```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 27
================================================================================
Generated: 2025-12-07T06:17:35.012359
Files in this batch: 17
================================================================================


================================================================================
FILE: tests/phases/test_adapter.py
================================================================================

"""Test Phase1â\206\222Phase2 Adapter Contract

Tests PAÃ\227DIM metadata preservation, sentence_metadata.extra structure.
"""
import pytest
from unittest.mock import MagicMock, patch
import hashlib

from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract


class TestAdapterContract:
    """Test adapter contract validation."""

    def test_adapter_preserves_pa_dim_metadata(self):
        """Test adapter preserves policy_area_id and dimension_id in sentence_metadata.extra."""
        from farfan_pipeline.processing.models import (
            CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
            TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest
        )

        contract = AdapterContract()

        chunk_graph = ChunkGraph()
        chunk = Chunk(
            id="test_chunk", text="Test chunk text",
            text_span=TextSpan(0, 50), resolution=ChunkResolution.MESO,
            bytes_hash=hashlib.blake2b(b"test").hexdigest(),
            policy_area_id="PA01", dimension_id="DIM01"
        )
        chunk_graph.add_chunk(chunk)

        cpp = CanonPolicyPackage(
            schema_version="SPC-2025.1", chunk_graph=chunk_graph,
            policy_manifest=PolicyManifest(),
            quality_metrics=QualityMetrics(provenance_completeness=0.9, structural_consistency=0.9),
            integrity_index=IntegrityIndex(blake2b_root="a"*64),
            metadata={"document_id": "test"}
        )

        result = contract.validate_input(cpp)
        assert result.passed

    def test_adapter_output_has_chunked_mode(self):
```

```
        """Test adapter output has processing_mode='chunked'."""
        contract = AdapterContract()

        invariant = next(inv for inv in contract.invariants if inv.name == "processing_mode_chunked")
        assert invariant is not None
        assert "chunked" in invariant.description

    def test_adapter_requires_chunk_id_in_extra(self):
        """Test adapter requires chunk_id in sentence_metadata.extra."""
        contract = AdapterContract()

        invariant = next(inv for inv in contract.invariants if inv.name == "chunk_id_preserved")
        assert invariant is not None
        assert "chunk_id" in invariant.description

    def test_adapter_requires_policy_area_id_in_extra(self):
        """Test adapter requires policy_area_id in sentence_metadata.extra."""
        contract = AdapterContract()

        invariant = next(inv for inv in contract.invariants if inv.name == "policy_area_id_preserved")
        assert invariant is not None
        assert "policy_area_id" in invariant.description

    def test_adapter_requires_dimension_id_in_extra(self):
        """Test adapter requires dimension_id in sentence_metadata.extra."""
        contract = AdapterContract()

        invariant = next(inv for inv in contract.invariants if inv.name == "dimension_id_preserved")
        assert invariant is not None
        assert "dimension_id" in invariant.description


================================================================================
FILE: tests/phases/test_failure_propagation.py
================================================================================

"""Test Failure Propagation (Phase N Failure â\206\222 ABORT)

Tests that phase failures propagate correctly and halt pipeline execution.
"""
import pytest
from unittest.mock import AsyncMock, MagicMock, patch
from datetime import datetime, timezone

from farfan_pipeline.core.phases.phase_protocol import PhaseContract, ContractValidationResult


class TestFailurePropagation:
    """Test phase failure propagation."""

    @pytest.mark.asyncio
    async def test_input_validation_failure_aborts(self):
        """Test input validation failure aborts phase execution."""

        class TestContract(PhaseContract):
```

```
            def validate_input(self, data):
                return ContractValidationResult(
                    passed=False, contract_type="input", phase_name="test",
                    errors=["Input validation failed"]
                )

            def validate_output(self, data):
                return ContractValidationResult(True, "output", "test")

            async def execute(self, data):
                return data

        contract = TestContract("test")

        with pytest.raises(ValueError) as exc_info:
            await contract.run("invalid_input")

        assert "Input contract validation failed" in str(exc_info.value)

    @pytest.mark.asyncio
    async def test_output_validation_failure_aborts(self):
        """Test output validation failure aborts phase execution."""

        class TestContract(PhaseContract):
            def validate_input(self, data):
                return ContractValidationResult(True, "input", "test")

            def validate_output(self, data):
                return ContractValidationResult(
                    passed=False, contract_type="output", phase_name="test",
                    errors=["Output validation failed"]
                )

            async def execute(self, data):
                return "invalid_output"

        contract = TestContract("test")

        with pytest.raises(ValueError) as exc_info:
            await contract.run("input")

        assert "Output contract validation failed" in str(exc_info.value)

    @pytest.mark.asyncio
    async def test_invariant_failure_aborts(self):
        """Test invariant failure aborts phase execution."""

        class TestContract(PhaseContract):
            def __init__(self):
                super().__init__("test")
                self.add_invariant(
                    "test_invariant", "Test invariant",
                    lambda data: False, "Invariant failed"
                )

            def validate_input(self, data):
```

```
                return ContractValidationResult(True, "input", "test")

            def validate_output(self, data):
                return ContractValidationResult(True, "output", "test")

            async def execute(self, data):
                return data

        contract = TestContract()

        with pytest.raises(RuntimeError) as exc_info:
            await contract.run("input")

        assert "Phase invariants failed" in str(exc_info.value)

    @pytest.mark.asyncio
    async def test_execution_error_captured_in_metadata(self):
        """Test execution errors are captured in phase metadata."""

        class TestContract(PhaseContract):
            def validate_input(self, data):
                return ContractValidationResult(True, "input", "test")

            def validate_output(self, data):
                return ContractValidationResult(True, "output", "test")

            async def execute(self, data):
                raise RuntimeError("Execution failed")

        contract = TestContract("test")

        with pytest.raises(RuntimeError):
            await contract.run("input")

        assert contract.metadata is not None
        assert contract.metadata.success is False
        assert "Execution failed" in contract.metadata.error

    @pytest.mark.asyncio
    async def test_phase_failure_prevents_subsequent_phases(self):
        """Test phase failure prevents subsequent phases from executing."""
        from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator
        from pathlib import Path

        orchestrator = PhaseOrchestrator()

        with patch.object(orchestrator.phase0, 'run', side_effect=ValueError("Phase 0 failed")):
            result = await orchestrator.run_pipeline(
                pdf_path=Path("nonexistent.pdf"),
                run_id="test_run",
                questionnaire_path=None
            )

            assert result.success is False
            assert len(result.errors) > 0
            assert result.phases_completed == 0
```

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

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

```
        chunk_hashes = {
            "chunk1": "a" * 128,
            "chunk2": "b" * 128,
            "chunk3": "c" * 128
        }

        combined = "".join(sorted(chunk_hashes.values()))
        root_hash = hashlib.blake2b(combined.encode()).hexdigest()

        integrity = IntegrityIndex(
            blake2b_root=root_hash,
            chunk_hashes=chunk_hashes
        )

        assert len(integrity.blake2b_root) == 128
        assert integrity.blake2b_root == root_hash

    def test_hash_stability_across_multiple_runs(self):
        """Test hash remains stable across multiple computations."""
        content = b"Stable content for multiple hash runs"

        hashes = [hashlib.blake2b(content).hexdigest() for _ in range(10)]

        assert len(set(hashes)) == 1
        assert all(h == hashes[0] for h in hashes)

    def test_empty_content_has_valid_hash(self):
        """Test empty content produces valid hash."""
        hash_empty = hashlib.blake2b(b"").hexdigest()

        assert len(hash_empty) == 128
        assert isinstance(hash_empty, str)

    def test_unicode_content_hash_stability(self):
        """Test hash stability with unicode content."""
        content = "PolÃ-tica de desarrollo con Ã± y tildes: Ã¡Ã©Ã-Ã³Ã°"
        bytes1 = content.encode('utf-8')
        bytes2 = content.encode('utf-8')

        hash1 = hashlib.blake2b(bytes1).hexdigest()
        hash2 = hashlib.blake2b(bytes2).hexdigest()

        assert hash1 == hash2



================================================================================
FILE: tests/phases/test_manifest_builder.py
================================================================================

"""Test Phase Manifest Builder

Tests manifest completeness, phase recording, and artifact tracking.
"""
import pytest
from pathlib import Path
```

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

```
            finished_at="2025-01-01T00:00:01Z",
            duration_ms=500.0,
            success=False,
            error="Test error"
        )

        input_validation = ContractValidationResult(
            passed=False, contract_type="input", phase_name="test_phase",
            errors=["Input validation failed"]
        )

        output_validation = ContractValidationResult(
            passed=False, contract_type="output", phase_name="test_phase"
        )

        builder.record_phase(
            phase_name="test_phase",
            metadata=metadata,
            input_validation=input_validation,
            output_validation=output_validation,
            invariants_checked=[],
            artifacts=[]
        )

        assert builder.phases["test_phase"]["status"] == "failed"
        assert builder.phases["test_phase"]["error"] == "Test error"

    def test_manifest_to_dict(self):
        """Test manifest conversion to dictionary."""
        builder = PhaseManifestBuilder()

        metadata = PhaseMetadata(
            phase_name="phase1", started_at="2025-01-01T00:00:00Z",
            finished_at="2025-01-01T00:00:01Z", success=True
        )

        builder.record_phase(
            "phase1", metadata,
            ContractValidationResult(True, "input", "phase1"),
            ContractValidationResult(True, "output", "phase1"),
            [], []
        )

        manifest_dict = builder.to_dict()
        assert "phases" in manifest_dict
        assert "total_phases" in manifest_dict
        assert "successful_phases" in manifest_dict
        assert "failed_phases" in manifest_dict
        assert manifest_dict["total_phases"] == 1
        assert manifest_dict["successful_phases"] == 1

    def test_manifest_completeness_all_phases(self):
        """Test manifest records all phases (0, 1, adapter, 2)."""
        builder = PhaseManifestBuilder()

        for phase_name in ["phase0_input_validation", "phase1_spc_ingestion",
```

```
                            "phase1_to_phase2_adapter", "phase2_microquestions"]:
            metadata = PhaseMetadata(
                phase_name=phase_name,
                started_at="2025-01-01T00:00:00Z",
                finished_at="2025-01-01T00:00:01Z",
                success=True
            )
            builder.record_phase(
                phase_name, metadata,
                ContractValidationResult(True, "input", phase_name),
                ContractValidationResult(True, "output", phase_name),
                [], []
            )

        manifest_dict = builder.to_dict()
        assert manifest_dict["total_phases"] == 4
        assert all(p in builder.phases for p in [
            "phase0_input_validation", "phase1_spc_ingestion",
            "phase1_to_phase2_adapter", "phase2_microquestions"
        ])

    def test_manifest_tracks_invariants(self):
        """Test manifest tracks which invariants were checked."""
        builder = PhaseManifestBuilder()

        metadata = PhaseMetadata(
            phase_name="test_phase",
            started_at="2025-01-01T00:00:00Z",
            success=True
        )

        invariants = ["validation_passed", "hash_format", "count_positive"]

        builder.record_phase(
            "test_phase", metadata,
            ContractValidationResult(True, "input", "test_phase"),
            ContractValidationResult(True, "output", "test_phase"),
            invariants, []
        )

        assert builder.phases["test_phase"]["invariants_checked"] == invariants
        assert builder.phases["test_phase"]["invariants_satisfied"] is True

    def test_manifest_save(self, tmp_path):
        """Test manifest can be saved to file."""
        builder = PhaseManifestBuilder()

        metadata = PhaseMetadata(
            phase_name="test_phase",
            started_at="2025-01-01T00:00:00Z",
            success=True
        )

        builder.record_phase(
            "test_phase", metadata,
            ContractValidationResult(True, "input", "test_phase"),
```

```
            ContractValidationResult(True, "output", "test_phase"),
            [], []
        )

        manifest_path = tmp_path / "manifest.json"
        builder.save(manifest_path)

        assert manifest_path.exists()
        import json
        with open(manifest_path) as f:
            saved = json.load(f)
        assert "phases" in saved
        assert "test_phase" in saved["phases"]


================================================================================
FILE: tests/phases/test_orchestrator_integration.py
================================================================================

"""Test Phase Orchestrator Integration

Tests full pipeline execution through orchestrator with all phases.
"""
import pytest
from pathlib import Path
from unittest.mock import AsyncMock, patch, MagicMock


class TestOrchestratorIntegration:
    """Test orchestrator executes all phases in sequence."""

    def test_orchestrator_has_all_phase_contracts(self):
        """Test orchestrator has phase0, phase1, adapter contracts."""
        from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator

        orchestrator = PhaseOrchestrator()

        assert hasattr(orchestrator, 'phase0')
        assert hasattr(orchestrator, 'phase1')
        assert hasattr(orchestrator, 'adapter')
        assert hasattr(orchestrator, 'manifest_builder')

    def test_orchestrator_manifest_builder_initialized(self):
        """Test orchestrator initializes manifest builder."""
        from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator

        orchestrator = PhaseOrchestrator()

        assert orchestrator.manifest_builder is not None
        assert len(orchestrator.manifest_builder.phases) == 0

    @pytest.mark.asyncio
    async def test_orchestrator_pipeline_result_structure(self):
        """Test PipelineResult has required fields."""
        from farfan_pipeline.core.phases.phase_orchestrator import PipelineResult
```

