

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 16
3: =====
4: Generated: 2025-12-07T06:17:22.889351
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: src/farfan_pipeline/core/orchestrator/verify_contracts.py
11: =====
12:
13: """Standalone contract verification utility for pre-execution validation.
14:
15: This module provides command-line and programmatic interfaces for verifying
16: all 30 base executor contracts (D1-Q1 through D6-Q5) before pipeline execution.
17:
18: Usage:
19:     # Verify all contracts with default class registry
20:     python -m farfan_pipeline.core.orchestrator.verify_contracts
21:
22:     # Verify with custom class registry
23:     python -m farfan_pipeline.core.orchestrator.verify_contracts --strict
24:
25:     # Programmatic usage
26:     from farfan_pipeline.core.orchestrator.verify_contracts import verify_all_contracts
27:     result = verify_all_contracts()
28:     if not result["passed"]:
29:         print(f"Validation failed: {result['errors']}")
30: """
31: from __future__ import annotations
32:
33: import argparse
34: import json
35: import logging
36: import sys
37: from typing import Any
38:
39: from farfan_pipeline.core.orchestrator.base_executor_with_contract import (
40:     BaseExecutorWithContract,
41: )
42:
43: logger = logging.getLogger(__name__)
44:
45:
46: def verify_all_contracts(
47:     class_registry: dict[str, type[object]] | None = None,
48:     strict: bool = True,
49:     verbose: bool = False,
50: ) -> dict[str, Any]:
51:     """Verify all 30 base executor contracts.
52:
53:     Args:
54:         class_registry: Optional class registry. If None, will build one.
55:         strict: If True, raise exception on any errors.
56:         verbose: If True, log detailed information.
```

```
57:
58:     Returns:
59:         Verification result dictionary with:
60:             - passed: bool
61:             - total_contracts: int
62:             - errors: list[str]
63:             - warnings: list[str]
64:             - verified_contracts: list[str]
65:
66:     Raises:
67:         RuntimeError: If strict=True and verification fails.
68:     """
69:     if verbose:
70:         logging.basicConfig(level=logging.INFO)
71:     else:
72:         logging.basicConfig(level=logging.WARNING)
73:
74:     logger.info("Starting contract verification for 30 base executors")
75:
76:     if class_registry is None:
77:         logger.info("Building class registry...")
78:     try:
79:         from farfan_pipeline.core.orchestrator.class_registry import (
80:             build_class_registry,
81:         )
82:         class_registry = build_class_registry()
83:         logger.info(f"Class registry built with {len(class_registry)} classes")
84:     except Exception as exc:
85:         logger.error(f"Failed to build class registry: {exc}")
86:         if strict:
87:             raise RuntimeError(f"Class registry construction failed: {exc}") from exc
88:         class_registry = None
89:
90:     result = BaseExecutorWithContract.verify_all_base_contracts(
91:         class_registry=class_registry
92:     )
93:
94:     logger.info(
95:         f"Verification complete: passed={result['passed']}, "
96:         f"verified={len(result['verified_contracts'])}/{result['total_contracts']}, "
97:         f"errors={len(result['errors'])}, warnings={len(result.get('warnings', []))}"
98:     )
99:
100:    if not result["passed"]:
101:        logger.error(f"Contract verification FAILED with {len(result['errors'])} errors")
102:        for error in result["errors"][:20]:
103:            logger.error(f"  - {error}")
104:
105:        if strict:
106:            raise RuntimeError(
107:                f"Contract verification failed with {len(result['errors'])} errors. "
108:                "See logs for details."
109:            )
110:
111:    if result.get("warnings"):
112:        logger.warning(f"Contract verification had {len(result['warnings'])} warnings")
```

```
113:         for warning in result["warnings"][:10]:
114:             logger.warning(f" - {warning}")
115:
116:     return result
117:
118:
119: def main() -> int:
120:     """Command-line entry point for contract verification.
121:
122:     Returns:
123:         Exit code: 0 if all contracts valid, 1 if any errors.
124:     """
125:     parser = argparse.ArgumentParser(
126:         description="Verify all 30 base executor contracts (D1-Q1 through D6-Q5)"
127:     )
128:     parser.add_argument(
129:         "--strict",
130:         action="store_true",
131:         help="Fail on any validation errors (default: False)",
132:     )
133:     parser.add_argument(
134:         "--verbose",
135:         "-v",
136:         action="store_true",
137:         help="Enable verbose logging",
138:     )
139:     parser.add_argument(
140:         "--json",
141:         action="store_true",
142:         help="Output results as JSON",
143:     )
144:     parser.add_argument(
145:         "--no-class-registry",
146:         action="store_true",
147:         help="Skip class registry validation (faster but incomplete)",
148:     )
149:
150:     args = parser.parse_args()
151:
152:     try:
153:         class_registry = None
154:         if not args.no_class_registry:
155:             from farfan_pipeline.core.orchestrator.class_registry import (
156:                 build_class_registry,
157:             )
158:             class_registry = build_class_registry()
159:
160:         result = verify_all_contracts(
161:             class_registry=class_registry,
162:             strict=args.strict,
163:             verbose=args.verbose,
164:         )
165:
166:         if args.json:
167:             print(json.dumps(result, indent=2))
168:         else:
```

```
169:         print(f"\n{'='*60}")
170:         print("CONTRACT VERIFICATION RESULTS")
171:         print(f"{'='*60}")
172:         print(f"Status: {'PASSED' if result['passed'] else 'FAILED'}")
173:         print(f"Verified: {len(result['verified_contracts'])}/{result['total_contracts']} contracts")
174:         print(f"Errors: {len(result['errors'])}")
175:         print(f"Warnings: {len(result.get('warnings', []))}")
176:
177:     if result["errors"]:
178:         print(f"\n{'='*60}")
179:         print("ERRORS")
180:         print(f"{'='*60}")
181:         for error in result["errors"][:20]:
182:             print(f"  - {error}")
183:         if len(result["errors"]) > 20:
184:             print(f"  ... and {len(result['errors']) - 20} more errors")
185:
186:     if result.get("warnings"):
187:         print(f"\n{'='*60}")
188:         print("WARNINGS")
189:         print(f"{'='*60}")
190:         for warning in result["warnings"][:10]:
191:             print(f"  - {warning}")
192:         if len(result["warnings"]) > 10:
193:             print(f"  ... and {len(result['warnings']) - 10} more warnings")
194:
195:     if result["verified_contracts"]:
196:         print(f"\n{'='*60}")
197:         print("VERIFIED CONTRACTS")
198:         print(f"{'='*60}")
199:         for i, contract in enumerate(result["verified_contracts"], 1):
200:             print(f"  {i:2d}. {contract}")
201:
202:     return 0 if result["passed"] else 1
203:
204: except Exception as exc:
205:     logger.error(f"Contract verification failed with exception: {exc}", exc_info=True)
206:     if args.json:
207:         print(json.dumps({"error": str(exc), "passed": False}))
208:     else:
209:         print(f"\nERROR: {exc}")
210:     return 1
211:
212:
213: if __name__ == "__main__":
214:     sys.exit(main())
215:
216:
217:
218: =====
219: FILE: src/farfan_pipeline/core/orchestrator/versions.py
220: =====
221:
222: """
223: Version Tracking for Contract Enforcement
224:
```

