test_metadata_structure():
3982:     """Verify _metadata contains required fields."""
3983:     cal_path = Path("config/intrinsic_calibration.json")
3984:     with open(cal_path) as f:
3985:         data = json.load(f)
3986:
3987:     assert "_metadata" in data, "JSON must contain _metadata"
3988:     metadata = data["_metadata"]
3989:
3990:     required_fields = {"version", "generated", "description", "total_methods",
3991:                        "computed_methods", "coverage_percent"}
3992:     actual_fields = set(metadata.keys())
3993:
3994:     missing = required_fields - actual_fields
3995:     assert not missing, f"_metadata missing required fields: {missing}"
3996:
3997:
3998: if __name__ == "__main__":
3999:     pytest.main([__file__, "-v"])
4000:
4001:
4002:
4003: =====
4004: FILE: tests/test_inventory_completeness.py
4005: =====
4006:
4007: #!/usr/bin/env python3
4008: """Test inventory completeness - verifies all critical methods are present
4009:
4010: Validates the methods_inventory_raw.json file generated by scan_methods_inventory.py.
4011: This test suite ensures proper scanner operation and inventory completeness.
4012:
4013: See METHODS_INVENTORY_README.md for details on the scanner workflow and integration.
4014: """
4015:
4016: import json
4017: from pathlib import Path
4018:
4019: import pytest
4020:
4021: # Test constants
4022: MINIMUM_METHOD_COUNT = 200
4023: MIN_CANONICAL_ID_PARTS = 2
4024: MIN_CLASS_METHOD_PARTS = 3
4025: MIN_TAG_RATIO = 0.3
4026: MIN_DEREK_BEACH_METHODS = 10
4027: MIN_EXECUTOR_METHODS = 5
4028:
```

```
4029: CRITICAL_METHODS = [
4030:     "analysis.derek_beach.CDAFException._format_message",
4031:     "analysis.derek_beach.CDAFException.to_dict",
4032:     "analysis.derek_beach.ConfigLoader._load_config",
4033:     "analysis.derek_beach.ConfigLoader._validate_config",
4034:     "analysis.derek_beach.BeachEvidentialTest.classify_test",
4035:     "analysis.derek_beach.BeachEvidentialTest.apply_test_logic",
4036:     "core.aggregation.AreaPolicyAggregator",
4037:     "core.aggregation.ClusterAggregator",
4038:     "core.aggregation.DimensionAggregator",
4039:     "core.aggregation.MacroAggregator",
4040:     "processing.aggregation.AreaPolicyAggregator",
4041:     "processing.aggregation.ClusterAggregator",
4042:     "processing.aggregation.DimensionAggregator",
4043:     "processing.aggregation.MacroAggregator",
4044:     "core.orchestrator.executors",
4045:     "core.orchestrator.executors_contract",
4046: ]
4047:
4048:
4049: @pytest.fixture
4050: def inventory():
4051:     """Load the inventory JSON file"""
4052:     inventory_path = Path("methods_inventory_raw.json")
4053:
4054:     if not inventory_path.exists():
4055:         pytest.fail(f"Inventory file not found: {inventory_path}")
4056:
4057:     with open(inventory_path, encoding="utf-8") as f:
4058:         return json.load(f)
4059:
4060:
4061: def test_inventory_exists():
4062:     """Verify inventory file exists"""
4063:     assert Path("methods_inventory_raw.json").exists(), "Inventory file must exist"
4064:
4065:
4066: def test_minimum_method_count(inventory):
4067:     """Verify at least 200 methods in inventory"""
4068:     total = inventory["metadata"]["total_methods"]
4069:     assert (
4070:         total >= MINIMUM_METHOD_COUNT
4071:     ), f"Insufficient methods: {total} < {MINIMUM_METHOD_COUNT}"
4072:
4073:
4074: def test_critical_files_present(inventory):
4075:     """Verify methods from critical files are present"""
4076:     methods = inventory["methods"]
4077:     source_files = {m["source_file"] for m in methods}
4078:
4079:     critical_files = [
4080:         "derek_beach.py",
4081:         "aggregation.py",
4082:         "executors.py",
4083:         "executors_contract.py",
4084:     ]
```

```
4085:  
4086:     for critical_file in critical_files:  
4087:         found = any(critical_file in sf for sf in source_files)  
4088:         assert found, f"Critical file not found in inventory: {critical_file}"  
4089:  
4090:  
4091: def test_critical_method_patterns(inventory):  
4092:     """Verify critical method patterns are present"""  
4093:     methods = inventory["methods"]  
4094:     canonical_ids = {m["canonical_identifier"] for m in methods}  
4095:  
4096:     patterns_to_check = [  
4097:         "derek_beach",  
4098:         "aggregation",  
4099:         "executors",  
4100:     ]  
4101:  
4102:     for pattern in patterns_to_check:  
4103:         found = any(pattern in cid.lower() for cid in canonical_ids)  
4104:         assert found, f"No methods found matching pattern: {pattern}"  
4105:  
4106:  
4107: def test_all_roles_present(inventory):  
4108:     """Verify all expected roles are present"""  
4109:     stats = inventory["statistics"]["by_role"]  
4110:  
4111:     expected_roles = [  
4112:         "ingest",  
4113:         "processor",  
4114:         "analyzer",  
4115:         "extractor",  
4116:         "score",  
4117:         "utility",  
4118:         "orchestrator",  
4119:         "core",  
4120:         "executor",  
4121:     ]  
4122:  
4123:     for role in expected_roles:  
4124:         assert role in stats, f"Role not found in inventory: {role}"  
4125:  
4126:  
4127: def test_calibration_flags_set(inventory):  
4128:     """Verify calibration flags are properly set"""  
4129:     methods = inventory["methods"]  
4130:  
4131:     calibration_count = sum(1 for m in methods if m["requiere_calibracion"])  
4132:     parametrization_count = sum(1 for m in methods if m["requiere_parametrizacion"])  
4133:  
4134:     assert calibration_count > 0, "No methods flagged for calibration"  
4135:     assert parametrization_count > 0, "No methods flagged for parametrization"  
4136:  
4137:  
4138: def test_canonical_identifier_format(inventory):  
4139:     """Verify canonical identifiers follow module.Class.method format"""  
4140:     methods = inventory["methods"]
```