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

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

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

```
class TestPhase0InputContract:
    """Test Phase 0 input contract validation."""

    def test_phase0_input_valid(self):
        """Test valid Phase0Input passes validation."""
        contract = Phase0ValidationContract()
        input_data = Phase0Input(
            pdf_path=Path("test.pdf"),
            run_id="test_run_001",
            questionnaire_path=Path("questions.json"),
        )

        result = contract.validate_input(input_data)
        assert result.passed
        assert len(result.errors) == 0

    def test_phase0_input_invalid_type(self):
        """Test invalid input type fails validation."""
        contract = Phase0ValidationContract()
        input_data = {"pdf_path": "test.pdf"}

        result = contract.validate_input(input_data)
        assert not result.passed
        assert len(result.errors) > 0
        assert "Phase0Input" in result.errors[0]

    def test_phase0_input_empty_run_id(self):
        """Test empty run_id fails validation."""
        contract = Phase0ValidationContract()
        input_data = Phase0Input(
            pdf_path=Path("test.pdf"),
            run_id="",
            questionnaire_path=None,
        )

        result = contract.validate_input(input_data)
        assert not result.passed

    def test_phase0_input_invalid_run_id_chars(self):
        """Test run_id with invalid filesystem chars fails validation."""
        contract = Phase0ValidationContract()
        input_data = Phase0Input(
            pdf_path=Path("test.pdf"),
            run_id="test/run:001",
            questionnaire_path=None,
        )

        result = contract.validate_input(input_data)
        assert not result.passed


class TestPhase0OutputContract:
    """Test Phase 0 output contract validation."""

    def test_canonical_input_valid(self, tmp_path):
        """Test valid CanonicalInput passes validation."""
```

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

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

```
            run_id="test_run_001",
            questionnaire_path=tmp_path / "questions.json",
        )

        with pytest.raises(FileNotFoundError) as exc_info:
            await contract.execute(input_data)

        assert "PDF not found" in str(exc_info.value)

    @pytest.mark.asyncio
    async def test_execute_missing_questionnaire(self, tmp_path):
        """Test execution fails for missing questionnaire."""
        pdf_path = tmp_path / "test.pdf"
        pdf_path.write_bytes(b"%PDF-1.4\n")

        contract = Phase0ValidationContract()
        input_data = Phase0Input(
            pdf_path=pdf_path,
            run_id="test_run_001",
            questionnaire_path=tmp_path / "nonexistent.json",
        )

        with pytest.raises(FileNotFoundError) as exc_info:
            await contract.execute(input_data)

        assert "Questionnaire not found" in str(exc_info.value)

    @pytest.mark.asyncio
    async def test_execute_success(self, tmp_path):
        """Test successful execution produces valid CanonicalInput."""
        pdf_path = tmp_path / "test.pdf"
        pdf_content = b"%PDF-1.4\n%Test PDF content"
        pdf_path.write_bytes(pdf_content)

        questionnaire_path = tmp_path / "questions.json"
        questionnaire_path.write_text('{"questions": []}')

        contract = Phase0ValidationContract()

        with patch.object(contract, "_get_pdf_page_count", return_value=5):
            input_data = Phase0Input(
                pdf_path=pdf_path,
                run_id="test_run_001",
                questionnaire_path=questionnaire_path,
            )

            canonical_input = await contract.execute(input_data)

            assert canonical_input.document_id == "test"
            assert canonical_input.run_id == "test_run_001"
            assert canonical_input.pdf_path == pdf_path
            assert len(canonical_input.pdf_sha256) == 64
            assert canonical_input.pdf_size_bytes == len(pdf_content)
            assert canonical_input.pdf_page_count == 5
            assert canonical_input.validation_passed is True
            assert len(canonical_input.validation_errors) == 0
```

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

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

```
        assert len(failed) > 0


class TestPhase0Run:
    """Test Phase 0 run() integration."""

    @pytest.mark.asyncio
    async def test_run_validates_input(self, tmp_path):
        """Test run() validates input contract."""
        contract = Phase0ValidationContract()
        invalid_input = "not a Phase0Input"

        with pytest.raises(ValueError) as exc_info:
            await contract.run(invalid_input)

        assert "Input contract validation failed" in str(exc_info.value)

    @pytest.mark.asyncio
    async def test_run_returns_metadata(self, tmp_path):
        """Test run() returns phase metadata."""
        pdf_path = tmp_path / "test.pdf"
        pdf_path.write_bytes(b"%PDF-1.4\n")
        questionnaire_path = tmp_path / "questions.json"
        questionnaire_path.write_text('{}')

        contract = Phase0ValidationContract()

        with patch.object(contract, "_get_pdf_page_count", return_value=3):
            input_data = Phase0Input(
                pdf_path=pdf_path,
                run_id="test_run",
                questionnaire_path=questionnaire_path,
            )

            output, metadata = await contract.run(input_data)

            assert metadata.phase_name == "phase0_input_validation"
            assert metadata.success is True
            assert metadata.error is None
            assert metadata.duration_ms is not None
            assert metadata.duration_ms >= 0


================================================================================
FILE: tests/phases/test_phase1_spc_ingestion.py
================================================================================

"""Test Phase 1: SPC Ingestion Contract

Tests 60 chunks, PAÃ\227DIM tagging, quality thresholds, hash stability.
"""
import hashlib
import pytest
from unittest.mock import AsyncMock, patch
from datetime import datetime, timezone
```

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

```
            policy_manifest=PolicyManifest(),
            quality_metrics=QualityMetrics(provenance_completeness=0.9, structural_consistency=0.9),
            integrity_index=IntegrityIndex(blake2b_root="a"*64),
            metadata={"document_id": "test"}
        )

        result = contract.validate_output(cpp)
        assert result.passed

    def test_provenance_threshold_enforced(self):
        """Test provenance_completeness >= 0.8 enforced."""
        from farfan_pipeline.processing.models import (
            CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
            TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest
        )
        contract = Phase1SPCIngestionContract()

        chunk_graph = ChunkGraph()
        for i, pa in enumerate(POLICY_AREAS):
            for j, dim in enumerate(DIMENSIONS):
                chunk = Chunk(
                    id=f"c_{pa}_{dim}", text="t",
                    text_span=TextSpan(0, 1), resolution=ChunkResolution.MESO,
                    bytes_hash=hashlib.blake2b(f"t{i}{j}".encode()).hexdigest(),
                    policy_area_id=pa, dimension_id=dim
                )
                chunk_graph.add_chunk(chunk)

        cpp = CanonPolicyPackage(
            schema_version="SPC-2025.1", chunk_graph=chunk_graph,
            policy_manifest=PolicyManifest(),
            quality_metrics=QualityMetrics(provenance_completeness=0.5, structural_consistency=0.9),
            integrity_index=IntegrityIndex(blake2b_root="a"*64),
            metadata={"document_id": "test"}
        )

        result = contract.validate_output(cpp)
        assert not result.passed


class TestPhase1HashStability:
    """Test BLAKE2b hash stability."""

    def test_blake2b_deterministic(self):
        """Test BLAKE2b produces same hash for same content."""
        content = b"test chunk content"
        hash1 = hashlib.blake2b(content).hexdigest()
        hash2 = hashlib.blake2b(content).hexdigest()
        assert hash1 == hash2
        assert len(hash1) == 128


================================================================================
FILE: tests/phases/test_phase2_executor_context_preparation.py
================================================================================
```

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

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

```
        assert task.policy_area_id.startswith("PA")
        assert len(task.policy_area_id) == POLICY_AREA_ID_LENGTH

    def test_extract_dim_id_from_question(self, sample_question):
        """Test dimension ID extraction."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert task.dimension_id == "DIM01"
        assert task.dimension_id.startswith("DIM")
        assert len(task.dimension_id) == DIMENSION_ID_LENGTH

    def test_extract_question_global_from_question(self, sample_question):
        """Test question_global extraction."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert task.question_global == 1
        assert isinstance(task.question_global, int)
        assert 1 <= task.question_global <= TOTAL_QUESTIONS

    def test_extract_question_id_from_question(self, sample_question):
        """Test question_id extraction."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert task.question_id == "MICRO_001"
        assert isinstance(task.question_id, str)

    def test_task_id_format_consistency(self, sample_question):
        """Test task_id follows MQC-{question_global:03d}_{policy_area_id} format."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert task.task_id == "MQC-001_PA01"
        assert task.task_id.startswith("MQC-")
```

```
        assert "_PA" in task.task_id

    def test_routing_keys_all_dimensions(self):
        """Test routing key extraction for all 6 dimensions."""
        for dim_num in range(1, 7):
            question = {
                "question_id": f"MICRO_{dim_num:03d}",
                "question_global": dim_num,
                "dimension_id": f"DIM0{dim_num}",
                "policy_area_id": "PA01",
                "expected_elements": [],
            }

            task = _construct_task(
                question=question,
                chunk={"id": "CHUNK_001", "expected_elements": []},
                patterns=[],
                signals={},
                generated_ids=set(),
            )

            assert task.dimension_id == f"DIM0{dim_num}"

    def test_routing_keys_all_policy_areas(self):
        """Test routing key extraction for all 10 policy areas."""
        for pa_num in range(1, 11):
            question = {
                "question_id": f"MICRO_{pa_num:03d}",
                "question_global": pa_num,
                "dimension_id": "DIM01",
                "policy_area_id": f"PA{pa_num:02d}",
                "expected_elements": [],
            }

            task = _construct_task(
                question=question,
                chunk={"id": "CHUNK_001", "expected_elements": []},
                patterns=[],
                signals={},
                generated_ids=set(),
            )

            assert task.policy_area_id == f"PA{pa_num:02d}"

    def test_chunk_id_extraction(self, sample_question):
        """Test chunk_id extraction from chunk metadata."""
        chunk = {"id": "CHUNK_042", "expected_elements": []}

        task = _construct_task(
            question=sample_question,
            chunk=chunk,
            patterns=[],
            signals={},
            generated_ids=set(),
        )

```

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

```
            patterns=sample_patterns,
            signals={},
            generated_ids=set(),
        )

        assert len(task.patterns) == 2
        assert task.patterns[0]["pattern_id"] == "PAT_001"
        assert task.patterns[1]["pattern_type"] == "gap_analysis"

    def test_metadata_cluster_id(self, sample_question):
        """Test cluster_id metadata extraction."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert task.metadata["cluster_id"] == "C01"

    def test_metadata_base_slot(self, sample_question):
        """Test base_slot metadata extraction."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert task.metadata["base_slot"] == "D1-Q1"

    def test_creation_timestamp_format(self, sample_question):
        """Test creation_timestamp is in ISO format."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        assert isinstance(task.creation_timestamp, str)
        assert "T" in task.creation_timestamp
        assert "Z" in task.creation_timestamp or "+" in task.creation_timestamp


class TestLoggingOutputStructure:
    """Test logging output structure and traceability."""

    def test_task_serialization_to_dict(self, sample_question):
        """Test ExecutableTask can be serialized to dict."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
```

```
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        task_dict = asdict(task)

        assert isinstance(task_dict, dict)
        assert "task_id" in task_dict
        assert "question_id" in task_dict
        assert "policy_area_id" in task_dict

    def test_task_serialization_to_json(self, sample_question):
        """Test ExecutableTask can be serialized to JSON."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        task_dict = asdict(task)
        json_str = json.dumps(task_dict)

        assert isinstance(json_str, str)
        parsed = json.loads(json_str)
        assert parsed["task_id"] == task.task_id

    def test_logging_contains_all_routing_keys(self, sample_question):
        """Test logging output includes all routing keys."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals={},
            generated_ids=set(),
        )

        task_dict = asdict(task)

        assert "policy_area_id" in task_dict
        assert "dimension_id" in task_dict
        assert "question_global" in task_dict
        assert "question_id" in task_dict

    def test_logging_contains_execution_metadata(self, sample_question, sample_signals):
        """Test logging output includes execution metadata."""
        task = _construct_task(
            question=sample_question,
            chunk={"id": "CHUNK_001", "expected_elements": []},
            patterns=[],
            signals=sample_signals,
            generated_ids=set(),
        )

```

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

```
            for i in range(1, 13)
        ]

        tasks = [
            _construct_task(
                question=q,
                chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
                patterns=[],
                signals={},
                generated_ids=set(),
            )
            for i, q in enumerate(questions, 1)
        ]

        sorted_tasks = sorted(tasks, key=lambda t: (t.dimension_id, t.policy_area_id))

        for i in range(len(sorted_tasks) - 1):
            assert sorted_tasks[i].dimension_id <= sorted_tasks[i + 1].dimension_id

    def test_policy_area_ordering_within_dimension(self):
        """Test tasks are ordered by policy area within each dimension."""
        questions = [
            {
                "question_id": f"Q{i:03d}",
                "question_global": i,
                "dimension_id": "DIM01",
                "policy_area_id": f"PA{(i % 10) + 1:02d}",
                "expected_elements": [],
            }
            for i in range(1, 21)
        ]

        tasks = [
            _construct_task(
                question=q,
                chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
                patterns=[],
                signals={},
                generated_ids=set(),
            )
            for i, q in enumerate(questions, 1)
        ]

        dim01_tasks = [t for t in tasks if t.dimension_id == "DIM01"]
        sorted_tasks = sorted(dim01_tasks, key=lambda t: t.policy_area_id)

        for i in range(len(sorted_tasks) - 1):
            assert sorted_tasks[i].policy_area_id <= sorted_tasks[i + 1].policy_area_id

    def test_ordering_deterministic_across_runs(self):
        """Test task ordering is deterministic across multiple runs."""
        questions = [
            {
                "question_id": f"Q{i:03d}",
                "question_global": i,
                "dimension_id": f"DIM0{((i - 1) // 50) + 1}",
```

```
                "policy_area_id": f"PA{((i - 1) % 10) + 1:02d}",
                "expected_elements": [],
            }
            for i in range(1, 31)
        ]

        task_ids_runs = []
        for _run in range(5):
            tasks = [
                _construct_task(
                    question=q,
                    chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
                    patterns=[],
                    signals={},
                    generated_ids=set(),
                )
                for i, q in enumerate(questions, 1)
            ]

            sorted_tasks = sorted(
                tasks, key=lambda t: (t.dimension_id, t.policy_area_id, t.question_id)
            )
            task_ids = [t.task_id for t in sorted_tasks]
            task_ids_runs.append(task_ids)

        first_run = task_ids_runs[0]
        for subsequent_run in task_ids_runs[1:]:
            assert first_run == subsequent_run

    def test_300_questions_ordering(self):
        """Test all 300 questions maintain deterministic ordering."""
        questions = [
            {
                "question_id": f"MICRO_{i:03d}",
                "question_global": i,
                "dimension_id": f"DIM{((i - 1) // 50) + 1:02d}",
                "policy_area_id": f"PA{((i - 1) % 10) + 1:02d}",
                "expected_elements": [],
            }
            for i in range(1, 301)
        ]

        generated_ids = set()
        tasks = [
            _construct_task(
                question=q,
                chunk={"id": f"CHUNK_{i:03d}", "expected_elements": []},
                patterns=[],
                signals={},
                generated_ids=generated_ids,
            )
            for i, q in enumerate(questions, 1)
        ]

        assert len(tasks) == TOTAL_QUESTIONS