```
225: Centralized version management for all contract enforcement components.
226: Enables compatibility checking and rollback safety.
227: """
228:
229: # Pipeline version
230: PIPELINE_VERSION = "2.0.0"
231:
232: # Calibration version (from calibration_registry.py)
233: CALIBRATION_VERSION = "2.0.0" # Should match calibration_registry.CALIBRATION_VERSION
234:
235: # Signal version
236: SIGNAL_VERSION = "1.0.0"
237:
238: # Advanced module version
239: ADVANCED_MODULE_VERSION = "1.0.0"
240:
241: # Seed registry version
242: SEED_VERSION = "sha256_v1" # Should match seed_registry.SEED_VERSION
243:
244: # Verification manifest version
245: MANIFEST_VERSION = "1.0.0" # Should match verification_manifest.MANIFEST_VERSION
246:
247: # Minimum supported versions for backward compatibility
248: MIN_CALIBRATION_VERSION = "2.0.0"
249: MIN_SIGNAL_VERSION = "1.0.0"
250: MIN_PIPELINE_VERSION = "2.0.0"
251:
252:
253: def check_version_compatibility(component: str, version: str, min_version: str) -> bool:
254:     """
255:         Check if a version meets minimum requirements.
256:
257:     Args:
258:         component: Component name (for error messages)
259:         version: Current version string (e.g., "2.1.0")
260:         min_version: Minimum required version (e.g., "2.0.0")
261:
262:     Returns:
263:         True if version >= min_version
264:
265:     Raises:
266:         ValueError: If version is incompatible
267:     """
268:     try:
269:         v_parts = [int(x) for x in version.split(".")]
270:         min_parts = [int(x) for x in min_version.split(".")]
271:
272:         # Pad to same length
273:         while len(v_parts) < len(min_parts):
274:             v_parts.append(0)
275:         while len(min_parts) < len(v_parts):
276:             min_parts.append(0)
277:
278:         # Compare tuple
279:         if tuple(v_parts) < tuple(min_parts):
280:             raise ValueError(
```

```
281:             f"{{component}} version {version} is below minimum required {min_version}. "
282:             "Please upgrade or regenerate calibration data."
283:         )
284:
285:         return True
286:     except (ValueError, AttributeError) as e:
287:         if "below minimum" in str(e):
288:             raise
289:         raise ValueError(
290:             f"Invalid version format for {component}: {version}. "
291:             "Expected semantic version like '1.0.0'"
292:         ) from e
293:
294:
295: def get_all_versions() -> dict[str, str]:
296:     """
297:         Get all component versions for manifest inclusion.
298:
299:         Returns:
300:             Dictionary mapping component names to version strings
301:     """
302:     return {
303:         "pipeline": PIPELINE_VERSION,
304:         "calibration": CALIBRATION_VERSION,
305:         "signal": SIGNAL_VERSION,
306:         "advanced_module": ADVANCED_MODULE_VERSION,
307:         "seed": SEED_VERSION,
308:         "manifest": MANIFEST_VERSION,
309:     }
310:
311:
312:
313: =====
314: FILE: src/farfan_pipeline/core/parameters/__init__.py
315: =====
316:
317: """Centralized parameter loading system."""
318:
319: from farfan_pipeline.core.parameters.parameter_loader_v2 import ParameterLoaderV2
320:
321: __all__ = ["ParameterLoaderV2"]
322:
323:
324:
325: =====
326: FILE: src/farfan_pipeline/core/parameters/parameter_loader_v2.py
327: =====
328:
329: """ParameterLoaderV2: Centralized parameter loading from canonical catalogue."""
330:
331: import json
332: import logging
333: from pathlib import Path
334: from typing import Any
335:
336: logger = logging.getLogger(__name__)
```

```
337:
338:
339: class ParameterLoaderV2:
340:     """
341:         Centralized parameter loader that reads from canonical_method_catalogue_v2.json.
342:         Replaces scattered parameter_loader calls with single source of truth.
343:     """
344:
345:     _instance: "ParameterLoaderV2 | None" = None
346:     _catalogue: dict[str, dict[str, Any]] | None = None
347:     _catalogue_path: Path | None = None
348:
349:     def __new__(cls) -> "ParameterLoaderV2":
350:         if cls._instance is None:
351:             cls._instance = super().__new__(cls)
352:         return cls._instance
353:
354:     def __init__(self) -> None:
355:         if self._catalogue is None:
356:             self._load_catalogue()
357:
358:     def _load_catalogue(self) -> None:
359:         """Load canonical method catalogue from JSON file."""
360:         if self._catalogue_path is None:
361:             self._catalogue_path = (
362:                 Path(__file__).parent / "canonical_method_catalogue_v2.json"
363:             )
364:
365:         try:
366:             with open(self._catalogue_path, encoding="utf-8") as f:
367:                 data = json.load(f)
368:                 self._catalogue = data.get("methods", {})
369:                 logger.info(
370:                     f"Loaded {len(self._catalogue)} methods from canonical catalogue v{data['metadata']['version']}"
371:                 )
372:         except FileNotFoundError:
373:             logger.error(f"Canonical catalogue not found at {self._catalogue_path}")
374:             self._catalogue = {}
375:         except json.JSONDecodeError as e:
376:             logger.error(f"Failed to parse canonical catalogue: {e}")
377:             self._catalogue = {}
378:
379:     @classmethod
380:     def get(cls, method_id: str, param_name: str, default: Any = None) -> Any:
381:         """
382:             Get a specific parameter value for a method.
383:
384:             Args:
385:                 method_id: Fully qualified method identifier
386:                 param_name: Parameter name to retrieve
387:                 default: Default value if parameter not found
388:
389:             Returns:
390:                 Parameter value or default
391:             """
392:         instance = cls()
```