```
4141:
4142:     for method in methods:
4143:         cid = method["canonical_identifier"]
4144:         parts = cid.split(".")
4145:
4146:         assert (
4147:             len(parts) >= MIN_CANONICAL_ID_PARTS
4148:         ), f"Invalid canonical ID format: {cid}"
4149:
4150:         if method["class_name"]:
4151:             assert (
4152:                 len(parts) >= MIN_CLASS_METHOD_PARTS
4153:             ), f"Class method must have at least {MIN_CLASS_METHOD_PARTS} parts: {cid}"
4154:
4155:
4156: def test_epistemology_tags_present(inventory):
4157:     """Verify epistemology tags are assigned"""
4158:     methods = inventory["methods"]
4159:
4160:     tagged_count = sum(1 for m in methods if m["epistemology_tags"])
4161:     total = len(methods)
4162:
4163:     assert tagged_count > 0, "No methods have epistemology tags"
4164:
4165:     tag_ratio = tagged_count / total
4166:     assert tag_ratio > MIN_TAG_RATIO, f"Too few methods tagged: {tag_ratio:.2%}"
4167:
4168:
4169: def test_derek_beach_methods_complete(inventory):
4170:     """Verify derek_beach.py methods are complete"""
4171:     methods = inventory["methods"]
4172:     derek_methods = [m for m in methods if "derek_beach" in m["source_file"]]
4173:
4174:     assert (
4175:         len(derek_methods) > MIN_DEREK_BEACH_METHODS
4176:     ), f"Too few derek_beach methods: {len(derek_methods)}"
4177:
4178:     required_patterns = ["_format_message", "to_dict", "_load_config", "classify_test"]
4179:
4180:     for pattern in required_patterns:
4181:         found = any(pattern in m["method_name"] for m in derek_methods)
4182:         assert found, f"Required derek_beach method not found: {pattern}"
4183:
4184:
4185: def test_aggregation_classes_present(inventory):
4186:     """Verify aggregation classes are present"""
4187:     methods = inventory["methods"]
4188:     aggregation_methods = [
4189:         m for m in methods if "aggregation" in m["source_file"].lower()
4190:     ]
4191:
4192:     assert len(aggregation_methods) > 0, "No aggregation methods found"
4193:
4194:     required_classes = [
4195:         "AreaPolicyAggregator",
4196:         "ClusterAggregator",
```

```
4197:         "DimensionAggregator",
4198:     ]
4199:
4200:     found_classes = {m["class_name"] for m in aggregation_methods if m["class_name"]}
4201:
4202:     for req_class in required_classes:
4203:         assert (
4204:             req_class in found_classes
4205:         ), f"Required aggregation class not found: {req_class}"
4206:
4207:
4208: def test_executor_methods_present(inventory):
4209:     """Verify executor methods are present"""
4210:     methods = inventory["methods"]
4211:     executor_methods = [m for m in methods if "executor" in m["source_file"].lower()]
4212:
4213:     assert (
4214:         len(executor_methods) > MIN_EXECUTOR_METHODS
4215:     ), f"Too few executor methods: {len(executor_methods)}"
4216:
4217:
4218: def test_no_duplicate_canonical_ids(inventory):
4219:     """Verify no duplicate canonical identifiers"""
4220:     methods = inventory["methods"]
4221:     canonical_ids = [m["canonical_identifier"] for m in methods]
4222:
4223:     duplicates = [cid for cid in canonical_ids if canonical_ids.count(cid) > 1]
4224:
4225:     assert len(duplicates) == 0, f"Duplicate canonical IDs found: {set(duplicates)}"
4226:
4227:
4228: def test_layer_requirements_complete(inventory):
4229:     """Verify LAYER_REQUIREMENTS table is complete"""
4230:     layer_requirements = inventory["layer_requirements"]
4231:
4232:     expected_layers = [
4233:         "ingest",
4234:         "processor",
4235:         "analyzer",
4236:         "extractor",
4237:         "score",
4238:         "utility",
4239:         "orchestrator",
4240:         "core",
4241:         "executor",
4242:     ]
4243:
4244:     for layer in expected_layers:
4245:         assert layer in layer_requirements, f"Layer missing from requirements: {layer}"
4246:         assert "description" in layer_requirements[layer]
4247:         assert "typical_patterns" in layer_requirements[layer]
4248:
4249:
4250: if __name__ == "__main__":
4251:     pytest.main([__file__, "-v"])
4252:
```

```
4253:  
4254:  
4255: =====  
4256: FILE: tests/test_layer_assignment.py  
4257: =====  
4258:  
4259: """  
4260: Test suite for layer assignment system.  
4261:  
4262: VERIFICATION CONDITIONS:  
4263: 1. All methods must have layer mappings  
4264: 2. All executors must have exactly 8 layers  
4265: 3. JSON contains ONLY metadata (no numeric scores outside weights)  
4266: 4. Sum of weights per executor must equal 1.0  
4267:  
4268: FAILURE CONDITIONS:  
4269: - If <30 executors identified: ABORT with 'layer assignment corrupted'  
4270: - If any score appears in JSON (outside weights): ABORT with 'layer assignment corrupted'  
4271:  
4272: DEPRECATED: Test assumes outdated canonic_inventorry_methods_layers.json structure.  
4273: See tests/DEPRECATED_TESTS.md for details.  
4274: """  
4275:  
4276: import json  
4277: from pathlib import Path  
4278:  
4279: import pytest  
4280:  
4281: pytestmark = pytest.mark.obsolete  
4282:  
4283: REPO_ROOT = Path(__file__).parent.parent  
4284: CONFIG_FILE = REPO_ROOT / "config" / "canonic_inventorry_methods_layers.json"  
4285: EXECUTORS_FILE = (  
4286:     REPO_ROOT / "src" / "farfan_pipeline" / "core" / "orchestrator" / "executors.py"  
4287: )  
4288:  
4289:  
4290: class TestLayerAssignment:  
4291:  
4292:     @pytest.fixture  
4293:     def inventory_data(self):  
4294:         """Load the canonical inventory JSON."""  
4295:         if not CONFIG_FILE.exists():  
4296:             pytest.fail(f"Config file not found: {CONFIG_FILE}")  
4297:  
4298:         with open(CONFIG_FILE) as f:  
4299:             return json.load(f)  
4300:  
4301:     def test_minimum_executor_count(self, inventory_data):  
4302:         """Verify at least 30 executors are identified."""  
4303:         method_count = len(inventory_data["methods"])  
4304:         assert (  
4305:             method_count >= 30  
4306:         ), f"layer assignment corrupted: Found {method_count} executors, expected 30"  
4307:  
4308:     def test_all_methods_have_layer_mappings(self, inventory_data):
```