```

```
        sorted_tasks = sorted(
            tasks, key=lambda t: (t.dimension_id, t.policy_area_id, t.question_global)
        )

        for i in range(len(sorted_tasks) - 1):
            if sorted_tasks[i].dimension_id == sorted_tasks[i + 1].dimension_id:
                assert (
                    sorted_tasks[i].policy_area_id <= sorted_tasks[i + 1].policy_area_id
                )


class TestCrossTaskValidation:
    """Test cross-task validation and consistency checks."""

    def test_chunk_usage_validation(self):
        """Test chunk usage validation (5 tasks per chunk expected)."""
        tasks = []
        for i in range(1, 6):
            task = ExecutableTask(
                task_id=f"MQC-{i:03d}_PA01",
                question_id=f"Q{i:03d}",
                question_global=i,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="CHUNK_001",
                patterns=[],
                signals={},
                creation_timestamp="2025-01-01T00:00:00Z",
            )
            tasks.append(task)

        with patch(
            "farfan_pipeline.core.orchestrator.task_planner.logger"
        ) as mock_logger:
            _validate_cross_task(tasks)

        chunk_warnings = [
            call
            for call in mock_logger.warning.call_args_list
            if "Chunk usage deviation" in str(call)
        ]
        assert len(chunk_warnings) == 0

    def test_chunk_usage_deviation_warning(self):
        """Test warning when chunk usage deviates from expected."""
        tasks = []
        for i in range(1, 4):
            task = ExecutableTask(
                task_id=f"MQC-{i:03d}_PA01",
                question_id=f"Q{i:03d}",
                question_global=i,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="CHUNK_001",
                patterns=[],
                signals={},
```

```
                creation_timestamp="2025-01-01T00:00:00Z",
            )
            tasks.append(task)

        with patch(
            "farfan_pipeline.core.orchestrator.task_planner.logger"
        ) as mock_logger:
            _validate_cross_task(tasks)

        mock_logger.warning.assert_called()
        chunk_warnings = [
            call[0][0]
            for call in mock_logger.warning.call_args_list
            if "Chunk usage deviation" in call[0][0]
        ]
        assert len(chunk_warnings) > 0
        assert "CHUNK_001" in chunk_warnings[0]

    def test_policy_area_usage_validation(self):
        """Test policy area usage validation (30 tasks per PA expected)."""
        tasks = []
        for i in range(1, 31):
            task = ExecutableTask(
                task_id=f"MQC-{i:03d}_PA01",
                question_id=f"Q{i:03d}",
                question_global=i,
                policy_area_id="PA01",
                dimension_id=f"DIM{((i - 1) // 5) + 1:02d}",
                chunk_id=f"CHUNK_{i:03d}",
                patterns=[],
                signals={},
                creation_timestamp="2025-01-01T00:00:00Z",
            )
            tasks.append(task)

        with patch(
            "farfan_pipeline.core.orchestrator.task_planner.logger"
        ) as mock_logger:
            _validate_cross_task(tasks)

        pa_warnings = [
            call
            for call in mock_logger.warning.call_args_list
            if "Policy area usage deviation" in str(call)
        ]
        assert len(pa_warnings) == 0

    def test_policy_area_usage_deviation_warning(self):
        """Test warning when policy area usage deviates from expected."""
        tasks = []
        for i in range(1, 16):
            task = ExecutableTask(
                task_id=f"MQC-{i:03d}_PA01",
                question_id=f"Q{i:03d}",
                question_global=i,
                policy_area_id="PA01",
```

```
                dimension_id="DIM01",
                chunk_id=f"CHUNK_{i:03d}",
                patterns=[],
                signals={},
                creation_timestamp="2025-01-01T00:00:00Z",
            )
            tasks.append(task)

        with patch(
            "farfan_pipeline.core.orchestrator.task_planner.logger"
        ) as mock_logger:
            _validate_cross_task(tasks)

        mock_logger.warning.assert_called()
        warning_call = mock_logger.warning.call_args[0][0]
        assert "Policy area usage deviation" in warning_call


class TestPhase2IntegrationScenarios:
    """Test complete Phase 2 integration scenarios."""

    def test_complete_task_construction_pipeline(
        self, sample_question, sample_chunk, sample_patterns, sample_signals
    ):
        """Test complete task construction with all components."""
        task = _construct_task(
            question=sample_question,
            chunk=sample_chunk,
            patterns=sample_patterns,
            signals=sample_signals,
            generated_ids=set(),
        )

        assert task.task_id == "MQC-001_PA01"
        assert task.question_id == "MICRO_001"
        assert task.policy_area_id == "PA01"
        assert task.dimension_id == "DIM01"
        assert len(task.patterns) == 2
        assert "required_signals" in task.signals
        assert len(task.expected_elements) == 2

    def test_dimension_scoped_execution_plan(self):
        """Test execution plan for single dimension (50 questions)."""
        questions = [
            {
                "question_id": f"MICRO_{i:03d}",
                "question_global": i,
                "dimension_id": "DIM01",
                "policy_area_id": f"PA{((i - 1) % 10) + 1:02d}",
                "expected_elements": [],
            }
            for i in range(1, 51)
        ]

        generated_ids = set()
        tasks = [
```

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

```
            for pa in range(1, 11):
                question = {
                    "question_id": f"Q_D{dim}_PA{pa:02d}",
                    "question_global": (dim - 1) * 10 + pa,
                    "dimension_id": f"DIM{dim:02d}",
                    "policy_area_id": f"PA{pa:02d}",
                    "expected_elements": [],
                }

                task = _construct_task(
                    question=question,
                    chunk={"id": f"CHUNK_D{dim}_PA{pa:02d}", "expected_elements": []},
                    patterns=[],
                    signals={},
                    generated_ids=set(),
                )

                key = (task.dimension_id, task.policy_area_id)
                if key not in tasks_matrix:
                    tasks_matrix[key] = []
                tasks_matrix[key].append(task)

        assert len(tasks_matrix) == PA_DIM_COMBINATIONS

        for (dim_id, pa_id), task_list in tasks_matrix.items():
            for task in task_list:
                assert task.dimension_id == dim_id
                assert task.policy_area_id == pa_id


================================================================================
FILE: tests/phases/test_phase2_microquestions.py
================================================================================

"""Test Phase 2: Microquestions Execution

Tests Phase 2 question generation and validation.
"""
import pytest
from farfan_pipeline.core.phases.phase2_types import (
    Phase2QuestionResult, Phase2Result, validate_phase2_result
)


class TestPhase2Contract:
    """Test Phase 2 result validation."""

    def test_phase2_result_structure(self):
        """Test Phase2Result has questions list."""
        question = Phase2QuestionResult(
            base_slot="D1Q1", question_id="q1", question_global=1,
            policy_area_id="PA01", dimension_id="DIM01",
            evidence={}, validation={}, trace={}
        )
        result = Phase2Result(questions=[question])
```

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

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

```
        key = (policy_area, dimension_id)
        assert key == ("PA01", "DIM01")
        assert isinstance(key, tuple)
        assert len(key) == 2


class TestPhase3ChunkMatrixLookup:
    """Test chunk_matrix.get_chunk() success path with valid keys."""

    @pytest.fixture
    def valid_chunk(self):
        """Create valid chunk with proper PA-DIM structure."""
        return ChunkData(
            id=0,
            text="Test chunk content with sufficient length for validation",
            chunk_type="diagnostic",
            sentences=[0, 1],
            tables=[],
            start_pos=0,
            end_pos=100,
            confidence=0.95,
            chunk_id="PA01-DIM01",
            policy_area_id="PA01",
            dimension_id="DIM01",
            provenance=Provenance(page_number=1, section_header="Introduction"),
        )

    @pytest.fixture
    def preprocessed_document_with_chunks(self, valid_chunk):
        """Create preprocessed document with 60 valid chunks."""
        chunks = []
        chunk_id = 0
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunk = replace(
                    valid_chunk,
                    id=chunk_id,
                    chunk_id=f"{pa_id}-{dim_id}",
                    policy_area_id=pa_id,
                    dimension_id=dim_id,
                    text=f"Content for {pa_id} {dim_id}",
                )
                chunks.append(chunk)
                chunk_id += 1

        return PreprocessedDocument(
            document_id="test_doc",
            raw_text="Test document content",
            sentences=[{"text": "Sentence 1"}],
            tables=[],
            metadata={"test": True, "chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )
```

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

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

```
        )

        matrix = ChunkMatrix(doc)

        with pytest.raises(KeyError, match=r"PA99-DIM99"):
            matrix.get_chunk("PA99", "DIM99")

    def test_valueerror_propagation_from_keyerror(self):
        """Test that synchronization failures propagate ValueError from KeyError."""
        chunks = []
        for i in range(1, 61):
            pa_id = f"PA{((i - 1) // 6) + 1:02d}"
            dim_id = f"DIM{((i - 1) % 6) + 1:02d}"
            chunk = ChunkData(
                id=i - 1,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[0],
                tables=[],
                start_pos=0,
                end_pos=50,
                confidence=0.9,
                policy_area_id=pa_id,
                dimension_id=dim_id,
            )
            chunks.append(chunk)

        doc = PreprocessedDocument(
            document_id="test",
            raw_text="Test",
            sentences=[],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

        matrix = ChunkMatrix(doc)

        try:
            matrix.get_chunk("PA11", "DIM01")
            pytest.fail("Expected KeyError to be raised")
        except KeyError as e:
            error_msg = str(e)
            assert "PA11" in error_msg or "DIM01" in error_msg


class TestPhase3MultiStageVerification:
    """Test multi-stage verification checks for PA/DIM/chunk_id consistency."""

    @pytest.fixture
    def document_with_full_metadata(self):
        """Create document with complete chunk metadata."""
        chunks = []
        for i in range(60):
            pa_num = (i // 6) + 1
```

```
            dim_num = (i % 6) + 1
            pa_id = f"PA{pa_num:02d}"
            dim_id = f"DIM{dim_num:02d}"

            chunk = ChunkData(
                id=i,
                text=f"Chunk content for {pa_id}-{dim_id}",
                chunk_type="diagnostic",
                sentences=[i],
                tables=[],
                start_pos=i * 100,
                end_pos=(i + 1) * 100,
                confidence=0.95,
                chunk_id=f"{pa_id}-{dim_id}",
                policy_area_id=pa_id,
                dimension_id=dim_id,
            )
            chunks.append(chunk)

        return PreprocessedDocument(
            document_id="full_meta",
            raw_text="Full metadata document",
            sentences=[{"text": f"Sentence {i}"} for i in range(60)],
            tables=[],
            metadata={"chunk_count": 60, "has_full_metadata": True},
            chunks=chunks,
            processing_mode="chunked",
        )

    def test_policy_area_id_match_verification(self, document_with_full_metadata):
        """Test policy_area_id matches between request and chunk."""
        matrix = ChunkMatrix(document_with_full_metadata)

        chunk = matrix.get_chunk("PA03", "DIM02")

        assert chunk.policy_area_id == "PA03"

    def test_dimension_id_match_verification(self, document_with_full_metadata):
        """Test dimension_id matches between request and chunk."""
        matrix = ChunkMatrix(document_with_full_metadata)

        chunk = matrix.get_chunk("PA07", "DIM05")

        assert chunk.dimension_id == "DIM05"

    def test_chunk_id_consistency_with_pa_dim(self, document_with_full_metadata):
        """Test chunk_id is consistent with policy_area_id-dimension_id."""
        matrix = ChunkMatrix(document_with_full_metadata)

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

                chunk = matrix.get_chunk(pa_id, dim_id)

```

```
                expected_chunk_id = f"{pa_id}-{dim_id}"
                assert chunk.chunk_id == expected_chunk_id
                assert chunk.policy_area_id == pa_id
                assert chunk.dimension_id == dim_id

    def test_chunk_id_derivation_from_pa_dim(self):
        """Test chunk_id is automatically derived from PA and DIM if not provided."""
        chunk = ChunkData(
            id=0,
            text="Test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=10,
            confidence=0.9,
            policy_area_id="PA02",
            dimension_id="DIM03",
        )

        assert chunk.chunk_id == "PA02-DIM03"


class TestPhase3ChunkPayloadExtraction:
    """Test chunk payload extraction with text, expected_elements, document_position."""

    @pytest.fixture
    def chunk_with_provenance(self):
        """Create chunk with complete provenance metadata."""
        return ChunkData(
            id=42,
            text="This is a complete chunk with sufficient content for testing payload extraction",
            chunk_type="activity",
            sentences=[10, 11, 12],
            tables=[2],
            start_pos=1000,
            end_pos=1500,
            confidence=0.98,
            chunk_id="PA05-DIM04",
            policy_area_id="PA05",
            dimension_id="DIM04",
            provenance=Provenance(
                page_number=5,
                section_header="Implementation Strategy",
                bbox=(100.0, 200.0, 400.0, 500.0),
                span_in_page=(50, 150),
                source_file="test_plan.pdf",
            ),
        )

    def test_text_extraction_non_empty_validation(self, chunk_with_provenance):
        """Test chunk text is non-empty and properly extracted."""
        assert chunk_with_provenance.text
        assert len(chunk_with_provenance.text) > 0
        assert isinstance(chunk_with_provenance.text, str)