```
393:         if instance._catalogue is None:
394:             return default
395:
396:         method_params = instance._catalogue.get(method_id, {})
397:         return method_params.get(param_name, default)
398:
399:     @classmethod
400:     def get_all(cls, method_id: str) -> dict[str, Any]:
401:         """
402:             Get all parameters for a method.
403:
404:             Args:
405:                 method_id: Fully qualified method identifier
406:
407:             Returns:
408:                 Dictionary of all parameters for the method
409:         """
410:         instance = cls()
411:         if instance._catalogue is None:
412:             return {}
413:
414:         return instance._catalogue.get(method_id, {})
415:
416:     @classmethod
417:     def reload(cls) -> None:
418:         """Force reload of the catalogue from disk."""
419:         if cls._instance is not None:
420:             cls._instance._load_catalogue()
421:
422:
423:
424: =====
425: FILE: src/farfan_pipeline/core/phases/__init__.py
426: =====
427:
428: """
429: Canonical Phase Implementations - F.A.R.F.A.N Pipeline
430: =====
431:
432: This package contains the canonical implementations of all pipeline phases:
433:
434: Phase 0: Input Validation (Raw input → Validated CanonicalInput)
435: Phase 1: SPC Ingestion (CanonicalInput → CanonPolicyPackage)
436: Phase 2: Micro Questions (PreprocessedDocument → Phase2Result)
437: Phase 3: Chunk Routing (Phase2Result → Phase3Result)
438: Phase 6: Schema Validation (Raw data → Validated schemas)
439: Phase 7: Task Construction (Validated schemas → ExecutableTask set)
440: Phase 8: Execution Plan Assembly (ExecutableTask set → ExecutionPlan)
441:
442: All phases follow the PhaseContract protocol with:
443: - Explicit input/output contracts
444: - Mandatory field validation
445: - Error propagation semantics
446: - Deterministic execution
447: - Provenance tracking
448: """
```

```
449:  
450: from farfan_pipeline.core.phases.phase0_input_validation import (  
451:     CanonicalInput,  
452:     Phase0Input,  
453:     Phase0ValidationContract,  
454: )  
455: from farfan_pipeline.core.phases.phase1_spc_ingestion import (  
456:     Phase1SPCIgestionContract,  
457: )  
458: from farfan_pipeline.core.phases.phase2_types import (  
459:     Phase2QuestionResult,  
460:     Phase2Result,  
461:     validate_phase2_result,  
462: )  
463: from farfan_pipeline.core.phases.phase3_chunk_routing import (  
464:     ChunkRoutingResult,  
465:     Phase3ChunkRoutingContract,  
466:     Phase3Input,  
467:     Phase3Result,  
468: )  
469: from src.farfant_pipeline.core.phases.phase6_schema_validation import (  
470:     Phase6SchemaValidationOutput,  
471:     ValidatedChunkSchema,  
472:     ValidatedQuestionSchema,  
473:     phase6_schema_validation,  
474: )  
475: from src.farfant_pipeline.core.phases.phase7_task_construction import (  
476:     Phase7TaskConstructionOutput,  
477:     phase7_task_construction,  
478: )  
479: from src.farfant_pipeline.core.phases.phase8_execution_plan import (  
480:     ExecutionPlan,  
481:     Phase8ExecutionPlanOutput,  
482:     phase8_execution_plan_assembly,  
483: )  
484: from farfan_pipeline.core.phases.phase_orchestrator import (  
485:     PhaseOrchestrator,  
486:     PipelineResult,  
487: )  
488: from farfan_pipeline.core.phases.phase_protocol import (  
489:     ContractValidationResult,  
490:     PhaseArtifact,  
491:     PhaseContract,  
492:     PhaseInvariant,  
493:     PhaseManifestBuilder,  
494:     PhaseMetadata,  
495:     compute_contract_hash,  
496: )  
497:  
498: __all__ = [  
499:     # Phase 0  
500:     "CanonicalInput",  
501:     "Phase0Input",  
502:     "Phase0ValidationContract",  
503:     # Phase 1  
504:     "Phase1SPCIgestionContract",
```

```
505:     # Phase 2
506:     "Phase2QuestionResult",
507:     "Phase2Result",
508:     "validate_phase2_result",
509:     # Phase 3
510:     "ChunkRoutingResult",
511:     "Phase3ChunkRoutingContract",
512:     "Phase3Input",
513:     "Phase3Result",
514:     # Phase 6
515:     "Phase6SchemaValidationOutput",
516:     "ValidatedQuestionSchema",
517:     "ValidatedChunkSchema",
518:     "phase6_schema_validation",
519:     # Phase 7
520:     "Phase7TaskConstructionOutput",
521:     "phase7_task_construction",
522:     # Phase 8
523:     "ExecutionPlan",
524:     "Phase8ExecutionPlanOutput",
525:     "phase8_execution_plan_assembly",
526:     # Orchestrator
527:     "PhaseOrchestrator",
528:     "PipelineResult",
529:     # Protocol
530:     "ContractValidationResult",
531:     "PhaseArtifact",
532:     "PhaseContract",
533:     "PhaseInvariant",
534:     "PhaseManifestBuilder",
535:     "PhaseMetadata",
536:     "compute_contract_hash",
537: ]
538:
539:
540:
541: =====
542: FILE: src/farfan_pipeline/core/phases/phase0_input_validation.py
543: =====
544:
545: """
546: Phase 0: Input Validation - Constitutional Implementation
547: =====
548:
549: This module implements Phase 0 of the canonical pipeline:
550:     Raw input \206\222 Validated CanonicalInput
551:
552: Responsibilities:
553: -----
554: 1. Validate PDF exists and is readable
555: 2. Compute SHA256 hash of PDF (deterministic fingerprint)
556: 3. Extract PDF metadata (page count, size)
557: 4. Validate questionnaire exists
558: 5. Compute SHA256 hash of questionnaire
559: 6. Package validated inputs into CanonicalInput
560:
```