```
4309:     """Verify every method has layer mappings."""
4310:     for method_id, method_data in inventory_data["methods"].items():
4311:         assert (
4312:             "layers" in method_data
4313:         ), f"layer assignment corrupted: Method {method_id} missing 'layers'"
4314:         assert isinstance(
4315:             method_data["layers"], list
4316:         ), f"layer assignment corrupted: Method {method_id} 'layers' is not a list"
4317:         assert (
4318:             len(method_data["layers"]) > 0
4319:         ), f"layer assignment corrupted: Method {method_id} has empty 'layers'"
4320:
4321:     def test_all_executors_have_8_layers(self, inventory_data):
4322:         """Verify all executors have exactly 8 layers."""
4323:         for method_id, method_data in inventory_data["methods"].items():
4324:             if method_data.get("role") == "executor":
4325:                 layer_count = len(method_data["layers"])
4326:                 assert layer_count == 8, (
4327:                     f"layer assignment corrupted: Executor {method_id} has "
4328:                     f"{layer_count} layers, expected 8"
4329:                 )
4330:
4331:     def test_no_numeric_scores_in_json(self, inventory_data):
4332:         """Verify JSON contains only metadata, not numeric scores."""
4333:
4334:         def check_for_scores(obj, path=""):
4335:             if isinstance(obj, dict):
4336:                 for key, value in obj.items():
4337:                     current_path = f"{path}.{key}" if path else key
4338:
4339:                     if key in ["weights", "interaction_weights"]:
4340:                         continue
4341:
4342:                     if key in ["version", "total_executors"]:
4343:                         continue
4344:
4345:                     if isinstance(value, int | float) and key not in [
4346:                         "version",
4347:                         "total_executors",
4348:                     ]:
4349:                         if current_path.endswith(("weights", "interaction_weights")):
4350:                             continue
4351:                         pytest.fail(
4352:                             f"layer assignment corrupted: Found numeric score in JSON "
4353:                             f"at {current_path}={value}"
4354:                         )
4355:
4356:                         check_for_scores(value, current_path)
4357:
4358:                     elif isinstance(obj, list):
4359:                         for i, item in enumerate(obj):
4360:                             check_for_scores(item, f"{path}[{i}]")
4361:
4362:                         check_for_scores(inventory_data)
4363:
4364:     def test_weights_sum_to_one(self, inventory_data):
```

```

4365:     """Verify sum of weights per executor equals 1.0."""
4366:     for method_id, method_data in inventory_data["methods"].items():
4367:         weights = method_data.get("weights", {})
4368:         interaction_weights = method_data.get("interaction_weights", {})
4369:
4370:         total = sum(weights.values()) + sum(interaction_weights.values())
4371:
4372:         assert abs(total - 1.0) < 0.01, (
4373:             f"layer assignment corrupted: Weights for {method_id} "
4374:             f"sum to {total:.4f}, expected 1.0"
4375:         )
4376:
4377:     def test_required_metadata_fields(self, inventory_data):
4378:         """Verify all methods have required metadata fields."""
4379:         required_fields = ["method_id", "role", "layers", "weights", "aggregator_type"]
4380:
4381:         for method_id, method_data in inventory_data["methods"].items():
4382:             for field in required_fields:
4383:                 assert field in method_data, (
4384:                     f"layer assignment corrupted: Method {method_id} missing "
4385:                     f"required field '{field}'"
4386:                 )
4387:
4388:     def test_layer_names_are_valid(self, inventory_data):
4389:         """Verify all layer names match the canonical set."""
4390:         valid_layers = {"@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"}
4391:
4392:         for method_id, method_data in inventory_data["methods"].items():
4393:             layers = set(method_data["layers"])
4394:             invalid = layers - valid_layers
4395:
4396:             assert not invalid, (
4397:                 f"layer assignment corrupted: Method {method_id} has invalid "
4398:                 f"layers: {invalid}"
4399:             )
4400:
4401:     def test_weights_match_layers(self, inventory_data):
4402:         """Verify weights dict keys match assigned layers."""
4403:         for method_id, method_data in inventory_data["methods"].items():
4404:             layers = set(method_data["layers"])
4405:             weight_keys = set(method_data["weights"].keys())
4406:
4407:             assert weight_keys == layers, (
4408:                 f"layer assignment corrupted: Method {method_id} weight keys "
4409:                 f"{weight_keys} don't match layers {layers}"
4410:             )
4411:
4412:     def test_executor_pattern_matching(self):
4413:         """Verify executor pattern D[1-6]Q[1-5] is correctly identified."""
4414:         import re
4415:
4416:         with open(EXECUTORS_FILE) as f:
4417:             content = f.read()
4418:
4419:             pattern = re.compile(r"class (D([1-6])_Q([1-5])_\w+)\(")
4420:             matches = pattern.findall(content)

```

```
4421:  
4422:     assert len(matches) >= 30, (  
4423:         f"layer assignment corrupted: Found {len(matches)} executor classes, "  
4424:         f"expected 30"  
4425:     )  
4426:  
4427:     for _class_name, dim, question in matches:  
4428:         assert dim in "123456", f"Invalid dimension: {dim}"  
4429:         assert question in "12345", f"Invalid question: {question}"  
4430:  
4431:     def test_aggregator_type_is_choquet(self, inventory_data):  
4432:         """Verify all executors use Choquet aggregator."""  
4433:         for method_id, method_data in inventory_data["methods"].items():  
4434:             if method_data.get("role") == "executor":  
4435:                 assert method_data["aggregator_type"] == "choquet", (  
4436:                     f"layer assignment corrupted: Executor {method_id} has "  
4437:                     f"aggregator_type '{method_data['aggregator_type']}'", expected 'choquet'  
4438:                 )  
4439:  
4440:     def test_interaction_weights_are_valid(self, inventory_data):  
4441:         """Verify interaction weights reference valid layer pairs."""  
4442:         for method_id, method_data in inventory_data["methods"].items():  
4443:             layers = set(method_data["layers"])  
4444:  
4445:             for pair_key in method_data.get("interaction_weights", {}):  
4446:                 if "," not in pair_key:  
4447:                     pytest.fail(  
4448:                         f"layer assignment corrupted: Invalid interaction weight key "  
4449:                         f"'{pair_key}' in {method_id}"  
4450:                     )  
4451:  
4452:                     l1, l2 = pair_key.split(",")  
4453:                     assert l1 in layers, (  
4454:                         f"layer assignment corrupted: Interaction weight {pair_key} in "  
4455:                         f"{method_id} references layer {l1} not in assigned layers"  
4456:                     )  
4457:                     assert l2 in layers, (  
4458:                         f"layer assignment corrupted: Interaction weight {pair_key} in "  
4459:                         f"{method_id} references layer {l2} not in assigned layers"  
4460:                     )  
4461:  
4462:     def test_dimension_and_question_metadata(self, inventory_data):  
4463:         """Verify executors have dimension and question metadata."""  
4464:         for method_id, method_data in inventory_data["methods"].items():  
4465:             if method_data.get("role") == "executor":  
4466:                 assert (  
4467:                     "dimension" in method_data  
4468:                 ), f"layer assignment corrupted: Executor {method_id} missing 'dimension'"  
4469:                 assert (  
4470:                     "question" in method_data  
4471:                 ), f"layer assignment corrupted: Executor {method_id} missing 'question'"  
4472:  
4473:                 dim = method_data["dimension"]  
4474:                 question = method_data["question"]  
4475:  
4476:                 assert (
```