```

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

```
        )

        assert chunk.provenance is None

    def test_provenance_metadata_extraction(self, chunk_with_provenance):
        """Test extraction of provenance metadata fields."""
        prov = chunk_with_provenance.provenance

        assert prov is not None
        assert prov.page_number == 5
        assert prov.section_header == "Implementation Strategy"
        assert prov.bbox == (100.0, 200.0, 400.0, 500.0)
        assert prov.span_in_page == (50, 150)
        assert prov.source_file == "test_plan.pdf"


class TestPhase3SynchronizationFailures:
    """Test ValueError raising with correct error message format on sync failures."""

    def test_missing_chunk_synchronization_error_format(self):
        """Test ValueError message format when chunk synchronization fails."""
        chunks = []
        for i in range(59):
            pa_id = f"PA{((i) // 6) + 1:02d}"
            dim_id = f"DIM{((i) % 6) + 1:02d}"
            chunk = ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id=pa_id,
                dimension_id=dim_id,
            )
            chunks.append(chunk)

        doc = PreprocessedDocument(
            document_id="incomplete",
            raw_text="Test",
            sentences=[],
            tables=[],
            metadata={"chunk_count": 59},
            chunks=chunks,
            processing_mode="chunked",
        )

        with pytest.raises(ValueError, match="Missing chunk combinations"):
            ChunkMatrix(doc)

    def test_duplicate_chunk_synchronization_error(self):
        """Test ValueError when duplicate PA-DIM combinations exist."""
        chunks = []
        for i in range(59):
```

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

```
                dimension_id="DIM01",
            )

    def test_invalid_dimension_format_error(self):
        """Test ValueError for invalid dimension_id format."""
        with pytest.raises(ValueError, match="Invalid chunk_id"):
            ChunkData(
                id=0,
                text="Test",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id="PA01",
                dimension_id="INVALID",
            )


class TestPhase3StructuredLogging:
    """Test structured logging output validation with required fields."""

    @pytest.fixture
    def valid_60_chunk_document(self):
        """Create valid document with 60 chunks for logging tests."""
        chunks = []
        for i in range(60):
            pa_id = f"PA{(i // 6) + 1:02d}"
            dim_id = f"DIM{(i % 6) + 1:02d}"
            chunk = ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[i],
                tables=[],
                start_pos=i * 100,
                end_pos=(i + 1) * 100,
                confidence=0.9,
                policy_area_id=pa_id,
                dimension_id=dim_id,
            )
            chunks.append(chunk)

        return PreprocessedDocument(
            document_id="logging_test",
            raw_text="Logging test document",
            sentences=[{"text": f"S{i}"} for i in range(60)],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

    def test_chunk_matrix_construction_logging(
        self, valid_60_chunk_document, caplog
```

```
    ):
        """Test chunk matrix construction emits structured log events."""
        with caplog.at_level(logging.INFO):
            ChunkMatrix(valid_60_chunk_document)

        log_records = [r for r in caplog.records if "chunk_matrix" in r.message.lower()]
        assert len(log_records) > 0

    def test_log_includes_chunk_count_field(
        self, valid_60_chunk_document, caplog
    ):
        """Test logs include chunk count information."""
        with caplog.at_level(logging.INFO):
            ChunkMatrix(valid_60_chunk_document)

        assert any(
            "60" in record.message or "chunk" in record.message.lower()
            for record in caplog.records
        )

    def test_chunk_routing_correlation_id_propagation(self):
        """Test correlation_id propagates through chunk routing operations."""
        from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
            IrrigationSynchronizer,
        )

        chunks = []
        for i in range(60):
            pa_id = f"PA{(i // 6) + 1:02d}"
            dim_id = f"DIM{(i % 6) + 1:02d}"
            chunk = ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id=pa_id,
                dimension_id=dim_id,
            )
            chunks.append(chunk)

        doc = PreprocessedDocument(
            document_id="test",
            raw_text="Test",
            sentences=[],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

        questionnaire = {
            "blocks": {
```

```
                "D1_Q01": {
                    "question": "Test question?",
                    "patterns": [],
                }
            }
        }

        synchronizer = IrrigationSynchronizer(
            questionnaire=questionnaire,
            preprocessed_document=doc,
        )

        assert hasattr(synchronizer, "correlation_id")
        assert synchronizer.correlation_id is not None
        assert len(synchronizer.correlation_id) > 0

    def test_log_event_field_present_in_structured_logs(
        self, valid_60_chunk_document, caplog
    ):
        """Test 'event' field is present in structured log output."""
        with caplog.at_level(logging.INFO):
            ChunkMatrix(valid_60_chunk_document)

        log_messages = [r.message for r in caplog.records]
        has_event_field = any("event" in msg or "Event" in msg for msg in log_messages)
        assert has_event_field or len(caplog.records) > 0

    def test_chunk_id_in_logging_context(self, valid_60_chunk_document):
        """Test chunk_id appears in logging context for operations."""
        matrix = ChunkMatrix(valid_60_chunk_document)

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

        assert chunk.chunk_id is not None
        assert chunk.chunk_id == "PA01-DIM01"

    def test_question_id_context_in_task_generation(self):
        """Test question_id context is available for logging in task generation."""
        from farfan_pipeline.core.orchestrator.irrigation_synchronizer import (
            IrrigationSynchronizer,
        )

        chunks = []
        for i in range(60):
            pa_id = f"PA{(i // 6) + 1:02d}"
            dim_id = f"DIM{(i % 6) + 1:02d}"
            chunk = ChunkData(
                id=i,
                text=f"Chunk {i}",
                chunk_type="diagnostic",
                sentences=[],
                tables=[],
                start_pos=0,
                end_pos=10,
                confidence=0.9,
                policy_area_id=pa_id,
```

```
                dimension_id=dim_id,
            )
            chunks.append(chunk)

        doc = PreprocessedDocument(
            document_id="test",
            raw_text="Test",
            sentences=[],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

        questionnaire = {
            "blocks": {
                "D1_Q01": {
                    "question": "Test question?",
                    "patterns": [],
                }
            }
        }

        synchronizer = IrrigationSynchronizer(
            questionnaire=questionnaire,
            preprocessed_document=doc,
        )

        plan = synchronizer.build_execution_plan()

        assert len(plan.tasks) > 0
        first_task = plan.tasks[0]
        assert hasattr(first_task, "question_id")
        assert first_task.question_id is not None


================================================================================
FILE: tests/phases/test_phase3_implementation.py
================================================================================

"""
Test Phase 3: Chunk Routing Implementation

This test file verifies the Phase 3 implementation against the specification,
including:
- Routing logic with strict PA-DIM enforcement
- ChunkRoutingResult construction with all 7 canonical fields
- ValueError exceptions with descriptive messages
- Phase specification hierarchical structure compliance
- Observability logging without task creep
"""

import pytest

from farfan_pipeline.core.phases.phase3_chunk_routing import (
```

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

```
                expected_elements=[],
                document_position=None
            )

    def test_chunk_id_empty_raises_error(self):
        """Test that empty chunk_id raises ValueError."""
        chunk = ChunkData(
            id=0,
            text="Test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=10,
            confidence=0.9,
            policy_area_id="PA01",
            dimension_id="DIM01",
        )

        with pytest.raises(ValueError, match="chunk_id cannot be empty"):
            ChunkRoutingResult(
                target_chunk=chunk,
                chunk_id="",
                policy_area_id="PA01",
                dimension_id="DIM01",
                text_content="Test",
                expected_elements=[],
                document_position=None
            )

    def test_expected_elements_none_raises_error(self):
        """Test that None expected_elements raises ValueError."""
        chunk = ChunkData(
            id=0,
            text="Test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=10,
            confidence=0.9,
            policy_area_id="PA01",
            dimension_id="DIM01",
        )

        with pytest.raises(ValueError, match="expected_elements cannot be None"):
            ChunkRoutingResult(
                target_chunk=chunk,
                chunk_id="PA01-DIM01",
                policy_area_id="PA01",
                dimension_id="DIM01",
                text_content="Test",
                expected_elements=None,
                document_position=None
            )

```

```
    def test_document_position_can_be_none(self):
        """Test that document_position can be None."""
        chunk = ChunkData(
            id=0,
            text="Test",
            chunk_type="diagnostic",
            sentences=[],
            tables=[],
            start_pos=0,
            end_pos=10,
            confidence=0.9,
            policy_area_id="PA01",
            dimension_id="DIM01",
        )

        result = ChunkRoutingResult(
            target_chunk=chunk,
            chunk_id="PA01-DIM01",
            policy_area_id="PA01",
            dimension_id="DIM01",
            text_content="Test",
            expected_elements=[],
            document_position=None
        )

        assert result.document_position is None


class TestStrictEqualityEnforcement:
    """Test strict policy_area_id and dimension_id equality enforcement."""

    @pytest.fixture
    def phase3_contract(self):
        """Create Phase 3 contract instance."""
        return Phase3ChunkRoutingContract()

    @pytest.fixture
    def preprocessed_doc_with_chunks(self):
        """Create preprocessed document with 60 chunks."""
        chunks = []
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunk = ChunkData(
                    id=len(chunks),
                    text=f"Content for {pa_id} {dim_id}",
                    chunk_type="diagnostic",
                    sentences=[len(chunks)],
                    tables=[],
                    start_pos=len(chunks) * 100,
                    end_pos=(len(chunks) + 1) * 100,
                    confidence=0.95,
                    chunk_id=f"{pa_id}-{dim_id}",
                    policy_area_id=pa_id,
                    dimension_id=dim_id,
```

```
                )
                chunks.append(chunk)

        return PreprocessedDocument(
            document_id="test_doc",
            raw_text="Test document",
            sentences=[{"text": f"Sentence {i}"} for i in range(60)],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

    @pytest.mark.asyncio
    async def test_exact_match_succeeds(self, phase3_contract, preprocessed_doc_with_chunks):
        """Test that exact PA and DIM match succeeds."""
        questions = [
            {
                "question_id": "Q001",
                "policy_area_id": "PA01",
                "dimension_id": "DIM01"
            }
        ]

        phase3_input = Phase3Input(
            preprocessed_document=preprocessed_doc_with_chunks,
            questions=questions
        )

        result = await phase3_contract.execute(phase3_input)

        assert result.successful_routes == 1
        assert result.failed_routes == 0
        assert len(result.routing_results) == 1

        routing_result = result.routing_results[0]
        assert routing_result.policy_area_id == "PA01"
        assert routing_result.dimension_id == "DIM01"

    @pytest.mark.asyncio
    async def test_dimension_format_normalization(self, phase3_contract, preprocessed_doc_with_chunks):
        """Test that D1 format is normalized to DIM01."""
        questions = [
            {
                "question_id": "Q001",
                "policy_area_id": "PA01",
                "dimension_id": "D1"  # Should be normalized to DIM01
            }
        ]

        phase3_input = Phase3Input(
            preprocessed_document=preprocessed_doc_with_chunks,
            questions=questions
        )

        result = await phase3_contract.execute(phase3_input)
```

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

```
                "policy_area_id": "PA10",
                "dimension_id": "DIM06"
            }
        ]

        # This should fail during chunk matrix validation
        phase3_input = Phase3Input(
            preprocessed_document=preprocessed_doc_incomplete,
            questions=questions
        )

        # The execution should fail during matrix construction
        with pytest.raises(ValueError, match="Expected 60 chunks"):
            await phase3_contract.execute(phase3_input)

    @pytest.mark.asyncio
    async def test_missing_policy_area_id_raises_error(self, phase3_contract):
        """Test that missing policy_area_id raises descriptive error."""
        chunks = []
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunk = ChunkData(
                    id=len(chunks),
                    text=f"Content for {pa_id} {dim_id}",
                    chunk_type="diagnostic",
                    sentences=[],
                    tables=[],
                    start_pos=0,
                    end_pos=100,
                    confidence=0.95,
                    chunk_id=f"{pa_id}-{dim_id}",
                    policy_area_id=pa_id,
                    dimension_id=dim_id,
                )
                chunks.append(chunk)

        doc = PreprocessedDocument(
            document_id="test",
            raw_text="Test",
            sentences=[],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

        questions = [
            {
                "question_id": "Q001",
                # Missing policy_area_id
                "dimension_id": "DIM01"
            }
        ]