```
561: Input Contract:  
562: -----  
563: Phase0Input:  
564:     - pdf_path: Path (must exist)  
565:     - run_id: str (unique execution identifier)  
566:     - questionnaire_path: Path | None (optional, defaults to canonical)  
567:  
568: Output Contract:  
569: -----  
570: CanonicalInput:  
571:     - document_id: str (derived from PDF stem or explicit)  
572:     - run_id: str (preserved from input)  
573:     - pdf_path: Path (validated)  
574:     - pdf_sha256: str (computed hash)  
575:     - pdf_size_bytes: int (file size)  
576:     - pdf_page_count: int (extracted from PDF)  
577:     - questionnaire_path: Path (validated)  
578:     - questionnaire_sha256: str (computed hash)  
579:     - created_at: datetime (UTC timestamp)  
580:     - phase0_version: str (schema version)  
581:     - validation_passed: bool (must be True for output)  
582:     - validation_errors: list[str] (empty if passed)  
583:     - validation_warnings: list[str] (may contain warnings)  
584:  
585: Invariants:  
586: -----  
587: 1. validation_passed == True  
588: 2. pdf_page_count > 0  
589: 3. pdf_size_bytes > 0  
590: 4. pdf_sha256 is 64-char hex string  
591: 5. questionnaire_sha256 is 64-char hex string  
592:  
593: Author: F.A.R.F.A.N Architecture Team  
594: Date: 2025-01-19  
595: """  
596:  
597: from __future__ import annotations  
598:  
599: import hashlib  
600: from dataclasses import dataclass, field  
601: from datetime import datetime, timezone  
602: from pathlib import Path  
603: from typing import Any  
604:  
605: from pydantic import BaseModel, Field, field_validator  
606:  
607: from farfan_pipeline.core.phases.phase_protocol import (  
608:     ContractValidationResult,  
609:     PhaseContract,  
610: )  
611:  
612: # Schema version for Phase 0  
613: PHASE0_VERSION = "1.0.0"  
614:  
615:  
616: # =====
```

```
617: # INPUT CONTRACT
618: # =====
619:
620:
621: @dataclass
622: class Phase0Input:
623:     """
624:         Input contract for Phase 0.
625:
626:         This is the raw, unvalidated input to the pipeline.
627:     """
628:
629:     pdf_path: Path
630:     run_id: str
631:     questionnaire_path: Path | None = None
632:
633:
634: class Phase0InputValidator(BaseModel):
635:     """Pydantic validator for Phase0Input."""
636:
637:     pdf_path: str = Field(description="Path to input PDF")
638:     run_id: str = Field(min_length=1, description="Unique run identifier")
639:     questionnaire_path: str | None = Field(
640:         default=None, description="Optional questionnaire path"
641:     )
642:
643:     @field_validator("pdf_path")
644:     @classmethod
645:     def validate_pdf_path(cls, v: str) -> str:
646:         """Validate PDF path format."""
647:         if not v or not v.strip():
648:             raise ValueError("pdf_path cannot be empty")
649:         return v
650:
651:     @field_validator("run_id")
652:     @classmethod
653:     def validate_run_id(cls, v: str) -> str:
654:         """Validate run_id format."""
655:         if not v or not v.strip():
656:             raise ValueError("run_id cannot be empty")
657:         # Ensure run_id is filesystem-safe
658:         if any(char in v for char in ['/', '\\', ':', '*', '?', '"', '<', '>', '|']):
659:             raise ValueError(
660:                 "run_id contains invalid characters (must be filesystem-safe)"
661:             )
662:         return v
663:
664:
665: # =====
666: # OUTPUT CONTRACT
667: # =====
668:
669:
670: @dataclass
671: class CanonicalInput:
672:     """
```

```
673:     Output contract for Phase 0.
674:
675:     This represents a validated, canonical input ready for Phase 1.
676:     All fields are required and validated.
677:     """
678:
679:     # Identity
680:     document_id: str
681:     run_id: str
682:
683:     # Input artifacts (immutable, validated)
684:     pdf_path: Path
685:     pdf_sha256: str
686:     pdf_size_bytes: int
687:     pdf_page_count: int
688:
689:     # Questionnaire (required for SIN_CARRETA compliance)
690:     questionnaire_path: Path
691:     questionnaire_sha256: str
692:
693:     # Metadata
694:     created_at: datetime
695:     phase0_version: str
696:
697:     # Validation results
698:     validation_passed: bool
699:     validation_errors: list[str] = field(default_factory=list)
700:     validation_warnings: list[str] = field(default_factory=list)
701:
702:
703: class CanonicalInputValidator(BaseModel):
704:     """Pydantic validator for CanonicalInput."""
705:
706:     document_id: str = Field(min_length=1)
707:     run_id: str = Field(min_length=1)
708:     pdf_path: str
709:     pdf_sha256: str = Field(min_length=64, max_length=64)
710:     pdf_size_bytes: int = Field(gt=0)
711:     pdf_page_count: int = Field(gt=0)
712:     questionnaire_path: str
713:     questionnaire_sha256: str = Field(min_length=64, max_length=64)
714:     created_at: str
715:     phase0_version: str
716:     validation_passed: bool
717:     validation_errors: list[str] = Field(default_factory=list)
718:     validation_warnings: list[str] = Field(default_factory=list)
719:
720:     @field_validator("validation_passed")
721:     @classmethod
722:     def validate_passed(cls, v: bool, info) -> bool:
723:         """Ensure validation_passed is True and consistent with errors."""
724:         if not v:
725:             raise ValueError(
726:                 "validation_passed must be True for valid CanonicalInput"
727:             )
728:         errors = info.data.get("validation_errors", [])
```

```
729:         if errors:
730:             raise ValueError(
731:                 f"validation_passed is True but validation_errors is not empty: {errors}")
732:         )
733:     return v
734:
735:     @field_validator("pdf_sha256", "questionnaire_sha256")
736:     @classmethod
737:     def validate_sha256(cls, v: str) -> str:
738:         """Validate SHA256 hash format."""
739:         if len(v) != 64:
740:             raise ValueError(f"SHA256 hash must be 64 characters, got {len(v)}")
741:         if not all(c in "0123456789abcdef" for c in v.lower()):
742:             raise ValueError("SHA256 hash must be hexadecimal")
743:         return v.lower()
744:
745:
746: # =====
747: # PHASE 0 CONTRACT IMPLEMENTATION
748: # =====
749:
750:
751: class Phase0ValidationContract(PhaseContract[Phase0Input, CanonicalInput]):
752:     """
753:     Phase 0: Input Validation Contract.
754:
755:     This class enforces the constitutional constraint that Phase 0:
756:     1. Accepts ONLY Phase0Input
757:     2. Produces ONLY CanonicalInput
758:     3. Validates all invariants
759:     4. Logs all operations
760:     """
761:
762:     def __init__(self):
763:         """Initialize Phase 0 contract with invariants."""
764:         super().__init__(phase_name="phase0_input_validation")
765:
766:         # Register invariants
767:         self.addInvariant(
768:             name="validation_passed",
769:             description="Output must have validation_passed=True",
770:             check=lambda data: data.validation_passed is True,
771:             error_message="validation_passed must be True",
772:         )
773:
774:         self.addInvariant(
775:             name="pdf_page_count_positive",
776:             description="PDF must have at least 1 page",
777:             check=lambda data: data.pdf_page_count > 0,
778:             error_message="pdf_page_count must be > 0",
779:         )
780:
781:         self.addInvariant(
782:             name="pdf_size_positive",
783:             description="PDF size must be > 0 bytes",
784:             check=lambda data: data.pdf_size_bytes > 0,
```