```
4477:             dim.startswith("D") and dim[1:].isdigit()
4478:         ), f"layer assignment corrupted: Invalid dimension '{dim}' in {method_id}"
4479:         assert (
4480:             question.startswith("Q") and question[1:].isdigit()
4481:         ), f"layer assignment corrupted: Invalid question '{question}' in {method_id}"
4482:
4483:
4484: class TestLayerRequirements:
4485:     """Test the LAYER_REQUIREMENTS table definition."""
4486:
4487:     def test_layer_requirements_completeness(self):
4488:         """Verify LAYER_REQUIREMENTS covers all specified roles."""
4489:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4490:
4491:         expected_roles = [
4492:             "ingest",
4493:             "processor",
4494:             "analyzer",
4495:             "score",
4496:             "executor",
4497:             "utility",
4498:             "orchestrator",
4499:             "core",
4500:             "extractor",
4501:         ]
4502:
4503:         for role in expected_roles:
4504:             assert (
4505:                 role in LAYER_REQUIREMENTS
4506:             ), f"layer assignment corrupted: Role '{role}' missing from LAYER_REQUIREMENTS"
4507:
4508:     def test_ingest_layers(self):
4509:         """Verify ingest role has correct layers."""
4510:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4511:
4512:         expected = ["@b", "@chain", "@u", "@m"]
4513:         assert LAYER_REQUIREMENTS["ingest"] == expected
4514:
4515:     def test_processor_layers(self):
4516:         """Verify processor role has correct layers."""
4517:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4518:
4519:         expected = ["@b", "@chain", "@u", "@m"]
4520:         assert LAYER_REQUIREMENTS["processor"] == expected
4521:
4522:     def test_analyzer_layers(self):
4523:         """Verify analyzer role has correct layers."""
4524:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4525:
4526:         expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
4527:         assert LAYER_REQUIREMENTS["analyzer"] == expected
4528:
4529:     def test_score_layers(self):
4530:         """Verify score role has all 8 layers."""
4531:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4532:
```

```
4533:     expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
4534:     assert LAYER_REQUIREMENTS["score"] == expected
4535:
4536:     def test_executor_layers(self):
4537:         """Verify executor role has all 8 layers."""
4538:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4539:
4540:         expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
4541:         assert LAYER_REQUIREMENTS["executor"] == expected
4542:
4543:     def test_utility_layers(self):
4544:         """Verify utility role has correct layers."""
4545:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4546:
4547:         expected = ["@b", "@chain", "@m"]
4548:         assert LAYER_REQUIREMENTS["utility"] == expected
4549:
4550:     def test_orchestrator_layers(self):
4551:         """Verify orchestrator role has correct layers."""
4552:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4553:
4554:         expected = ["@b", "@chain", "@m"]
4555:         assert LAYER_REQUIREMENTS["orchestrator"] == expected
4556:
4557:     def test_core_layers(self):
4558:         """Verify core role has all 8 layers."""
4559:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4560:
4561:         expected = ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"]
4562:         assert LAYER_REQUIREMENTS["core"] == expected
4563:
4564:     def test_extractor_layers(self):
4565:         """Verify extractor role has correct layers."""
4566:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS
4567:
4568:         expected = ["@b", "@chain", "@u", "@m"]
4569:         assert LAYER_REQUIREMENTS["extractor"] == expected
4570:
4571:
4572: if __name__ == "__main__":
4573:     pytest.main([__file__, "-v"])
4574:
4575:
4576:
4577: =====
4578: FILE: tests/test_memory_safety.py
4579: =====
4580:
4581: """
4582: Tests for memory safety guards in executor system.
4583: """
4584:
4585: import pytest
4586: from farfan_pipeline.core.orchestrator.memory_safety import (
4587:     MemorySafetyGuard,
4588:     MemorySafetyConfig,
```

```
4589:     ExecutorType,
4590:     ObjectSizeEstimator,
4591:     FallbackStrategy,
4592:     MemoryPressureDetector,
4593: )
4594:
4595:
4596: class TestObjectSizeEstimator:
4597:     def test_estimate_object_size_simple(self):
4598:         assert ObjectSizeEstimator.estimate_object_size(42) > 0
4599:         assert ObjectSizeEstimator.estimate_object_size("hello") > 0
4600:         assert ObjectSizeEstimator.estimate_object_size([1, 2, 3]) > 0
4601:
4602:     def test_estimate_json_size(self):
4603:         assert ObjectSizeEstimator.estimate_json_size(None) == 4
4604:         assert ObjectSizeEstimator.estimate_json_size(True) == 5
4605:         assert ObjectSizeEstimator.estimate_json_size(42) > 0
4606:         assert ObjectSizeEstimator.estimate_json_size("test") > 0
4607:         assert ObjectSizeEstimator.estimate_json_size({"key": "value"}) > 0
4608:         assert ObjectSizeEstimator.estimate_json_size([1, 2, 3]) > 0
4609:
4610:     def test_estimate_large_list(self):
4611:         large_list = [{"id": i, "data": "x" * 100} for i in range(1000)]
4612:         size = ObjectSizeEstimator.estimate_json_size(large_list)
4613:         assert size > 100_000
4614:
4615:
4616: class TestFallbackStrategy:
4617:     def test_sample_list(self):
4618:         items = list(range(1000))
4619:         sampled = FallbackStrategy.sample_list(items, 100)
4620:         assert len(sampled) == 100
4621:         assert all(item in items for item in sampled)
4622:
4623:     def test_sample_list_preserve_order(self):
4624:         items = list(range(100))
4625:         sampled = FallbackStrategy.sample_list(items, 10, preserve_order=True)
4626:         assert len(sampled) == 10
4627:         assert sampled == sorted(sampled)
4628:
4629:     def test_truncate_string(self):
4630:         long_str = "x" * 1000
4631:         truncated = FallbackStrategy.truncate_string(long_str, 100)
4632:         assert len(truncated) == 100
4633:         assert truncated.endswith("...") 
4634:
4635:     def test_truncate_dict(self):
4636:         large_dict = {f"key_{i}": i for i in range(1000)}
4637:         truncated = FallbackStrategy.truncate_dict(large_dict, 10)
4638:         assert len(truncated) <= 10
4639:
4640:     def test_apply_recursive_truncation(self):
4641:         config = MemorySafetyConfig(
4642:             max_list_elements=10,
4643:             max_string_length=50,
4644:             max_dict_keys=5
```