```

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

```
    def complete_document(self):
        """Create complete 60-chunk document."""
        chunks = []
        for pa_num in range(1, 11):
            for dim_num in range(1, 7):
                pa_id = f"PA{pa_num:02d}"
                dim_id = f"DIM{dim_num:02d}"
                chunk = ChunkData(
                    id=len(chunks),
                    text=f"Content for {pa_id} {dim_id}",
                    chunk_type="diagnostic",
                    sentences=[],
                    tables=[],
                    start_pos=0,
                    end_pos=100,
                    confidence=0.95,
                    chunk_id=f"{pa_id}-{dim_id}",
                    policy_area_id=pa_id,
                    dimension_id=dim_id,
                )
                chunks.append(chunk)

        return PreprocessedDocument(
            document_id="complete",
            raw_text="Test",
            sentences=[],
            tables=[],
            metadata={"chunk_count": 60},
            chunks=chunks,
            processing_mode="chunked",
        )

    @pytest.mark.asyncio
    async def test_routing_outcomes_recorded(self, phase3_contract, complete_document):
        """Test that routing outcomes are recorded."""
        questions = [
            {"question_id": f"Q{i:03d}", "policy_area_id": f"PA{(i % 10) + 1:02d}",
             "dimension_id": f"DIM{(i % 6) + 1:02d}"}
            for i in range(30)
        ]

        phase3_input = Phase3Input(
            preprocessed_document=complete_document,
            questions=questions
        )

        result = await phase3_contract.execute(phase3_input)

        # Verify outcomes are recorded
        assert result.total_questions == 30
        assert result.successful_routes + result.failed_routes == result.total_questions

    @pytest.mark.asyncio
    async def test_policy_area_distribution_recorded(self, phase3_contract, complete_document):
        """Test that policy area distribution is recorded."""
        questions = [
```

```
            {"question_id": "Q001", "policy_area_id": "PA01", "dimension_id": "DIM01"},
            {"question_id": "Q002", "policy_area_id": "PA01", "dimension_id": "DIM02"},
            {"question_id": "Q003", "policy_area_id": "PA02", "dimension_id": "DIM01"},
        ]

        phase3_input = Phase3Input(
            preprocessed_document=complete_document,
            questions=questions
        )

        result = await phase3_contract.execute(phase3_input)

        # Verify PA distribution
        assert result.policy_area_distribution["PA01"] == 2
        assert result.policy_area_distribution["PA02"] == 1

    @pytest.mark.asyncio
    async def test_dimension_distribution_recorded(self, phase3_contract, complete_document):
        """Test that dimension distribution is recorded."""
        questions = [
            {"question_id": "Q001", "policy_area_id": "PA01", "dimension_id": "DIM01"},
            {"question_id": "Q002", "policy_area_id": "PA02", "dimension_id": "DIM01"},
            {"question_id": "Q003", "policy_area_id": "PA03", "dimension_id": "DIM02"},
        ]

        phase3_input = Phase3Input(
            preprocessed_document=complete_document,
            questions=questions
        )

        result = await phase3_contract.execute(phase3_input)

        # Verify DIM distribution
        assert result.dimension_distribution["DIM01"] == 2
        assert result.dimension_distribution["DIM02"] == 1


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



================================================================================
FILE: tests/phases/test_phase4_pattern_filtering.py
================================================================================

"""Test Phase 4: Pattern Filtering with Context-Aware Scoping

Tests Phase 4 pattern filtering logic including:
- policy_area_id strict equality filtering (no fuzzy matching)
- Immutable tuple returns for filtered patterns
- Context-based pattern scoping (section, chapter, page)
- Context requirement matching (exact, list, comparison operators)
- Filter statistics tracking
- Pattern preservation (no mutations during filtering)
- Empty pattern list handling
```

```
- Invalid context handling with graceful degradation
"""

from farfan_pipeline.core.orchestrator.signal_context_scoper import (
    context_matches,
    create_document_context,
    evaluate_comparison,
    filter_patterns_by_context,
    in_scope,
)


class TestPhase4PolicyAreaStrictEquality:
    """Test policy_area_id strict equality filtering with no fuzzy matching."""

    def test_exact_policy_area_match(self):
        """Test exact policy_area_id match filters pattern correctly."""
        patterns = [
            {"id": "p1", "pattern": "test", "policy_area_id": "PA01"},
            {"id": "p2", "pattern": "test2", "policy_area_id": "PA02"},
        ]
        context = {"policy_area": "PA01"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert (
            len(filtered) == 2
        )  # No policy_area filtering in base filter_patterns_by_context

    def test_policy_area_id_case_sensitive(self):
        """Test policy_area_id matching is case-sensitive."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"policy_area": "PA01"},
            },
            {
                "id": "p2",
                "pattern": "test2",
                "context_requirement": {"policy_area": "pa01"},
            },
        ]
        context = {"policy_area": "PA01"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        policy_areas = [
            p.get("context_requirement", {}).get("policy_area") for p in filtered
        ]
        assert "PA01" in policy_areas
        assert "pa01" not in [
            p.get("context_requirement", {}).get("policy_area")
            for p in patterns
            if p not in filtered
        ]
```

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

```
            },
            {
                "id": "p2",
                "pattern": "test2",
                "context_requirement": {"policy_area": "PA0?"},
            },
        ]
        context = {"policy_area": "PA01"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert (
            len(
                [
                    p
                    for p in filtered
                    if "PA*" in str(p.get("context_requirement", {}).get("policy_area"))
                ]
            )
            == 0
        )

    def test_multiple_policy_areas_no_cross_match(self):
        """Test patterns for different policy areas don't cross-match."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"policy_area": "PA01"},
            },
            {
                "id": "p2",
                "pattern": "test",
                "context_requirement": {"policy_area": "PA02"},
            },
            {
                "id": "p3",
                "pattern": "test",
                "context_requirement": {"policy_area": "PA10"},
            },
        ]
        context = {"policy_area": "PA02"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        matched_ids = [
            p["id"]
            for p in filtered
            if p.get("context_requirement", {}).get("policy_area") == "PA02"
        ]
        assert "p2" in matched_ids
        assert "p1" not in [
            p["id"]
            for p in filtered
            if p.get("context_requirement", {}).get("policy_area") == "PA01"
        ]
```

```
        assert "p3" not in [
            p["id"]
            for p in filtered
            if p.get("context_requirement", {}).get("policy_area") == "PA10"
        ]

    def test_policy_area_range_not_supported(self):
        """Test policy area ranges like PA01-PA05 are not expanded."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"policy_area": "PA01-PA05"},
            },
        ]
        context = {"policy_area": "PA03"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert (
            len(
                [
                    p
                    for p in filtered
                    if p.get("context_requirement", {}).get("policy_area") == "PA03"
                ]
            )
            == 0
        )


class TestPhase4ImmutableTupleReturns:
    """Test filtered patterns returned as immutable tuples."""

    def test_filtered_patterns_is_list(self):
        """Test filter_patterns_by_context returns list (not tuple in this implementation)."""
        patterns = [{"id": "p1", "pattern": "test", "context_scope": "global"}]
        context = {}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert isinstance(filtered, list)

    def test_filtered_patterns_preserves_order(self):
        """Test filtered patterns preserve original order."""
        patterns = [
            {"id": "p1", "pattern": "test1", "context_scope": "global"},
            {"id": "p2", "pattern": "test2", "context_scope": "global"},
            {"id": "p3", "pattern": "test3", "context_scope": "global"},
        ]
        context = {}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert [p["id"] for p in filtered] == ["p1", "p2", "p3"]

```

```
    def test_pattern_objects_not_mutated(self):
        """Test original pattern objects are not mutated during filtering."""
        original_pattern = {
            "id": "p1",
            "pattern": "test",
            "context_scope": "global",
            "extra": "data",
        }
        patterns = [original_pattern.copy()]
        context = {}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert original_pattern["extra"] == "data"
        assert filtered[0]["extra"] == "data"

    def test_empty_filter_result_returns_empty_list(self):
        """Test empty filter results return empty list."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"section": "budget"},
            },
        ]
        context = {"section": "introduction"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert isinstance(filtered, list)
        assert len(filtered) == 0

    def test_pattern_dict_structure_preserved(self):
        """Test pattern dictionary structure is preserved after filtering."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_scope": "global",
                "metadata": {"source": "test"},
                "nested": {"key": "value"},
            }
        ]
        context = {}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert filtered[0]["id"] == "p1"
        assert filtered[0]["metadata"]["source"] == "test"
        assert filtered[0]["nested"]["key"] == "value"

    def test_no_reference_sharing_between_input_output(self):
        """Test filtered patterns don't share references with input patterns."""
        pattern = {"id": "p1", "pattern": "test", "context_scope": "global"}
        patterns = [pattern]
        context = {}
```

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

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

```
    def test_list_context_requirement_match(self):
        """Test list of acceptable values in context requirement."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"section": ["budget", "financial", "economic"]},
            }
        ]

        # Match first value
        filtered, _ = filter_patterns_by_context(patterns, {"section": "budget"})
        assert len(filtered) == 1

        # Match middle value
        filtered, _ = filter_patterns_by_context(patterns, {"section": "financial"})
        assert len(filtered) == 1

        # No match
        filtered, _ = filter_patterns_by_context(patterns, {"section": "introduction"})
        assert len(filtered) == 0

    def test_comparison_operator_greater_than(self):
        """Test > comparison operator in context requirement."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_requirement": {"chapter": ">2"}},
        ]

        # Chapter 3 > 2
        filtered, _ = filter_patterns_by_context(patterns, {"chapter": 3})
        assert len(filtered) == 1

        # Chapter 2 not > 2
        filtered, _ = filter_patterns_by_context(patterns, {"chapter": 2})
        assert len(filtered) == 0

        # Chapter 1 not > 2
        filtered, _ = filter_patterns_by_context(patterns, {"chapter": 1})
        assert len(filtered) == 0

    def test_comparison_operator_greater_than_or_equal(self):
        """Test >= comparison operator in context requirement."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_requirement": {"page": ">=10"}},
        ]

        # Page 10 >= 10
        filtered, _ = filter_patterns_by_context(patterns, {"page": 10})
        assert len(filtered) == 1

        # Page 11 >= 10
        filtered, _ = filter_patterns_by_context(patterns, {"page": 11})
        assert len(filtered) == 1

        # Page 9 not >= 10
        filtered, _ = filter_patterns_by_context(patterns, {"page": 9})
```

```
        assert len(filtered) == 0

    def test_comparison_operator_less_than(self):
        """Test < comparison operator in context requirement."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_requirement": {"chapter": "<5"}},
        ]

        # Chapter 4 < 5
        filtered, _ = filter_patterns_by_context(patterns, {"chapter": 4})
        assert len(filtered) == 1

        # Chapter 5 not < 5
        filtered, _ = filter_patterns_by_context(patterns, {"chapter": 5})
        assert len(filtered) == 0

    def test_comparison_operator_less_than_or_equal(self):
        """Test <= comparison operator in context requirement."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_requirement": {"page": "<=20"}},
        ]

        # Page 20 <= 20
        filtered, _ = filter_patterns_by_context(patterns, {"page": 20})
        assert len(filtered) == 1

        # Page 19 <= 20
        filtered, _ = filter_patterns_by_context(patterns, {"page": 19})
        assert len(filtered) == 1

        # Page 21 not <= 20
        filtered, _ = filter_patterns_by_context(patterns, {"page": 21})
        assert len(filtered) == 0

    def test_multiple_context_requirements_all_must_match(self):
        """Test multiple context requirements all must match (AND logic)."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"section": "budget", "chapter": ">2"},
            }
        ]

        # Both match
        filtered, _ = filter_patterns_by_context(
            patterns, {"section": "budget", "chapter": 3}
        )
        assert len(filtered) == 1

        # Section matches, chapter doesn't
        filtered, _ = filter_patterns_by_context(
            patterns, {"section": "budget", "chapter": 1}
        )
        assert len(filtered) == 0

```

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

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

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

```
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"section": "budget"},
            },
        ]
        context = {"section": "introduction"}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert len(filtered) == 0


class TestPhase4InvalidContextHandling:
    """Test invalid context handling with graceful degradation."""

    def test_none_context_allows_global_patterns(self):
        """Test None context allows global scope patterns."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_scope": "global"},
        ]

        filtered, stats = filter_patterns_by_context(patterns, {})

        assert len(filtered) == 1

    def test_missing_required_context_field_filters_pattern(self):
        """Test missing required context field filters pattern out."""
        patterns = [
            {
                "id": "p1",
                "pattern": "test",
                "context_requirement": {"section": "budget"},
            },
        ]
        context = {"chapter": 1}  # Missing section field

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert len(filtered) == 0

    def test_invalid_comparison_value_filters_pattern(self):
        """Test invalid comparison value (non-numeric) filters pattern."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_requirement": {"chapter": ">2"}},
        ]
        context = {"chapter": "invalid"}  # String instead of number

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert len(filtered) == 0

    def test_empty_context_requirement_always_matches(self):
        """Test empty context requirement always matches."""
        patterns = [
```

```
            {"id": "p1", "pattern": "test", "context_requirement": {}},
        ]
        context = {}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert len(filtered) == 1

    def test_none_context_requirement_always_matches(self):
        """Test None context requirement always matches."""
        patterns = [
            {"id": "p1", "pattern": "test", "context_requirement": None},
        ]
        context = {}

        filtered, stats = filter_patterns_by_context(patterns, context)

        assert len(filtered) == 1


class TestPhase4HelperFunctions:
    """Test helper functions: context_matches, in_scope, evaluate_comparison."""

    def test_context_matches_exact(self):
        """Test context_matches with exact value match."""
        assert context_matches({"section": "budget"}, {"section": "budget"}) is True
        assert (
            context_matches({"section": "budget"}, {"section": "introduction"}) is False
        )

    def test_context_matches_list(self):
        """Test context_matches with list of values."""
        requirement = {"section": ["budget", "financial"]}
        assert context_matches({"section": "budget"}, requirement) is True
        assert context_matches({"section": "financial"}, requirement) is True
        assert context_matches({"section": "introduction"}, requirement) is False

    def test_context_matches_comparison(self):
        """Test context_matches with comparison operators."""
        assert context_matches({"chapter": 5}, {"chapter": ">2"}) is True
        assert context_matches({"chapter": 1}, {"chapter": ">2"}) is False
        assert context_matches({"page": 10}, {"page": ">=10"}) is True
        assert context_matches({"page": 9}, {"page": ">=10"}) is False

    def test_in_scope_global(self):
        """Test in_scope with global scope."""
        assert in_scope({}, "global") is True
        assert in_scope({"section": "budget"}, "global") is True

    def test_in_scope_section(self):
        """Test in_scope with section scope."""
        assert in_scope({"section": "budget"}, "section") is True
        assert in_scope({}, "section") is False

    def test_in_scope_chapter(self):
        """Test in_scope with chapter scope."""
```