```
785:             error_message="pdf_size_bytes must be > 0",
786:         )
787:
788:         self.addInvariant(
789:             name="sha256_format",
790:             description="SHA256 hashes must be valid",
791:             check=lambda data: (
792:                 len(data.pdf_sha256) == 64
793:                 and len(data.questionnaire_sha256) == 64
794:                 and all(c in "0123456789abcdef" for c in data.pdf_sha256.lower())
795:                 and all(c in "0123456789abcdef" for c in data.questionnaire_sha256.lower())
796:             ),
797:             error_message="SHA256 hashes must be 64-char hexadecimal",
798:         )
799:
800:         self.addInvariant(
801:             name="no_validation_errors",
802:             description="validation_errors must be empty",
803:             check=lambda data: len(data.validation_errors) == 0,
804:             error_message="validation_errors must be empty for valid output",
805:         )
806:
807:     def validate_input(self, input_data: Any) -> ContractValidationResult:
808:         """
809:             Validate Phase0Input contract.
810:
811:             Args:
812:                 input_data: Input to validate
813:
814:             Returns:
815:                 ContractValidationResult
816:         """
817:         errors = []
818:         warnings = []
819:
820:         # Type check
821:         if not isinstance(input_data, Phase0Input):
822:             errors.append(
823:                 f"Expected Phase0Input, got {type(input_data).__name__}"
824:             )
825:         return ContractValidationResult(
826:             passed=False,
827:             contract_type="input",
828:             phase_name=self.phase_name,
829:             errors=errors,
830:         )
831:
832:         # Validate using Pydantic
833:         try:
834:             Phase0InputValidator(
835:                 pdf_path=str(input_data.pdf_path),
836:                 run_id=input_data.run_id,
837:                 questionnaire_path=(
838:                     str(input_data.questionnaire_path)
839:                     if input_data.questionnaire_path
840:                     else None
```

```
841:         ),
842:     )
843: except Exception as e:
844:     errors.append(f"Pydantic validation failed: {e}")
845:
846: return ContractValidationResult(
847:     passed=len(errors) == 0,
848:     contract_type="input",
849:     phase_name=self.phase_name,
850:     errors=errors,
851:     warnings=warnings,
852: )
853:
854: def validate_output(self, output_data: Any) -> ContractValidationResult:
855:     """
856:         Validate CanonicalInput contract.
857:
858:     Args:
859:         output_data: Output to validate
860:
861:     Returns:
862:         ContractValidationResult
863:     """
864:     errors = []
865:     warnings = []
866:
867:     # Type check
868:     if not isinstance(output_data, CanonicalInput):
869:         errors.append(
870:             f"Expected CanonicalInput, got {type(output_data).__name__}"
871:         )
872:     return ContractValidationResult(
873:         passed=False,
874:         contract_type="output",
875:         phase_name=self.phase_name,
876:         errors=errors,
877:     )
878:
879:     # Validate using Pydantic
880: try:
881:     CanonicalInputValidator(
882:         document_id=output_data.document_id,
883:         run_id=output_data.run_id,
884:         pdf_path=str(output_data.pdf_path),
885:         pdf_sha256=output_data.pdf_sha256,
886:         pdf_size_bytes=output_data.pdf_size_bytes,
887:         pdf_page_count=output_data.pdf_page_count,
888:         questionnaire_path=str(output_data.questionnaire_path),
889:         questionnaire_sha256=output_data.questionnaire_sha256,
890:         created_at=output_data.created_at.isoformat(),
891:         phase0_version=output_data.phase0_version,
892:         validation_passed=output_data.validation_passed,
893:         validation_errors=output_data.validation_errors,
894:         validation_warnings=output_data.validation_warnings,
895:     )
896: except Exception as e:
```

```
897:         errors.append(f"Pydantic validation failed: {e}")
898:
899:     return ContractValidationResult(
900:         passed=len(errors) == 0,
901:         contract_type="output",
902:         phase_name=self.phase_name,
903:         errors=errors,
904:         warnings=warnings,
905:     )
906:
907:     async def execute(self, input_data: Phase0Input) -> CanonicalInput:
908:         """
909:             Execute Phase 0: Input Validation.
910:
911:             Args:
912:                 input_data: Phase0Input with raw paths
913:
914:             Returns:
915:                 CanonicalInput with validated data
916:
917:             Raises:
918:                 FileNotFoundError: If PDF or questionnaire doesn't exist
919:                 ValueError: If validation fails
920:
921:         errors = []
922:         warnings = []
923:
924:         # 1. Resolve questionnaire path
925:         questionnaire_path = input_data.questionnaire_path
926:         if questionnaire_path is None:
927:             from farfan_pipeline.config.paths import QUESTIONNAIRE_FILE
928:
929:             questionnaire_path = QUESTIONNAIRE_FILE
930:             warnings.append(
931:                 f"questionnaire_path not provided, using default: {questionnaire_path}"
932:             )
933:
934:         # 2. Validate PDF exists
935:         if not input_data.pdf_path.exists():
936:             errors.append(f"PDF not found: {input_data.pdf_path}")
937:         if not input_data.pdf_path.is_file():
938:             errors.append(f"PDF path is not a file: {input_data.pdf_path}")
939:
940:         # 3. Validate questionnaire exists
941:         if not questionnaire_path.exists():
942:             errors.append(f"Questionnaire not found: {questionnaire_path}")
943:         if not questionnaire_path.is_file():
944:             errors.append(f"Questionnaire path is not a file: {questionnaire_path}")
945:
946:         # If basic validation failed, abort
947:         if errors:
948:             raise FileNotFoundError(f"Input validation failed: {errors}")
949:
950:         # 4. Compute PDF hash and metadata
951:         pdf_sha256 = self._compute_sha256(input_data.pdf_path)
952:         pdf_size_bytes = input_data.pdf_path.stat().st_size
```