```
4645:         )
4646:
4647:     obj = {
4648:         "list": list(range(100)),
4649:         "string": "x" * 1000,
4650:         "dict": {f"key_{i}": i for i in range(100)},
4651:         "nested": {
4652:             "data": list(range(50))
4653:         }
4654:     }
4655:
4656:     result, modified = FallbackStrategy.apply_recursive_truncation(obj, config)
4657:     assert modified
4658:     assert len(result["list"]) <= 10
4659:     assert len(result["string"]) <= 50
4660:     assert len(result["dict"]) <= 5
4661:     assert len(result["nested"]["data"]) <= 10
4662:
4663:
4664: class TestMemorySafetyGuard:
4665:     def test_check_small_object(self):
4666:         guard = MemorySafetyGuard()
4667:         obj = {"id": 1, "name": "test"}
4668:         result, metrics = guard.check_and_process(obj, ExecutorType.GENERIC, "test_obj")
4669:
4670:         assert result == obj
4671:         assert metrics.object_size_bytes > 0
4672:         assert metrics.json_size_bytes > 0
4673:         assert not metrics.was_truncated
4674:         assert not metrics.was_sampled
4675:
4676:     def test_check_large_object_entity_type(self):
4677:         guard = MemorySafetyGuard(MemorySafetyConfig(
4678:             entity_limit_mb=0.001,
4679:             enable_auto_truncation=True
4680:         ))
4681:
4682:         large_obj = {"entities": [{"id": i, "data": "x" * 1000} for i in range(100)]}
4683:         result, metrics = guard.check_and_process(large_obj, ExecutorType.ENTITY, "entities")
4684:
4685:         assert metrics.object_size_bytes > guard.config.get_limit_bytes(ExecutorType.ENTITY) or \
4686:             metrics.json_size_bytes > guard.config.get_limit_bytes(ExecutorType.ENTITY) or \
4687:                 metrics.was_truncated
4688:
4689:     def test_check_large_list(self):
4690:         guard = MemorySafetyGuard(MemorySafetyConfig(
4691:             generic_limit_mb=0.001,
4692:             max_list_elements=10,
4693:             enable_auto_truncation=True
4694:         ))
4695:
4696:         large_list = list(range(1000))
4697:         result, metrics = guard.check_and_process(large_list, ExecutorType.GENERIC, "large_list")
4698:
4699:         if metrics.was_truncated:
4700:             assert len(result) <= 10
```

```
4701:
4702:     def test_get_metrics_summary(self):
4703:         guard = MemorySafetyGuard()
4704:
4705:         for i in range(5):
4706:             obj = {"id": i, "data": "test" * 100}
4707:             guard.check_and_process(obj, ExecutorType.GENERIC, f"obj_{i}")
4708:
4709:             summary = guard.get_metrics_summary()
4710:             assert summary["total_operations"] == 5
4711:             assert summary["avg_object_size_mb"] >= 0
4712:             assert summary["max_object_size_mb"] >= summary["avg_object_size_mb"]
4713:
4714:
4715: class TestMemoryPressureDetector:
4716:     def test_get_memory_pressure_pct(self):
4717:         pressure = MemoryPressureDetector.get_memory_pressure_pct()
4718:         if pressure is not None:
4719:             assert 0 <= pressure <= 100
4720:
4721:     def test_is_under_pressure(self):
4722:         result = MemoryPressureDetector.is_under_pressure(threshold_pct=99.0)
4723:         assert isinstance(result, bool)
4724:
4725:
4726: class TestMemorySafetyConfig:
4727:     def test_get_limit_bytes(self):
4728:         config = MemorySafetyConfig()
4729:
4730:         entity_limit = config.get_limit_bytes(ExecutorType.ENTITY)
4731:         assert entity_limit == 1 * 1024 * 1024
4732:
4733:         dag_limit = config.get_limit_bytes(ExecutorType.DAG)
4734:         assert dag_limit == 5 * 1024 * 1024
4735:
4736:         causal_limit = config.get_limit_bytes(ExecutorType.CAUSAL_EFFECTS)
4737:         assert causal_limit == 10 * 1024 * 1024
4738:
4739:     def test_custom_config(self):
4740:         config = MemorySafetyConfig(
4741:             entity_limit_mb=2.0,
4742:             dag_limit_mb=10.0,
4743:             max_list_elements=500
4744:         )
4745:
4746:         assert config.get_limit_bytes(ExecutorType.ENTITY) == 2 * 1024 * 1024
4747:         assert config.get_limit_bytes(ExecutorType.DAG) == 10 * 1024 * 1024
4748:         assert config.max_list_elements == 500
4749:
4750:
4751:
4752: =====
4753: FILE: tests/test_opentelemetry_observability.py
4754: =====
4755:
4756: """Tests for production OpenTelemetry integration.
```

```
4757:  
4758: Tests cover:  
4759: - Configuration and initialization  
4760: - Tracing functionality  
4761: - Metrics collection  
4762: - Executor instrumentation  
4763: - Pipeline instrumentation  
4764: - Context propagation  
4765:  
4766: DEPRECATED: Test uses outdated farfan_core module path, should use farfan_pipeline.  
4767: See tests/DEPRECATED_TESTS.md for details.  
4768: """  
4769:  
4770: import pytest  
4771: from unittest.mock import MagicMock, patch  
4772:  
4773: from farfan_core.observability import (  
4774:     OpenTelemetryConfig,  
4775:     FARFANObservability,  
4776:     get_global_observability,  
4777:     initialize_observability,  
4778: )  
4779:  
4780: pytestmark = pytest.mark.obsolete  
4781:  
4782:  
4783: class TestOpenTelemetryConfig:  
4784:     """Test OpenTelemetry configuration."""  
4785:  
4786:     def test_default_config(self):  
4787:         config = OpenTelemetryConfig()  
4788:         assert config.service_name == "farfan-pipeline"  
4789:         assert config.service_version == "1.0.0"  
4790:         assert config.prometheus_port == 9090  
4791:  
4792:     def test_custom_config(self):  
4793:         config = OpenTelemetryConfig(  
4794:             service_name="test-service",  
4795:             service_version="2.0.0",  
4796:             prometheus_port=8080,  
4797:         )  
4798:         assert config.service_name == "test-service"  
4799:         assert config.service_version == "2.0.0"  
4800:         assert config.prometheus_port == 8080  
4801:  
4802:  
4803: class TestFARFANObservability:  
4804:     """Test main observability class."""  
4805:  
4806:     @patch("farfan_core.observability.opentelemetry_integration.OTEL_AVAILABLE", False)  
4807:     def test_initialization_without_otel(self):  
4808:         obs = FARFANObservability()  
4809:         assert not obs._initialized  
4810:         assert obs.get_tracer() is None  
4811:         assert obs.get_meter() is None  
4812:
```