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

```
Tests Phase 5 signal resolution logic including:
- Signal registry integration and query interface
- Missing signal hard stops (no fallbacks or degraded modes)
- Immutable signal tuple returns
- Set-based signal validation
- Per-chunk signal caching
- Required vs optional signal distinction
- Signal type validation
- Error message clarity for missing signals
"""

import pytest

from farfan_pipeline.core.orchestrator.signal_resolution import (
    Chunk,
    Question,
    Signal,
    _resolve_signals,
)


class MockSignalRegistry:
    """Mock signal registry for testing."""

    def __init__(self, signals_to_return=None):
        self.signals_to_return = signals_to_return or []
        self.calls = []

    def get_signals_for_chunk(self, chunk, required_types):
        """Mock get_signals_for_chunk method."""
        self.calls.append(
            {
                "chunk": chunk,
                "required_types": required_types,
            }
        )
        return self.signals_to_return


class TestPhase5SignalRegistryIntegration:
    """Test signal registry integration and query interface."""

    def test_resolve_signals_queries_registry(self):
        """Test _resolve_signals queries signal registry."""
        chunk = Chunk(chunk_id="PA01-DIM01", text="Test chunk")
        question = Question(question_id="Q01", signal_requirements={"signal_type_1"})
        signals = [Signal(signal_type="signal_type_1", content=None)]
        registry = MockSignalRegistry(signals_to_return=signals)

        _resolve_signals(chunk, question, registry)

        assert len(registry.calls) == 1
        assert registry.calls[0]["chunk"] == chunk
        assert registry.calls[0]["required_types"] == {"signal_type_1"}

    def test_resolve_signals_passes_chunk_to_registry(self):
```

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

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

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

```
        """Test Signal objects are immutable NamedTuples."""
        signal = Signal(signal_type="test", content=None)

        assert isinstance(signal, tuple)
        assert hasattr(signal, "signal_type")
        assert hasattr(signal, "content")

        with pytest.raises(AttributeError):
            signal.signal_type = "modified"

    def test_empty_result_returns_empty_tuple(self):
        """Test empty signal list returns empty tuple."""
        chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
        question = Question(question_id="Q15", signal_requirements=set())
        registry = MockSignalRegistry(signals_to_return=[])

        result = _resolve_signals(chunk, question, registry)

        assert result == ()
        assert isinstance(result, tuple)

    def test_signal_order_preserved_in_tuple(self):
        """Test signal order preserved in returned tuple."""
        chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
        question = Question(
            question_id="Q16", signal_requirements={"type_1", "type_2", "type_3"}
        )
        signals = [
            Signal(signal_type="type_1", content="first"),
            Signal(signal_type="type_2", content="second"),
            Signal(signal_type="type_3", content="third"),
        ]
        registry = MockSignalRegistry(signals_to_return=signals)

        result = _resolve_signals(chunk, question, registry)

        assert result[0].content == "first"
        assert result[1].content == "second"
        assert result[2].content == "third"


class TestPhase5SetBasedValidation:
    """Test set-based signal validation."""

    def test_required_types_compared_as_set(self):
        """Test required signal types compared as set (order independent)."""
        chunk = Chunk(chunk_id="PA01-DIM01", text="Test")
        question = Question(
            question_id="Q17", signal_requirements={"type_c", "type_a", "type_b"}
        )
        signals = [
            Signal(signal_type="type_a", content=None),
            Signal(signal_type="type_b", content=None),
            Signal(signal_type="type_c", content=None),
        ]
        registry = MockSignalRegistry(signals_to_return=signals)
```

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

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

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

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

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

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

```
                "scoring": {"method": "weighted"},
            },
            "integrity": {"checksum": "abc123"},
        }

    def test_validate_top_level_keys_present(self, valid_monolith):
        """Test validation checks for required top-level keys."""
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(valid_monolith, strict=False)

        assert report.validation_passed

    def test_missing_schema_version_fails_validation(self):
        """Test missing schema_version fails validation."""
        monolith = {
            "version": "1.0.0",
            "blocks": {},
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.validation_passed is False
        assert any("schema_version" in e for e in report.errors)

    def test_missing_blocks_fails_validation(self):
        """Test missing blocks section fails validation."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.validation_passed is False
        assert any("blocks" in e for e in report.errors)

    def test_required_blocks_present(self, valid_monolith):
        """Test all required blocks are present."""
        validator = MonolithSchemaValidator()

        validator.validate_monolith(valid_monolith, strict=False)

        assert "niveles_abstraccion" in valid_monolith["blocks"]
        assert "micro_questions" in valid_monolith["blocks"]
        assert "meso_questions" in valid_monolith["blocks"]
        assert "macro_question" in valid_monolith["blocks"]

    def test_missing_required_block_fails_validation(self, valid_monolith):
        """Test missing required block fails validation."""
        del valid_monolith["blocks"]["micro_questions"]
        validator = MonolithSchemaValidator()
```

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

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

```
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.question_counts["meso_questions"] == 4

    def test_incorrect_meso_questions_count_fails(self):
        """Test incorrect meso_questions count fails validation."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 4)],  # Only 3
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.validation_passed is False

    def test_macro_question_count_1(self):
        """Test macro_question is single dict (count=1)."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.question_counts.get("macro_question") == 1
```

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

```
        # Check for required_field presence
        has_required = [("required_field" in q) for q in questions]
        assert not all(has_required)

    def test_extra_key_in_subset_detected(self):
        """Test extra key in subset of questions detected."""
        questions = [
            {"id": "Q001", "text": "Question 1"},
            {"id": "Q002", "text": "Question 2", "extra": "field"},  # Extra key
            {"id": "Q003", "text": "Question 3"},
        ]

        key_counts = {}
        for q in questions:
            for key in q.keys():
                key_counts[key] = key_counts.get(key, 0) + 1

        assert key_counts["extra"] == 1  # Only in one question

    def test_nested_dict_key_consistency(self):
        """Test nested dict key consistency validation."""
        questions = [
            {"id": "Q001", "metadata": {"author": "A", "date": "2024"}},
            {"id": "Q002", "metadata": {"author": "B", "date": "2024"}},
            {"id": "Q003", "metadata": {"author": "C", "date": "2024"}},
        ]

        nested_key_sets = [set(q["metadata"].keys()) for q in questions]
        assert all(keys == nested_key_sets[0] for keys in nested_key_sets)


class TestPhase6SemanticValidation:
    """Test semantic validation including type and required field rules."""

    def test_required_field_id_present(self):
        """Test required 'id' field is present in all questions."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [
                    {"id": "Q001", "text": "Q1"},
                    {"id": "Q002", "text": "Q2"},
                ],
                "meso_questions": [{"id": "M01"}],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }

        # Check all questions have id
        all_have_id = all("id" in q for q in monolith["blocks"]["micro_questions"])
        assert all_have_id is True

```

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

```
        question_type = "invalid_type"

        assert question_type not in valid_types


class TestPhase6MinimumValueConstraints:
    """Test minimum value constraints validation."""

    def test_question_count_minimum_300(self):
        """Test micro_questions minimum count is 300."""
        validator = MonolithSchemaValidator()
        assert validator.EXPECTED_MICRO_QUESTIONS == 300

    def test_below_minimum_count_fails(self):
        """Test question count below minimum fails validation."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [
                    {"id": f"Q{i:03d}"} for i in range(1, 100)
                ],  # Only 99
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.validation_passed is False

    def test_weight_minimum_zero(self):
        """Test weight field minimum value is 0."""
        weight = 0.5
        assert weight >= 0

    def test_negative_weight_invalid(self):
        """Test negative weight is invalid."""
        weight = -0.1
        assert weight < 0  # Should be detected as invalid

    def test_confidence_range_0_to_1(self):
        """Test confidence value in range [0, 1]."""
        valid_confidences = [0.0, 0.5, 0.75, 1.0]

        for conf in valid_confidences:
            assert 0.0 <= conf <= 1.0

    def test_confidence_out_of_range_detected(self):
        """Test confidence value outside range detected."""
        invalid_confidences = [-0.1, 1.5, 2.0]

```

```
        for conf in invalid_confidences:
            assert not (0.0 <= conf <= 1.0)


class TestPhase6SchemaVersionValidation:
    """Test schema version validation."""

    def test_schema_version_format_validation(self):
        """Test schema version follows semantic versioning."""
        version = "2.0.0"
        parts = version.split(".")

        assert len(parts) == 3
        assert all(part.isdigit() for part in parts)

    def test_expected_schema_version_2_0_0(self):
        """Test expected schema version is 2.0.0."""
        validator = MonolithSchemaValidator()
        assert validator.EXPECTED_SCHEMA_VERSION == "2.0.0"

    def test_different_schema_version_warning(self):
        """Test different schema version generates warning."""
        monolith = {
            "schema_version": "1.5.0",  # Different version
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert len(report.warnings) > 0

    def test_missing_schema_version_error(self):
        """Test missing schema_version generates error."""
        monolith = {
            "version": "1.0.0",
            "blocks": {},
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.validation_passed is False
        assert report.schema_version == ""


class TestPhase6ReferentialIntegrity:
```

```
    """Test referential integrity checking."""

    def test_question_id_uniqueness(self):
        """Test all question IDs are unique."""
        questions = [
            {"id": "Q001"},
            {"id": "Q002"},
            {"id": "Q003"},
        ]

        ids = [q["id"] for q in questions]
        assert len(ids) == len(set(ids))

    def test_duplicate_question_id_detected(self):
        """Test duplicate question ID detected."""
        questions = [
            {"id": "Q001"},
            {"id": "Q002"},
            {"id": "Q001"},  # Duplicate
        ]

        ids = [q["id"] for q in questions]
        assert len(ids) != len(set(ids))

    def test_cross_reference_validation(self):
        """Test cross-reference between blocks validated."""
        monolith = {
            "blocks": {
                "micro_questions": [
                    {"id": "Q001", "meso_parent": "M01"},
                    {"id": "Q002", "meso_parent": "M01"},
                ],
                "meso_questions": [
                    {"id": "M01"},
                ],
            }
        }

        # Check all referenced meso IDs exist
        meso_ids = {q["id"] for q in monolith["blocks"]["meso_questions"]}
        referenced_meso = {
            q.get("meso_parent")
            for q in monolith["blocks"]["micro_questions"]
            if "meso_parent" in q
        }

        assert referenced_meso.issubset(meso_ids)

    def test_broken_cross_reference_detected(self):
        """Test broken cross-reference detected."""
        monolith = {
            "blocks": {
                "micro_questions": [
                    {"id": "Q001", "meso_parent": "M01"},
                    {"id": "Q002", "meso_parent": "M99"},  # Doesn't exist
                ],
```

```
                "meso_questions": [
                    {"id": "M01"},
                ],
            }
        }

        meso_ids = {q["id"] for q in monolith["blocks"]["meso_questions"]}
        referenced_meso = {
            q.get("meso_parent")
            for q in monolith["blocks"]["micro_questions"]
            if "meso_parent" in q
        }

        assert not referenced_meso.issubset(meso_ids)


class TestPhase6FieldCoverageValidation:
    """Test field coverage validation."""

    def test_all_questions_have_required_fields(self):
        """Test all questions have required fields."""
        required_fields = {"id", "text"}
        questions = [
            {"id": "Q001", "text": "Question 1", "dimension": "D1"},
            {"id": "Q002", "text": "Question 2", "dimension": "D2"},
        ]

        for q in questions:
            assert required_fields.issubset(set(q.keys()))

    def test_missing_optional_field_allowed(self):
        """Test missing optional field is allowed."""
        required_fields = {"id", "text"}
        question = {"id": "Q001", "text": "Question 1"}

        assert required_fields.issubset(set(question.keys()))
        assert "optional_field" not in question

    def test_field_coverage_percentage(self):
        """Test field coverage percentage calculation."""
        all_possible_fields = {"id", "text", "dimension", "weight", "type"}
        questions = [
            {"id": "Q001", "text": "Q1", "dimension": "D1"},
            {"id": "Q002", "text": "Q2", "dimension": "D2", "weight": 1.0},
        ]

        total_fields = len(all_possible_fields) * len(questions)
        present_fields = sum(
            len(set(q.keys()) & all_possible_fields) for q in questions
        )
        coverage = present_fields / total_fields

        assert 0.0 <= coverage <= 1.0


class TestPhase6HashCalculation:
```

```
    """Test hash calculation and verification."""

    def test_schema_hash_calculated(self):
        """Test schema hash is calculated in report."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.schema_hash
        assert len(report.schema_hash) > 0

    def test_same_monolith_same_hash(self):
        """Test same monolith produces same hash."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {"test": "data"},
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report1 = validator.validate_monolith(monolith, strict=False)
        report2 = validator.validate_monolith(monolith, strict=False)

        assert report1.schema_hash == report2.schema_hash

    def test_different_monolith_different_hash(self):
        """Test different monolith produces different hash."""
        monolith1 = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {"test": "data1"},
            "integrity": {},
        }
        monolith2 = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {"test": "data2"},
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report1 = validator.validate_monolith(monolith1, strict=False)
        report2 = validator.validate_monolith(monolith2, strict=False)
```