```
953:     pdf_page_count = self._get_pdf_page_count(input_data.pdf_path)
954:
955:     # 5. Compute questionnaire hash
956:     questionnaire_sha256 = self._compute_sha256(questionnaire_path)
957:
958:     # 6. Determine document_id
959:     document_id = input_data.pdf_path.stem
960:
961:     # 7. Create CanonicalInput
962:     canonical_input = CanonicalInput(
963:         document_id=document_id,
964:         run_id=input_data.run_id,
965:         pdf_path=input_data.pdf_path,
966:         pdf_sha256=pdf_sha256,
967:         pdf_size_bytes=pdf_size_bytes,
968:         pdf_page_count=pdf_page_count,
969:         questionnaire_path=questionnaire_path,
970:         questionnaire_sha256=questionnaire_sha256,
971:         created_at=datetime.now(timezone.utc),
972:         phase0_version=PHASE0_VERSION,
973:         validation_passed=len(errors) == 0,
974:         validation_errors=errors,
975:         validation_warnings=warnings,
976:     )
977:
978:     return canonical_input
979:
980: @staticmethod
981: def _compute_sha256(file_path: Path) -> str:
982:     """
983:     Compute SHA256 hash of a file.
984:
985:     Args:
986:         file_path: Path to file
987:
988:     Returns:
989:         Hex-encoded SHA256 hash (lowercase)
990:     """
991:     sha256_hash = hashlib.sha256()
992:     with open(file_path, "rb") as f:
993:         for byte_block in iter(lambda: f.read(4096), b""):
994:             sha256_hash.update(byte_block)
995:     return sha256_hash.hexdigest().lower()
996:
997: @staticmethod
998: def _get_pdf_page_count(pdf_path: Path) -> int:
999:     """
1000:     Extract page count from PDF.
1001:
1002:     Args:
1003:         pdf_path: Path to PDF file
1004:
1005:     Returns:
1006:         Number of pages
1007:
1008:     Raises:
```

```
1009:             ImportError: If PyMuPDF is not available
1010:             RuntimeError: If PDF cannot be opened
1011:             """
1012:             try:
1013:                 import fitz # PyMuPDF
1014:
1015:                 doc = fitz.open(pdf_path)
1016:                 page_count = len(doc)
1017:                 doc.close()
1018:                 return page_count
1019:             except ImportError:
1020:                 raise ImportError(
1021:                     "PyMuPDF (fitz) required for PDF page count extraction. "
1022:                     "Install with: pip install PyMuPDF"
1023:                 )
1024:             except Exception as e:
1025:                 raise RuntimeError(f"Failed to open PDF {pdf_path}: {e}")
1026:
1027:
1028:     __all__ = [
1029:         "Phase0Input",
1030:         "CanonicalInput",
1031:         "Phase0ValidationContract",
1032:         "PHASE0_VERSION",
1033:     ]
1034:
1035:
1036:
1037: =====
1038: FILE: src/farfan_pipeline/core/phases/phasel_models.py
1039: =====
1040:
1041: """
1042: Phase 1 Models - Strict Data Structures
1043: =====
1044:
1045: Data models for the Phase 1 SPC Ingestion Execution Contract.
1046: These models enforce strict typing and validation for the pipeline.
1047: """
1048:
1049: from __future__ import annotations
1050:
1051: from dataclasses import dataclass, field
1052: from typing import Any, Dict, List, Optional, Tuple
1053: from enum import Enum
1054:
1055: @dataclass
1056: class LanguageData:
1057:     """
1058:         Output of SPO - Language Detection.
1059:     """
1060:     primary_language: str
1061:     secondary_languages: List[str]
1062:     confidence_scores: Dict[str, float]
1063:     detection_method: str
1064:     normalized_text: Optional[str] = None
```

```
1065:     _sealed: bool = False
1066:
1067: @dataclass
1068: class PreprocessedDoc:
1069:     """
1070:     Output of SP1 - Advanced Preprocessing.
1071:     """
1072:     tokens: List[Any] = field(default_factory=list)
1073:     sentences: List[Any] = field(default_factory=list)
1074:     paragraphs: List[Any] = field(default_factory=list)
1075:     normalized_text: str = ""
1076:     original_to_normalized_mapping: Dict[Tuple[int, int], Tuple[int, int]] = field(default_factory=dict)
1077:     _hash: str = ""
1078:
1079: @dataclass
1080: class StructureData:
1081:     """
1082:     Output of SP2 - Structural Analysis.
1083:     """
1084:     sections: List[Any] = field(default_factory=list)
1085:     hierarchy: Dict[str, Optional[str]] = field(default_factory=dict)
1086:     paragraph_mapping: Dict[int, str] = field(default_factory=dict)
1087:     unassigned_paragraphs: List[int] = field(default_factory=list)
1088:     tables: List[Any] = field(default_factory=list)
1089:     lists: List[Any] = field(default_factory=list)
1090:
1091: @dataclass
1092: class KGNode:
1093:     """Node in the Knowledge Graph."""
1094:     id: str
1095:     type: str
1096:     text: str
1097:     signal_tags: List[str] = field(default_factory=list)
1098:     signal_importance: float = 0.0
1099:     policy_area_relevance: Dict[str, float] = field(default_factory=dict)
1100:
1101: @dataclass
1102: class KGEedge:
1103:     """Edge in the Knowledge Graph."""
1104:     source: str
1105:     target: str
1106:     type: str
1107:     weight: float = 1.0
1108:
1109: @dataclass
1110: class KnowledgeGraph:
1111:     """
1112:     Output of SP3 - Knowledge Graph Construction.
1113:     """
1114:     nodes: List[KGNode] = field(default_factory=list)
1115:     edges: List[KGEedge] = field(default_factory=list)
1116:     span_to_node_mapping: Dict[Tuple[int, int], str] = field(default_factory=dict)
1117:
1118: @dataclass
1119: class CausalGraph:
1120:     """Local causal graph for a chunk."""
```