```
4813:     def test_start_span_without_otel(self):
4814:         obs = FARFANObservability()
4815:         obs._initialized = False
4816:
4817:         with obs.start_span("test_span") as span:
4818:             assert span is None
4819:
4820:     def test_trace_executor_without_otel(self):
4821:         obs = FARFANObservability()
4822:         obs._initialized = False
4823:
4824:         @obs.trace_executor("D1_Q1_TestExecutor")
4825:         def test_execute(context):
4826:             return {"result": "success"}
4827:
4828:         result = test_execute({"test": "data"})
4829:         assert result == {"result": "success"}
4830:
4831:     def test_propagate_context_without_otel(self):
4832:         obs = FARFANObservability()
4833:         obs._initialized = False
4834:
4835:         carrier = obs.propagate_context()
4836:         assert carrier == {}
4837:
4838:
4839: class TestGlobalObservability:
4840:     """Test global observability singleton."""
4841:
4842:     def test_get_global_observability(self):
4843:         obs1 = get_global_observability()
4844:         obs2 = get_global_observability()
4845:         assert obs1 is obs2
4846:
4847:     def test_initialize_observability(self):
4848:         obs = initialize_observability(
4849:             service_name="test-service",
4850:             enable_jaeger=False,
4851:             enable_prometheus=False,
4852:             auto_instrument=False,
4853:         )
4854:         assert isinstance(obs, FARFANObservability)
4855:         assert obs.config.service_name == "test-service"
4856:
4857:
4858: if __name__ == "__main__":
4859:     pytest.main([__file__, "-v"])
4860:
4861:
4862:
4863: =====
4864: FILE: tests/test_phase6_7_8_integration.py
4865: =====
4866:
4867: """
4868: Integration Tests for Phase 6-7-8 Pipeline
```

```
4869: =====
4870:
4871: Tests the complete flow:
4872: Phase 6 (Schema Validation) à\206\222 Phase 7 (Task Construction) à\206\222 Phase 8 (Execution Plan)
4873:
4874: Verifies:
4875: 1. Successful end-to-end integration with valid data
4876: 2. Error propagation from Phase 6 à\206\222 Phase 7 à\206\222 Phase 8
4877: 3. Field validation and type checking
4878: 4. Duplicate detection across phases
4879: 5. Non-null constraint enforcement
4880: 6. Deterministic identifier generation
4881: 7. Integrity hash computation
4882: """
4883:
4884: import pytest
4885: from typing import Any
4886:
4887: from src.farfan_pipeline.core.phases.phase6_schema_validation import (
4888:     phase6_schema_validation,
4889:     Phase6SchemaValidationOutput,
4890:     PHASE6_VERSION,
4891: )
4892: from src.farfan_pipeline.core.phases.phase7_task_construction import (
4893:     phase7_task_construction,
4894:     Phase7TaskConstructionOutput,
4895:     PHASE7_VERSION,
4896:     TASK_ID_FORMAT,
4897: )
4898: from src.farfan_pipeline.core.phases.phase8_execution_plan import (
4899:     phase8_execution_plan_assembly,
4900:     Phase8ExecutionPlanOutput,
4901:     PHASE8_VERSION,
4902: )
4903:
4904:
4905: def create_valid_question(index: int) -> dict[str, Any]:
4906:     """Create a valid question for testing."""
4907:     return {
4908:         "question_id": f"Q{index:03d}",
4909:         "question_global": index,
4910:         "dimension_id": f"D{(index % 6) + 1}",
4911:         "policy_area_id": f"PA{(index % 10) + 1:02d}",
4912:         "base_slot": f"slot_{index}",
4913:         "cluster_id": f"cluster_{index % 4}",
4914:         "patterns": [{"type": "regex", "value": "test.*"}],
4915:         "signals": {"signal_type": "evidence", "strength": 0.8},
4916:         "expected_elements": [{"type": "text", "required": True}],
4917:         "metadata": {"source": "test"},
4918:     }
4919:
4920:
4921: def create_valid_chunk(index: int) -> dict[str, Any]:
4922:     """Create a valid chunk for testing."""
4923:     return {
4924:         "chunk_id": f"chunk_{index:03d}",
```

```
4925:         "policy_area_id": f"PA{(index % 10) + 1:02d}",
4926:         "dimension_id": f"D{(index % 6) + 1}",
4927:         "document_position": index,
4928:         "content": f"Test content for chunk {index}",
4929:         "metadata": {"page": index // 10},
4930:     }
4931:
4932:
4933: class TestPhase6SchemaValidation:
4934:     """Test Phase 6 schema validation functionality."""
4935:
4936:     def test_valid_questions_and_chunks(self):
4937:         """Test successful validation with valid data."""
4938:         questions = [create_valid_question(i) for i in range(10)]
4939:         chunks = [create_valid_chunk(i) for i in range(10)]
4940:
4941:         result = phase6_schema_validation(questions, chunks)
4942:
4943:         assert result.schema_validation_passed
4944:         assert len(result.validation_errors) == 0
4945:         assert result.question_count == 10
4946:         assert result.chunk_count == 10
4947:         assert len(result.validated_questions) == 10
4948:         assert len(result.validated_chunks) == 10
4949:         assert result.phase6_version == PHASE6_VERSION
4950:
4951:     def test_missing_required_field_question(self):
4952:         """Test error handling for missing required field in question."""
4953:         questions = [{"question_id": "Q001"}]
4954:         chunks = []
4955:
4956:         result = phase6_schema_validation(questions, chunks)
4957:
4958:         assert not result.schema_validation_passed
4959:         assert len(result.validation_errors) > 0
4960:         assert any("question_global" in err for err in result.validation_errors)
4961:         assert any("KeyError" in err for err in result.validation_errors)
4962:
4963:     def test_invalid_type_question_global(self):
4964:         """Test error handling for invalid type."""
4965:         question = create_valid_question(0)
4966:         question["question_global"] = "not_an_int"
4967:
4968:         result = phase6_schema_validation([question], [])
4969:
4970:         assert not result.schema_validation_passed
4971:         assert any("invalid type" in err for err in result.validation_errors)
4972:         assert any("expected int" in err for err in result.validation_errors)
4973:
4974:     def test_out_of_range_question_global(self):
4975:         """Test error handling for out-of-range value."""
4976:         question = create_valid_question(0)
4977:         question["question_global"] = 1000
4978:
4979:         result = phase6_schema_validation([question], [])
4980:
```