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

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

```
        """Test validation report contains referential integrity dict."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [{"id": f"Q{i:03d}"} for i in range(1, 301)],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert isinstance(report.referential_integrity, dict)


class TestPhase6StrictMode:
    """Test strict mode validation behavior."""

    def test_strict_mode_raises_exception_on_error(self):
        """Test strict mode raises SchemaInitializationError on validation failure."""
        monolith = {
            "version": "1.0.0",  # Missing schema_version
            "blocks": {},
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        with pytest.raises(SchemaInitializationError):
            validator.validate_monolith(monolith, strict=True)

    def test_non_strict_mode_returns_report(self):
        """Test non-strict mode returns report without raising."""
        monolith = {
            "version": "1.0.0",  # Missing schema_version
            "blocks": {},
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert isinstance(report, MonolithIntegrityReport)
        assert report.validation_passed is False

    def test_strict_mode_exception_contains_error_details(self):
        """Test strict mode exception contains detailed error messages."""
        monolith = {
            "version": "1.0.0",
            "blocks": {},
            "integrity": {},
        }
```

```
        validator = MonolithSchemaValidator()

        with pytest.raises(SchemaInitializationError) as exc_info:
            validator.validate_monolith(monolith, strict=True)

        error_msg = str(exc_info.value)
        assert "Schema initialization failed" in error_msg


class TestPhase6EdgeCases:
    """Test edge cases in schema validation."""

    def test_empty_monolith_dict(self):
        """Test validation with empty monolith dict."""
        monolith = {}
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        assert report.validation_passed is False
        assert len(report.errors) > 0

    def test_none_monolith_handled(self):
        """Test validation handles None monolith gracefully."""
        validator = MonolithSchemaValidator()

        with pytest.raises((TypeError, AttributeError)):
            validator.validate_monolith(None, strict=False)

    def test_deeply_nested_structure(self):
        """Test validation handles deeply nested structures."""
        monolith = {
            "schema_version": "2.0.0",
            "version": "1.0.0",
            "blocks": {
                "niveles_abstraccion": ["micro", "meso", "macro"],
                "micro_questions": [
                    {
                        "id": f"Q{i:03d}",
                        "nested": {"level1": {"level2": {"level3": "deep_value"}}},
                    }
                    for i in range(1, 301)
                ],
                "meso_questions": [{"id": f"M{i:02d}"} for i in range(1, 5)],
                "macro_question": {"id": "MACRO_01"},
                "scoring": {},
            },
            "integrity": {},
        }
        validator = MonolithSchemaValidator()

        report = validator.validate_monolith(monolith, strict=False)

        # Should not crash
        assert isinstance(report, MonolithIntegrityReport)

```

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

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

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

```
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )

        with pytest.raises(FrozenInstanceError):
            task.question_id = "D1-Q2"  # type: ignore[misc]

    def test_question_global_immutable(self):
        """Verify question_global cannot be modified after creation."""
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )

        with pytest.raises(FrozenInstanceError):
            task.question_global = 2  # type: ignore[misc]

    def test_policy_area_id_immutable(self):
        """Verify policy_area_id cannot be modified after creation."""
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )

        with pytest.raises(FrozenInstanceError):
            task.policy_area_id = "PA02"  # type: ignore[misc]

    def test_dimension_id_immutable(self):
        """Verify dimension_id cannot be modified after creation."""
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
```

```
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )

        with pytest.raises(FrozenInstanceError):
            task.dimension_id = "DIM02"  # type: ignore[misc]

    def test_chunk_id_immutable(self):
        """Verify chunk_id cannot be modified after creation."""
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )

        with pytest.raises(FrozenInstanceError):
            task.chunk_id = "chunk_002"  # type: ignore[misc]

    def test_patterns_immutable(self):
        """Verify patterns list cannot be reassigned."""
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[{"type": "pattern1"}],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )

        with pytest.raises(FrozenInstanceError):
            task.patterns = [{"type": "pattern2"}]  # type: ignore[misc]

    def test_signals_immutable(self):
        """Verify signals dict cannot be reassigned."""
        task = ExecutableTask(
            task_id="MQC-001_PA01",
            question_id="D1-Q1",
            question_global=1,
            policy_area_id="PA01",
```

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

```
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={"key": "value"},
        )

        with pytest.raises(FrozenInstanceError):
            task.metadata = {}  # type: ignore[misc]


class TestPhase7PostInitValidation:
    """Test __post_init__ validation for all mandatory fields."""

    def test_empty_task_id_raises_error(self):
        """Test empty task_id raises ValueError."""
        with pytest.raises(ValueError, match="task_id cannot be empty"):
            ExecutableTask(
                task_id="",
                question_id="D1-Q1",
                question_global=1,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_empty_question_id_raises_error(self):
        """Test empty question_id raises ValueError."""
        with pytest.raises(ValueError, match="question_id cannot be empty"):
            ExecutableTask(
                task_id="MQC-001_PA01",
                question_id="",
                question_global=1,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_non_integer_question_global_raises_error(self):
        """Test non-integer question_global raises ValueError."""
        with pytest.raises(
            ValueError, match="question_global must be an integer, got str"
        ):
```

```
            ExecutableTask(
                task_id="MQC-001_PA01",
                question_id="D1-Q1",
                question_global="1",  # type: ignore[arg-type]
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_question_global_below_zero_raises_error(self):
        """Test question_global below 0 raises ValueError."""
        with pytest.raises(
            ValueError,
            match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}",
        ):
            ExecutableTask(
                task_id="MQC-001_PA01",
                question_id="D1-Q1",
                question_global=-1,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_question_global_above_max_raises_error(self):
        """Test question_global above MAX_QUESTION_GLOBAL raises ValueError."""
        with pytest.raises(
            ValueError,
            match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}",
        ):
            ExecutableTask(
                task_id="MQC-999_PA01",
                question_id="D1-Q1",
                question_global=MAX_QUESTION_GLOBAL + 1,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_empty_policy_area_id_raises_error(self):
        """Test empty policy_area_id raises ValueError."""
```

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

```
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="",
                expected_elements=[],
                metadata={},
            )

    def test_valid_minimum_question_global(self):
        """Test valid minimum question_global value (0)."""
        task = ExecutableTask(
            task_id="MQC-000_PA01",
            question_id="D1-Q0",
            question_global=0,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )
        assert task.question_global == 0

    def test_valid_maximum_question_global(self):
        """Test valid maximum question_global value (MAX_QUESTION_GLOBAL)."""
        task = ExecutableTask(
            task_id=f"MQC-{MAX_QUESTION_GLOBAL:03d}_PA01",
            question_id="D1-Q999",
            question_global=MAX_QUESTION_GLOBAL,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )
        assert task.question_global == MAX_QUESTION_GLOBAL

    def test_all_valid_fields_create_task(self):
        """Test task creation with all valid fields."""
        task = ExecutableTask(
            task_id="MQC-042_PA05",
            question_id="D2-Q12",
            question_global=42,
            policy_area_id="PA05",
            dimension_id="DIM02",
            chunk_id="chunk_010",
            patterns=[{"type": "pattern1"}, {"type": "pattern2"}],
            signals={"signal1": 0.8, "signal2": 0.9},
            creation_timestamp="2024-01-01T12:30:45.123456Z",
```

```
            expected_elements=[{"type": "test", "minimum": 2}],
            metadata={"key1": "value1", "key2": "value2"},
        )
        assert task.task_id == "MQC-042_PA05"
        assert task.question_id == "D2-Q12"
        assert task.question_global == 42
        assert task.policy_area_id == "PA05"
        assert task.dimension_id == "DIM02"
        assert task.chunk_id == "chunk_010"


class TestPhase7ConstructTaskIntegration:
    """Test integration with _construct_task function."""

    def test_construct_task_generates_correct_task_id(self):
        """Test _construct_task generates correct task_id format."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result(policy_area_id="PA01")
        generated_task_ids: set[str] = set()
        correlation_id = "corr-123"

        task = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids,
            correlation_id,
        )

        assert task.task_id == "MQC-001_PA01"
        assert "MQC-001_PA01" in generated_task_ids

    def test_construct_task_with_various_question_globals(self):
        """Test _construct_task with various question_global values."""
        test_cases = [
            (0, "MQC-000_PA01"),
            (1, "MQC-001_PA01"),
            (50, "MQC-050_PA01"),
            (150, "MQC-150_PA01"),
            (300, "MQC-300_PA01"),
            (999, "MQC-999_PA01"),
        ]

        for question_global, expected_task_id in test_cases:
            question = {
                "question_id": f"Q{question_global}",
                "question_global": question_global,
                "dimension_id": "DIM01",
```

```
                "base_slot": f"Q{question_global}",
                "cluster_id": "CL01",
                "expected_elements": [],
            }
            routing_result = create_test_chunk_routing_result(policy_area_id="PA01")
            generated_task_ids: set[str] = set()

            task = _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

            assert task.task_id == expected_task_id
            assert task.question_global == question_global

    def test_construct_task_detects_duplicate_task_ids(self):
        """Test _construct_task detects duplicate task_ids."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result(policy_area_id="PA01")
        generated_task_ids = {"MQC-001_PA01"}  # Pre-populate with duplicate

        with pytest.raises(
            ValueError, match="Duplicate task_id detected: MQC-001_PA01"
        ):
            _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

    def test_construct_task_missing_question_global(self):
        """Test _construct_task raises error when question_global is missing."""
        question = {
            "question_id": "D1-Q1",
            # question_global is missing
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        with pytest.raises(ValueError, match="question_global field missing or None"):
```

```
            _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

    def test_construct_task_question_global_none(self):
        """Test _construct_task raises error when question_global is None."""
        question = {
            "question_id": "D1-Q1",
            "question_global": None,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        with pytest.raises(ValueError, match="question_global field missing or None"):
            _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

    def test_construct_task_question_global_not_integer(self):
        """Test _construct_task raises error when question_global is not integer."""
        question = {
            "question_id": "D1-Q1",
            "question_global": "1",  # String instead of int
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        with pytest.raises(
            ValueError, match="question_global must be an integer, got str"
        ):
            _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

    def test_construct_task_question_global_below_range(self):
        """Test _construct_task raises error when question_global is below 0."""
        question = {
```

```
            "question_id": "D1-Q1",
            "question_global": -1,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        with pytest.raises(
            ValueError,
            match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}, got -1",
        ):
            _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

    def test_construct_task_question_global_above_range(self):
        """Test _construct_task raises error when question_global exceeds MAX."""
        question = {
            "question_id": "D1-Q1000",
            "question_global": 1000,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        with pytest.raises(
            ValueError,
            match=f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}, got 1000",
        ):
            _construct_task(
                question,
                routing_result,
                (),
                (),
                generated_task_ids,
                "corr-id",
            )

    def test_construct_task_with_patterns_tuple(self):
        """Test _construct_task handles patterns as tuple."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
```

```
        applicable_patterns = (
            {"type": "pattern1", "value": 0.8},
            {"type": "pattern2", "value": 0.9},
        )
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            applicable_patterns,
            (),
            generated_task_ids,
            "corr-id",
        )

        assert len(task.patterns) == 2
        assert isinstance(task.patterns, list)
        assert task.patterns[0]["type"] == "pattern1"
        assert task.patterns[1]["type"] == "pattern2"

    def test_construct_task_with_signals_tuple(self):
        """Test _construct_task handles signals as tuple and converts to dict."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()

        # Signals as tuple of dicts with signal_type
        resolved_signals = (
            {"signal_type": "signal1", "value": 0.8},
            {"signal_type": "signal2", "value": 0.9},
        )
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            (),
            resolved_signals,
            generated_task_ids,
            "corr-id",
        )

        assert isinstance(task.signals, dict)
        assert "signal1" in task.signals
        assert "signal2" in task.signals
        assert task.signals["signal1"]["value"] == 0.8
        assert task.signals["signal2"]["value"] == 0.9

    def test_construct_task_with_expected_elements(self):
        """Test _construct_task includes expected_elements in task."""
```