```
1121:     events: List[Any] = field(default_factory=list)
1122:     causes: List[Any] = field(default_factory=list)
1123:     effects: List[Any] = field(default_factory=list)
1124:
1125: @dataclass
1126: class Chunk:
1127:     """
1128:     Intermediate chunk representation (SP4-SP10).
1129:     """
1130:     chunk_id: str = ""
1131:     policy_area_id: str = ""
1132:     dimension_id: str = ""
1133:     chunk_index: int = -1
1134:
1135:     text_spans: List[Tuple[int, int]] = field(default_factory=list)
1136:     sentence_ids: List[int] = field(default_factory=list)
1137:     paragraph_ids: List[int] = field(default_factory=list)
1138:
1139:     signal_tags: List[str] = field(default_factory=list)
1140:     signal_scores: Dict[str, float] = field(default_factory=dict)
1141:
1142:     overlap_flag: bool = False
1143:     segmentation_metadata: Dict[str, Any] = field(default_factory=dict)
1144:
1145:     # Enrichment fields (populated in SP5-SP10)
1146:     causal_graph: Optional[CausalGraph] = None
1147:     arguments: Optional[Dict[str, Any]] = None
1148:     temporal_markers: Optional[Dict[str, Any]] = None
1149:     discourse_mode: str = ""
1150:     rhetorical_strategies: List[str] = field(default_factory=list)
1151:     signal_patterns: List[str] = field(default_factory=list)
1152:
1153:     signal_weighted_importance: float = 0.0
1154:     policy_area_priority: float = 0.0
1155:     risk_weight: float = 0.0
1156:     governance_threshold: float = 0.0
1157:
1158: @dataclass
1159: class CausalChains:
1160:     """Output of SP5."""
1161:     chains: List[Any] = field(default_factory=list)
1162:
1163: @dataclass
1164: class IntegratedCausal:
1165:     """Output of SP6."""
1166:     global_graph: Any = None
1167:
1168: @dataclass
1169: class Arguments:
1170:     """Output of SP7."""
1171:     arguments_map: Dict[str, Any] = field(default_factory=dict)
1172:
1173: @dataclass
1174: class Temporal:
1175:     """Output of SP8."""
1176:     timeline: List[Any] = field(default_factory=list)
```

```
1177:  
1178: @dataclass  
1179: class Discourse:  
1180:     """Output of SP9."""  
1181:     patterns: Dict[str, Any] = field(default_factory=dict)  
1182:  
1183: @dataclass  
1184: class Strategic:  
1185:     """Output of SP10."""  
1186:     priorities: Dict[str, float] = field(default_factory=dict)  
1187:  
1188: @dataclass  
1189: class SmartChunk:  
1190:     """  
1191:     Final chunk representation (SP11-SP15).  
1192:     """  
1193:     policy_area_id: str = ""  
1194:     dimension_id: str = ""  
1195:     chunk_index: int = -1  
1196:  
1197:     causal_graph: CausalGraph = field(default_factory=CausalGraph)  
1198:     temporal_markers: Dict[str, Any] = field(default_factory=dict)  
1199:     arguments: Dict[str, Any] = field(default_factory=dict)  
1200:     discourse_mode: str = "unknown"  
1201:     strategic_rank: int = -1  
1202:     irrigation_links: List[Any] = field(default_factory=list)  
1203:  
1204:     signal_tags: List[str] = field(default_factory=list)  
1205:     signal_scores: Dict[str, float] = field(default_factory=dict)  
1206:     signal_version: str = "v1.0.0"  
1207:  
1208:     rank_score: float = 0.0  
1209:     signal_weighted_score: float = 0.0  
1210:  
1211: @dataclass  
1212: class ValidationResult:  
1213:     """Output of SP13 - Integrity Validation."""  
1214:     status: str = "INVALID"  
1215:     chunk_count: int = 0  
1216:     violations: List[str] = field(default_factory=list)  
1217:     pa_dim_coverage: str = "INCOMPLETE"  
1218:  
1219:  
1220:  
1221: ======  
1222: FILE: src/farfan_pipeline/core/phases/phase1_spc_ingestion.py  
1223: ======  
1224:  
1225: """  
1226: Phase 1: SPC Ingestion - Constitutional Implementation  
1227: ======  
1228:  
1229: This module implements Phase 1 of the canonical pipeline:  
1230:     CanonicalInput \206\222 CanonPolicyPackage  
1231:  
1232: Responsibilities:
```

```
1233: -----
1234: 1. Load and validate document from CanonicalInput
1235: 2. Execute 15 subfases of Strategic Chunking System
1236: 3. Generate exactly 60 chunks structured as 10 PA \227 6 DIM
1237: 4. Validate quality metrics and integrity
1238: 5. Package results into CanonPolicyPackage
1239:
1240: Input Contract:
1241: -----
1242: CanonicalInput (from Phase 0):
1243:   - document_id, pdf_path, pdf_sha256
1244:   - questionnaire_path
1245:   - All validation passed
1246:
1247: Output Contract:
1248: -----
1249: CanonPolicyPackage:
1250:   - schema_version: str ("SPC-2025.1")
1251:   - document_id: str (preserved from input)
1252:   - chunk_graph: ChunkGraph (60 chunks, PA\227DIM)
1253:   - policy_manifest: PolicyManifest
1254:   - quality_metrics: QualityMetrics
1255:   - integrity_index: IntegrityIndex
1256:   - metadata: dict
1257:
1258: 15 Subfases (Internal to Phase 1):
1259: -----
1260: Subfase 0: Language detection & model selection
1261: Subfase 1: Advanced preprocessing
1262: Subfase 2: Structural analysis & hierarchy extraction
1263: Subfase 3: Topic modeling & global KG construction
1264: Subfase 4: Structured (PA\227DIM) segmentation \206\222 60 chunks
1265: Subfase 5: Complete causal chain extraction
1266: Subfase 6: Causal integration
1267: Subfase 7: Deep argumentative analysis
1268: Subfase 8: Temporal and sequential analysis
1269: Subfase 9: Discourse and rhetorical analysis
1270: Subfase 10: Multi-scale strategic integration
1271: Subfase 11: Smart Policy Chunk generation
1272: Subfase 12: Inter-chunk relationship enrichment
1273: Subfase 13: Strategic integrity validation
1274: Subfase 14: Intelligent deduplication
1275: Subfase 15: Strategic importance ranking
1276:
1277: Invariants:
1278: -----
1279: 1. chunk_count == 60 (10 PA \227 6 DIM)
1280: 2. All chunks have policy_area_id (PA01-PA10)
1281: 3. All chunks have dimension_id (DIM01-DIM06)
1282: 4. quality_metrics.provenance_completeness >= 0.8
1283: 5. quality_metrics.structural_consistency >= 0.85
1284: 6. All chunks pass integrity validation
1285:
1286: Author: F.A.R.F.A.N Architecture Team
1287: Date: 2025-01-19
1288: """
```