```
4981:         assert not result.schema_validation_passed
4982:         assert any("not in range" in err for err in result.validation_errors)
4983:
4984:     def test_empty_string_field(self):
4985:         """Test error handling for empty string in required field."""
4986:         question = create_valid_question(0)
4987:         question["question_id"] = ""
4988:
4989:         result = phase6_schema_validation([question], [])
4990:
4991:         assert not result.schema_validation_passed
4992:         assert any("empty string" in err for err in result.validation_errors)
4993:
4994:     def test_missing_chunk_content(self):
4995:         """Test error handling for missing chunk content."""
4996:         chunks = [{"chunk_id": "chunk_001"}]
4997:
4998:         result = phase6_schema_validation([], chunks)
4999:
5000:         assert not result.schema_validation_passed
5001:         assert any("content" in err for err in result.validation_errors)
5002:         assert any("KeyError" in err for err in result.validation_errors)
5003:
5004:
5005: class TestPhase7TaskConstruction:
5006:     """Test Phase 7 task construction functionality."""
5007:
5008:     def test_successful_task_construction(self):
5009:         """Test successful task construction with valid Phase 6 output."""
5010:         questions = [create_valid_question(i) for i in range(5)]
5011:         chunks = [create_valid_chunk(i) for i in range(5)]
5012:
5013:         phase6_output = phase6_schema_validation(questions, chunks)
5014:         assert phase6_output.schema_validation_passed
5015:
5016:         phase7_output = phase7_task_construction(phase6_output)
5017:
5018:         assert phase7_output.construction_passed
5019:         assert len(phase7_output.construction_errors) == 0
5020:         assert phase7_output.task_count == 5
5021:         assert len(phase7_output.tasks) == 5
5022:         assert phase7_output.phase7_version == PHASE7_VERSION
5023:
5024:     def test_task_id_format(self):
5025:         """Test task ID generation with zero-padded format."""
5026:         questions = [create_valid_question(i) for i in range(3)]
5027:         chunks = [create_valid_chunk(i) for i in range(3)]
5028:
5029:         phase6_output = phase6_schema_validation(questions, chunks)
5030:         phase7_output = phase7_task_construction(phase6_output)
5031:
5032:         assert phase7_output.construction_passed
5033:
5034:         for i, task in enumerate(phase7_output.tasks):
5035:             expected_id = TASK_ID_FORMAT.format(
5036:                 question_global=i,
```

```
5037:             policy_area_id=f"PA{(i % 10) + 1:02d}"
5038:         )
5039:         assert task.task_id == expected_id
5040:         assert len(task.task_id.split("-")[1].split("_")) == 3
5041:
5042:     def test_phase6_failure_propagation(self):
5043:         """Test error propagation when Phase 6 fails."""
5044:         phase6_output = Phase6SchemaValidationOutput(
5045:             validated_questions=[],
5046:             validated_chunks=[],
5047:             schema_validation_passed=False,
5048:             validation_errors=["Phase 6 test error"],
5049:             validation_warnings=[],
5050:             question_count=0,
5051:             chunk_count=0,
5052:             validation_timestamp="2025-01-19T00:00:00Z",
5053:         )
5054:
5055:         phase7_output = phase7_task_construction(phase6_output)
5056:
5057:         assert not phase7_output.construction_passed
5058:         assert len(phase7_output.construction_errors) > 0
5059:         assert any("Phase 6" in err for err in phase7_output.construction_errors)
5060:         assert phase7_output.task_count == 0
5061:         assert len(phase7_output.tasks) == 0
5062:
5063:     def test_duplicate_task_id_detection(self):
5064:         """Test duplicate task ID detection."""
5065:         questions = [
5066:             create_valid_question(0),
5067:             create_valid_question(0),
5068:         ]
5069:         chunks = [create_valid_chunk(0)]
5070:
5071:         phase6_output = phase6_schema_validation(questions, chunks)
5072:         phase7_output = phase7_task_construction(phase6_output)
5073:
5074:         assert not phase7_output.construction_passed
5075:         assert len(phase7_output.duplicate_task_ids) > 0
5076:         assert any("Duplicate" in err for err in phase7_output.construction_errors)
5077:
5078:     def test_missing_field_tracking(self):
5079:         """Test tracking of missing fields by task."""
5080:         from src.farfan_pipeline.core.phases.phase7_task_construction import (
5081:             ValidatedQuestionSchema,
5082:             ValidatedChunkSchema,
5083:             Phase6SchemaValidationOutput,
5084:         )
5085:
5086:         question = ValidatedQuestionSchema(
5087:             question_id="",
5088:             question_global=0,
5089:             dimension_id="D1",
5090:             policy_area_id="PA01",
5091:             base_slot="slot_0",
5092:             cluster_id="cluster_0",
```

```
5093:         patterns=[],  
5094:         signals={},  
5095:         expected_elements=[],  
5096:         metadata={},  
5097:     )  
5098:  
5099:     chunk = ValidatedChunkSchema(  
5100:         chunk_id="chunk_000",  
5101:         policy_area_id="PA01",  
5102:         dimension_id="D1",  
5103:         document_position=0,  
5104:         content="test",  
5105:         metadata={},  
5106:     )  
5107:  
5108:     phase6_output = Phase6SchemaValidationOutput(  
5109:         validated_questions=[question],  
5110:         validated_chunks=[chunk],  
5111:         schema_validation_passed=True,  
5112:         validation_errors=[],  
5113:         validation_warnings=[],  
5114:         question_count=1,  
5115:         chunk_count=1,  
5116:         validation_timestamp="2025-01-19T00:00:00Z",  
5117:     )  
5118:  
5119:     phase7_output = phase7_task_construction(phase6_output)  
5120:  
5121:     assert not phase7_output.construction_passed  
5122:     assert any("empty string" in err for err in phase7_output.construction_errors)  
5123:  
5124:  
5125: class TestPhase8ExecutionPlanAssembly:  
5126:     """Test Phase 8 execution plan assembly functionality."""  
5127:  
5128:     def test_successful_plan_assembly(self):  
5129:         """Test successful execution plan assembly."""  
5130:         questions = [create_valid_question(i) for i in range(10)]  
5131:         chunks = [create_valid_chunk(i) for i in range(10)]  
5132:  
5133:         phase6_output = phase6_schema_validation(questions, chunks)  
5134:         phase7_output = phase7_task_construction(phase6_output)  
5135:  
5136:         assert phase7_output.construction_passed  
5137:  
5138:         phase8_output = phase8_execution_plan_assembly(phase7_output)  
5139:  
5140:         assert phase8_output.assembly_passed  
5141:         assert phase8_output.execution_plan is not None  
5142:         assert phase8_output.task_count == 10  
5143:         assert len(phase8_output.assembly_errors) == 0  
5144:         assert phase8_output.integrity_hash != ""  
5145:         assert len(phase8_output.integrity_hash) == 64  
5146:         assert phase8_output.phase8_version == PHASE8_VERSION  
5147:  
5148:     def test_phase7_failure_propagation(self):
```