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

```
        assert "document_position" in task.metadata
        assert "synchronizer_version" in task.metadata
        assert "correlation_id" in task.metadata
        assert "original_pattern_count" in task.metadata
        assert "original_signal_count" in task.metadata
        assert "filtered_pattern_count" in task.metadata
        assert "resolved_signal_count" in task.metadata
        assert "schema_element_count" in task.metadata

        assert task.metadata["base_slot"] == "D1-Q1"
        assert task.metadata["cluster_id"] == "CL01"
        assert task.metadata["document_position"] == (100, 200)
        assert task.metadata["correlation_id"] == "corr-abc-123"
        assert task.metadata["original_pattern_count"] == 2
        assert task.metadata["original_signal_count"] == 1
        assert task.metadata["filtered_pattern_count"] == 2
        assert task.metadata["resolved_signal_count"] == 1
        assert task.metadata["schema_element_count"] == 1

    def test_construct_task_timestamp_format(self):
        """Test _construct_task creates ISO 8601 timestamp."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids,
            "corr-id",
        )

        # Timestamp should be ISO 8601 format with timezone
        assert "T" in task.creation_timestamp
        # Should be parseable by datetime
        parsed = datetime.fromisoformat(task.creation_timestamp)
        assert parsed.tzinfo is not None

    def test_construct_task_uses_routing_result_dimension(self):
        """Test _construct_task uses dimension_id from routing_result."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",  # Different from routing_result
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
```

```
        }
        routing_result = create_test_chunk_routing_result(dimension_id="DIM02")
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids,
            "corr-id",
        )

        # Should use routing_result dimension_id, not question dimension_id
        assert task.dimension_id == "DIM02"


class TestPhase7ConstructTaskLegacy:
    """Test legacy _construct_task_legacy function."""

    def test_construct_task_legacy_generates_correct_task_id(self):
        """Test _construct_task_legacy generates correct task_id format."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "policy_area_id": "PA01",
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_001", "expected_elements": []}
        patterns: list[dict[str, Any]] = []
        signals: dict[str, Any] = {}
        generated_task_ids: set[str] = set()
        routing_result = MockRoutingResult(policy_area_id="PA01")

        task = _construct_task_legacy(
            question, chunk, patterns, signals, generated_task_ids, routing_result
        )

        assert task.task_id == "MQC-001_PA01"
        assert "MQC-001_PA01" in generated_task_ids

    def test_construct_task_legacy_detects_duplicate_task_ids(self):
        """Test _construct_task_legacy detects duplicate task_ids."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_001", "expected_elements": []}
        generated_task_ids = {"MQC-001_PA01"}
        routing_result = MockRoutingResult(policy_area_id="PA01")

        with pytest.raises(
            ValueError,
```

```
            match="Duplicate task_id detected: MQC-001_PA01 for question D1-Q1",
        ):
            _construct_task_legacy(
                question, chunk, [], {}, generated_task_ids, routing_result
            )

    def test_construct_task_legacy_invalid_question_global_type(self):
        """Test _construct_task_legacy raises error for non-integer question_global."""
        question = {
            "question_id": "D1-Q1",
            "question_global": "not_an_int",
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_001"}
        generated_task_ids: set[str] = set()
        routing_result = MockRoutingResult()

        with pytest.raises(
            ValueError, match="Invalid question_global.*Must be an integer in range"
        ):
            _construct_task_legacy(
                question, chunk, [], {}, generated_task_ids, routing_result
            )

    def test_construct_task_legacy_question_global_below_range(self):
        """Test _construct_task_legacy raises error for negative question_global."""
        question = {
            "question_id": "D1-Q1",
            "question_global": -1,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_001"}
        generated_task_ids: set[str] = set()
        routing_result = MockRoutingResult()

        with pytest.raises(
            ValueError, match="Invalid question_global.*Must be an integer in range"
        ):
            _construct_task_legacy(
                question, chunk, [], {}, generated_task_ids, routing_result
            )

    def test_construct_task_legacy_question_global_above_range(self):
        """Test _construct_task_legacy raises error for question_global > MAX."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1000,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        chunk = {"id": "chunk_001"}
        generated_task_ids: set[str] = set()
        routing_result = MockRoutingResult()

```

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

```
        generated_task_ids: set[str] = set()
        routing_result = MockRoutingResult()

        task = _construct_task_legacy(
            question, chunk, [], {}, generated_task_ids, routing_result
        )

        # Should have "T" separator for ISO 8601
        assert "T" in task.creation_timestamp
        # Should be parseable
        parsed = datetime.fromisoformat(task.creation_timestamp)
        assert parsed is not None


class TestPhase7BoundaryConditions:
    """Test boundary conditions for question_global range."""

    def test_question_global_boundary_zero(self):
        """Test question_global=0 is valid."""
        task = ExecutableTask(
            task_id="MQC-000_PA01",
            question_id="Q0",
            question_global=0,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )
        assert task.question_global == 0

    def test_question_global_boundary_max(self):
        """Test question_global=MAX_QUESTION_GLOBAL is valid."""
        task = ExecutableTask(
            task_id=f"MQC-{MAX_QUESTION_GLOBAL:03d}_PA01",
            question_id=f"Q{MAX_QUESTION_GLOBAL}",
            question_global=MAX_QUESTION_GLOBAL,
            policy_area_id="PA01",
            dimension_id="DIM01",
            chunk_id="chunk_001",
            patterns=[],
            signals={},
            creation_timestamp="2024-01-01T00:00:00Z",
            expected_elements=[],
            metadata={},
        )
        assert task.question_global == MAX_QUESTION_GLOBAL

    def test_question_global_just_below_zero_invalid(self):
        """Test question_global=-1 is invalid."""
        with pytest.raises(ValueError):
            ExecutableTask(
                task_id="MQC-001_PA01",
```

```
                question_id="Q-1",
                question_global=-1,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_question_global_just_above_max_invalid(self):
        """Test question_global=MAX_QUESTION_GLOBAL+1 is invalid."""
        with pytest.raises(ValueError):
            ExecutableTask(
                task_id=f"MQC-{MAX_QUESTION_GLOBAL+1:03d}_PA01",
                question_id=f"Q{MAX_QUESTION_GLOBAL+1}",
                question_global=MAX_QUESTION_GLOBAL + 1,
                policy_area_id="PA01",
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )

    def test_construct_task_boundaries_validated(self):
        """Test _construct_task validates question_global boundaries."""
        # Test minimum valid
        question_min = {
            "question_id": "Q0",
            "question_global": 0,
            "dimension_id": "DIM01",
            "base_slot": "Q0",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        task_min = _construct_task(
            question_min,
            routing_result,
            (),
            (),
            generated_task_ids,
            "corr-id",
        )
        assert task_min.question_global == 0

        # Test maximum valid
        question_max = {
            "question_id": f"Q{MAX_QUESTION_GLOBAL}",
```

```
            "question_global": MAX_QUESTION_GLOBAL,
            "dimension_id": "DIM01",
            "base_slot": f"Q{MAX_QUESTION_GLOBAL}",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        generated_task_ids_max: set[str] = set()

        task_max = _construct_task(
            question_max,
            routing_result,
            (),
            (),
            generated_task_ids_max,
            "corr-id",
        )
        assert task_max.question_global == MAX_QUESTION_GLOBAL

        # Test below minimum invalid
        question_below = {
            "question_id": "Q-1",
            "question_global": -1,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        generated_task_ids_below: set[str] = set()

        with pytest.raises(ValueError, match="question_global must be in range"):
            _construct_task(
                question_below,
                routing_result,
                (),
                (),
                generated_task_ids_below,
                "corr-id",
            )

        # Test above maximum invalid
        question_above = {
            "question_id": f"Q{MAX_QUESTION_GLOBAL+1}",
            "question_global": MAX_QUESTION_GLOBAL + 1,
            "dimension_id": "DIM01",
            "expected_elements": [],
        }
        generated_task_ids_above: set[str] = set()

        with pytest.raises(ValueError, match="question_global must be in range"):
            _construct_task(
                question_above,
                routing_result,
                (),
                (),
                generated_task_ids_above,
                "corr-id",
            )

```

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

```
                task_id=expected_task_id,
                question_id="Q1",
                question_global=1,
                policy_area_id=policy_area_id,
                dimension_id="DIM01",
                chunk_id="chunk_001",
                patterns=[],
                signals={},
                creation_timestamp="2024-01-01T00:00:00Z",
                expected_elements=[],
                metadata={},
            )
            assert task.task_id == expected_task_id
            assert policy_area_id in task.task_id

    def test_construct_task_generates_consistent_task_id(self):
        """Test _construct_task generates consistent task_id from inputs."""
        question = {
            "question_id": "D2-Q25",
            "question_global": 75,
            "dimension_id": "DIM02",
            "base_slot": "D2-Q25",
            "cluster_id": "CL02",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result(policy_area_id="PA07")
        generated_task_ids: set[str] = set()

        task1 = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids,
            "corr-id",
        )

        # Reset and create again
        generated_task_ids2: set[str] = set()
        task2 = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids2,
            "corr-id",
        )

        assert task1.task_id == task2.task_id == "MQC-075_PA07"


class TestPhase7ProvenanceTracking:
    """Test provenance and metadata tracking in task construction."""

    def test_metadata_includes_base_slot(self):
        """Test metadata includes base_slot from question."""
```

```
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1-SLOT",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids,
            "corr-id",
        )

        assert task.metadata["base_slot"] == "D1-Q1-SLOT"

    def test_metadata_includes_cluster_id(self):
        """Test metadata includes cluster_id from question."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CLUSTER-ABC",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            (),
            (),
            generated_task_ids,
            "corr-id",
        )

        assert task.metadata["cluster_id"] == "CLUSTER-ABC"

    def test_metadata_includes_correlation_id(self):
        """Test metadata includes correlation_id for tracing."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
```

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

```
            applicable_patterns,
            (),
            generated_task_ids,
            "corr-id",
        )

        assert task.metadata["original_pattern_count"] == 3
        assert task.metadata["filtered_pattern_count"] == 3

    def test_metadata_tracks_signal_counts(self):
        """Test metadata tracks original and resolved signal counts."""
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": [],
        }
        routing_result = create_test_chunk_routing_result()
        resolved_signals = (
            {"signal_type": "s1", "v": 0.5},
            {"signal_type": "s2", "v": 0.7},
        )
        generated_task_ids: set[str] = set()

        task = _construct_task(
            question,
            routing_result,
            (),
            resolved_signals,
            generated_task_ids,
            "corr-id",
        )

        assert task.metadata["original_signal_count"] == 2
        assert task.metadata["resolved_signal_count"] == 2

    def test_metadata_tracks_schema_element_count(self):
        """Test metadata tracks expected_elements count."""
        expected_elements = [
            {"type": "elem1"},
            {"type": "elem2"},
            {"type": "elem3"},
        ]
        question = {
            "question_id": "D1-Q1",
            "question_global": 1,
            "dimension_id": "DIM01",
            "base_slot": "D1-Q1",
            "cluster_id": "CL01",
            "expected_elements": expected_elements,
        }
        routing_result = create_test_chunk_routing_result()
        generated_task_ids: set[str] = set()

```

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

```
        from farfan_pipeline.core.phases.phase1_spc_ingestion import Phase1SPCIngestionContract

        phase0 = Phase0ValidationContract()
        phase1 = Phase1SPCIngestionContract()

        assert phase0.phase_name == "phase0_input_validation"
        assert phase1.phase_name == "phase1_spc_ingestion"

    def test_phase1_output_is_adapter_input(self):
        """Test Phase 1 output (CanonPolicyPackage) is adapter input."""
        from farfan_pipeline.core.phases.phase1_spc_ingestion import Phase1SPCIngestionContract
        from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract

        phase1 = Phase1SPCIngestionContract()
        adapter = AdapterContract()

        assert adapter.phase_name == "phase1_to_phase2_adapter"

    def test_adapter_output_is_phase2_input(self):
        """Test adapter output (PreprocessedDocument) is Phase 2 input."""
        from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract

        adapter = AdapterContract()
        assert "phase2" in adapter.phase_name.lower() or "adapter" in adapter.phase_name.lower()

    def test_no_phase_can_be_skipped(self):
        """Test orchestrator enforces sequential execution."""
        from farfan_pipeline.core.phases.phase_orchestrator import PhaseOrchestrator

        orchestrator = PhaseOrchestrator()
        assert hasattr(orchestrator, 'phase0')
        assert hasattr(orchestrator, 'phase1')
        assert hasattr(orchestrator, 'adapter')

    @pytest.mark.asyncio
    async def test_phase_contract_validates_input_output(self):
        """Test PhaseContract.run() validates input and output."""
        from farfan_pipeline.core.phases.phase_protocol import PhaseContract

        class TestContract(PhaseContract):
            def validate_input(self, data):
                from farfan_pipeline.core.phases.phase_protocol import ContractValidationResult
                return ContractValidationResult(True, "input", "test")

            def validate_output(self, data):
                from farfan_pipeline.core.phases.phase_protocol import ContractValidationResult
                return ContractValidationResult(True, "output", "test")

            async def execute(self, data):
                return data

        contract = TestContract("test")
        output, metadata = await contract.run("test_input")
        assert output == "test_input"
        assert metadata.success is True
```

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

```
        )
        assert prov.source_page == 5
        assert prov.source_section == "Section 2.1"
        assert prov.extraction_method == "semantic_chunking"

    def test_cpp_quality_metrics_has_provenance_completeness(self):
        """Test CanonPolicyPackage tracks provenance_completeness."""
        from farfan_pipeline.processing.models import (
            CanonPolicyPackage, ChunkGraph, QualityMetrics,
            IntegrityIndex, PolicyManifest
        )

        cpp = CanonPolicyPackage(
            schema_version="SPC-2025.1",
            chunk_graph=ChunkGraph(),
            quality_metrics=QualityMetrics(
                provenance_completeness=1.0,
                structural_consistency=0.95
            ),
            integrity_index=IntegrityIndex(blake2b_root="a"*64),
            metadata={}
        )
        assert cpp.quality_metrics.provenance_completeness == 1.0

    def test_all_chunks_should_have_provenance(self):
        """Test that chunks in a complete CPP have provenance data."""
        from farfan_pipeline.processing.models import (
            CanonPolicyPackage, ChunkGraph, Chunk, ChunkResolution,
            TextSpan, QualityMetrics, IntegrityIndex, PolicyManifest,
            ProvenanceMap
        )
        from farfan_pipeline.core.phases.phase1_spc_ingestion import POLICY_AREAS, DIMENSIONS

        chunk_graph = ChunkGraph()
        chunks_with_provenance = 0
        total_chunks = 0

        for i, pa in enumerate(POLICY_AREAS):
            for j, dim in enumerate(DIMENSIONS):
                chunk = Chunk(
                    id=f"c_{pa}_{dim}",
                    text=f"Test chunk {i*6+j}",
                    text_span=TextSpan(i*100, i*100+50),
                    resolution=ChunkResolution.MESO,
                    bytes_hash=hashlib.blake2b(f"t{i}{j}".encode()).hexdigest(),
                    policy_area_id=pa,
                    dimension_id=dim,
                    provenance=ProvenanceMap(
                        source_page=i+1,
                        source_section=f"Section {i+1}"
                    )
                )
                chunk_graph.add_chunk(chunk)
                total_chunks += 1
                if chunk.provenance is not None:
                    chunks_with_provenance += 1
```

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

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