```
5149:     """Test error propagation when Phase 7 fails."""
5150:     phase7_output = Phase7TaskConstructionOutput(
5151:         tasks=[],
5152:         task_count=0,
5153:         construction_passed=False,
5154:         construction_errors=["Phase 7 test error"],
5155:         construction_warnings=[],
5156:         duplicate_task_ids=[],
5157:         missing_fields_by_task={},
5158:         construction_timestamp="2025-01-19T00:00:00Z",
5159:         metadata={}
5160:     )
5161:
5162:     phase8_output = phase8_execution_plan_assembly(phase7_output)
5163:
5164:     assert not phase8_output.assembly_passed
5165:     assert phase8_output.execution_plan is None
5166:     assert len(phase8_output.assembly_errors) > 0
5167:     assert any("Phase 7" in err for err in phase8_output.assembly_errors)
5168:
5169: def test_integrity_hash_determinism(self):
5170:     """Test that integrity hash is deterministic."""
5171:     questions = [create_valid_question(i) for i in range(5)]
5172:     chunks = [create_valid_chunk(i) for i in range(5)]
5173:
5174:     phase6_output = phase6_schema_validation(questions, chunks)
5175:     phase7_output = phase7_task_construction(phase6_output)
5176:
5177:     phase8_output1 = phase8_execution_plan_assembly(phase7_output)
5178:     phase8_output2 = phase8_execution_plan_assembly(phase7_output)
5179:
5180:     assert phase8_output1.integrity_hash == phase8_output2.integrity_hash
5181:
5182: def test_execution_plan_immutability(self):
5183:     """Test that execution plan tasks are immutable tuple."""
5184:     questions = [create_valid_question(i) for i in range(5)]
5185:     chunks = [create_valid_chunk(i) for i in range(5)]
5186:
5187:     phase6_output = phase6_schema_validation(questions, chunks)
5188:     phase7_output = phase7_task_construction(phase6_output)
5189:     phase8_output = phase8_execution_plan_assembly(phase7_output)
5190:
5191:     assert phase8_output.execution_plan is not None
5192:     assert isinstance(phase8_output.execution_plan.tasks, tuple)
5193:
5194:     with pytest.raises(AttributeError):
5195:         phase8_output.execution_plan.tasks.append(None)
5196:
5197:
5198: class TestPhase6_7_8_Integration:
5199:     """Integration tests for the complete Phase 6-7-8 pipeline."""
5200:
5201:     def test_end_to_end_success(self):
5202:         """Test successful end-to-end pipeline execution."""
5203:         questions = [create_valid_question(i) for i in range(20)]
5204:         chunks = [create_valid_chunk(i) for i in range(20)]
```

```
5205:
5206:     phase6_output = phase6_schema_validation(questions, chunks)
5207:     assert phase6_output.schema_validation_passed
5208:
5209:     phase7_output = phase7_task_construction(phase6_output)
5210:     assert phase7_output.construction_passed
5211:
5212:     phase8_output = phase8_execution_plan_assembly(phase7_output)
5213:     assert phase8_output.assembly_passed
5214:     assert phase8_output.execution_plan is not None
5215:
5216:     def test_error_cascade_from_phase6(self):
5217:         """Test that errors cascade correctly from Phase 6 through Phase 8."""
5218:         questions = [{"invalid": "data"}]
5219:         chunks = []
5220:
5221:         phase6_output = phase6_schema_validation(questions, chunks)
5222:         assert not phase6_output.schema_validation_passed
5223:
5224:         phase7_output = phase7_task_construction(phase6_output)
5225:         assert not phase7_output.construction_passed
5226:
5227:         phase8_output = phase8_execution_plan_assembly(phase7_output)
5228:         assert not phase8_output.assembly_passed
5229:         assert phase8_output.execution_plan is None
5230:
5231:     def test_deterministic_processing_order(self):
5232:         """Test that processing order is deterministic."""
5233:         questions = [create_valid_question(i) for i in range(10)]
5234:         chunks = [create_valid_chunk(i) for i in range(10)]
5235:
5236:         run1_phase6 = phase6_schema_validation(questions, chunks)
5237:         run1_phase7 = phase7_task_construction(run1_phase6)
5238:         run1_phase8 = phase8_execution_plan_assembly(run1_phase7)
5239:
5240:         run2_phase6 = phase6_schema_validation(questions, chunks)
5241:         run2_phase7 = phase7_task_construction(run2_phase6)
5242:         run2_phase8 = phase8_execution_plan_assembly(run2_phase7)
5243:
5244:         assert len(run1_phase7.tasks) == len(run2_phase7.tasks)
5245:         for task1, task2 in zip(run1_phase7.tasks, run2_phase7.tasks):
5246:             assert task1.task_id == task2.task_id
5247:             assert task1.question_global == task2.question_global
5248:
5249:         assert run1_phase8.integrity_hash == run2_phase8.integrity_hash
5250:
5251:     def test_metadata_propagation(self):
5252:         """Test that metadata is properly propagated through phases."""
5253:         questions = [create_valid_question(i) for i in range(5)]
5254:         chunks = [create_valid_chunk(i) for i in range(5)]
5255:
5256:         phase6_output = phase6_schema_validation(questions, chunks)
5257:         phase7_output = phase7_task_construction(phase6_output)
5258:         phase8_output = phase8_execution_plan_assembly(phase7_output)
5259:
5260:         assert phase7_output.metadata["question_count"] == 5
```

12/07/25
01:17:36

/Users/recovered/Applications/F.A.R.F.A.N -MECHANISTIC-PIPELINE/code_audit_pdfs/batch_28_combined.txt

95

```
5261:     assert phase7_output.metadata["chunk_count"] == 5
5262:
5263:     assert phase8_output.metadata["task_count"] == 5
5264:     assert "integrity_hash_length" in phase8_output.metadata
5265:
5266:
```