```
1289:  
1290: from __future__ import annotations  
1291:  
1292: import logging  
1293: from dataclasses import dataclass, field  
1294: from pathlib import Path  
1295: from typing import Any  
1296:  
1297: from pydantic import BaseModel, Field, field_validator  
1298:  
1299: from farfan_pipeline.core.phases.phase_protocol import (  
1300:     ContractValidationResult,  
1301:     PhaseContract,  
1302: )  
1303: from farfan_pipeline.core.phases.phase0_input_validation import CanonicalInput  
1304: from farfan_pipeline.processing.cpp_ingestion.models import CanonPolicyPackage  
1305:  
1306: logger = logging.getLogger(__name__)  
1307:  
1308: # Schema version for Phase 1  
1309: PHASE1_VERSION = "SPC-2025.1"  
1310:  
1311: # Expected chunk count (10 PA \ 227 6 DIM)  
1312: EXPECTED_CHUNK_COUNT = 60  
1313:  
1314: # Policy Areas (PA01-PA10)  
1315: POLICY AREAS = [  
1316:     "PA01", # Mujeres y equidad de gÃ©nero  
1317:     "PA02", # Paz, seguridad y convivencia  
1318:     "PA03", # Ambiente y cambio climÃ¡tico  
1319:     "PA04", # Derechos econÃ³micos, sociales y culturales  
1320:     "PA05", # VÃ-ctimas y construcciÃ³n de paz  
1321:     "PA06", # NiÃ±ez, adolescencia y juventud  
1322:     "PA07", # Tierras y territorios  
1323:     "PA08", # LÃ-deres y defensores de DDHH  
1324:     "PA09", # Privadas de libertad  
1325:     "PA10", # MigraciÃ³n transfronteriza  
1326: ]  
1327:  
1328: # Dimensions (DIM01-DIM06)  
1329: DIMENSIONS = [  
1330:     "DIM01", # DiagnÃ³stico y recursos  
1331:     "DIM02", # Actividades e intervenciones  
1332:     "DIM03", # Productos (outputs)  
1333:     "DIM04", # Resultados (outcomes)  
1334:     "DIM05", # Impactos de largo plazo  
1335:     "DIM06", # Causalidad y teorÃ-a de cambio  
1336: ]  
1337:  
1338:  
1339: # ======  
1340: # SUBFASE TRACKING  
1341: # ======  
1342:  
1343:  
1344: @dataclass
```

```
1345: class SubfaseMetadata:
1346:     """Metadata for a single subfase execution."""
1347:
1348:     subfase_number: int
1349:     subfase_name: str
1350:     started_at: str
1351:     finished_at: str | None = None
1352:     duration_ms: float | None = None
1353:     success: bool = False
1354:     error: str | None = None
1355:
1356:
1357: # =====
1358: # OUTPUT CONTRACT VALIDATOR
1359: # =====
1360:
1361:
1362: class CanonPolicyPackageValidator(BaseModel):
1363:     """Pydantic validator for CanonPolicyPackage."""
1364:
1365:     schema_version: str = Field(description="Must be SPC-2025.1")
1366:     document_id: str = Field(min_length=1)
1367:     chunk_count: int = Field(ge=1, description="Number of chunks in chunk_graph")
1368:     has_policy_manifest: bool
1369:     has_quality_metrics: bool
1370:     has_integrity_index: bool
1371:     provenance_completeness: float = Field(ge=0.0, le=1.0)
1372:     structural_consistency: float = Field(ge=0.0, le=1.0)
1373:
1374:     @field_validator("schema_version")
1375:     @classmethod
1376:     def validate_schema_version(cls, v: str) -> str:
1377:         """Ensure schema version is correct."""
1378:         if v != PHASE1_VERSION:
1379:             raise ValueError(
1380:                 f"schema_version must be '{PHASE1_VERSION}', got '{v}'"
1381:             )
1382:         return v
1383:
1384:     @field_validator("chunk_count")
1385:     @classmethod
1386:     def validate_chunk_count(cls, v: int) -> int:
1387:         """Validate chunk count."""
1388:         if v != EXPECTED_CHUNK_COUNT:
1389:             raise ValueError(
1390:                 f"Expected {EXPECTED_CHUNK_COUNT} chunks (10 PA \u2227 6 DIM), got {v}"
1391:             )
1392:         return v
1393:
1394:     @field_validator("provenance_completeness")
1395:     @classmethod
1396:     def validate_provenance(cls, v: float) -> float:
1397:         """Ensure provenance completeness meets threshold."""
1398:         if v < 0.8:
1399:             raise ValueError(
1400:                 f"provenance_completeness must be >= 0.8, got {v:.2f}"
```

```
1401:         )
1402:     return v
1403:
1404:     @field_validator("structural_consistency")
1405:     @classmethod
1406:     def validate_structural(cls, v: float) -> float:
1407:         """Ensure structural consistency meets threshold."""
1408:         if v < 0.85:
1409:             raise ValueError(
1410:                 f"structural_consistency must be >= 0.85, got {v:.2f}")
1411:         )
1412:     return v
1413:
1414:
1415: # =====
1416: # PHASE 1 CONTRACT IMPLEMENTATION
1417: # =====
1418:
1419:
1420: class Phase1SPCIgestionContract(PhaseContract[CanonicalInput, CanonPolicyPackage]):
1421:     """
1422:     Phase 1: SPC Ingestion Contract.
1423:
1424:     This class enforces the constitutional constraint that Phase 1:
1425:     1. Accepts ONLY CanonicalInput (from Phase 0)
1426:     2. Produces ONLY CanonPolicyPackage
1427:     3. Executes all 15 subfases in order
1428:     4. Generates exactly 60 chunks (10 PA \227 6 DIM)
1429:     5. Validates all quality metrics
1430:
1431:     """
1432:     def __init__(self):
1433:         """Initialize Phase 1 contract with invariants."""
1434:         super().__init__(phase_name="phasel_spc_ingestion")
1435:
1436:         # Track subfases
1437:         self.subfases: list[SubfaseMetadata] = []
1438:
1439:         # Register invariants
1440:         self.addInvariant(
1441:             name="chunk_count_60",
1442:             description="Must have exactly 60 chunks (10 PA \227 6 DIM)",
1443:             check=lambda data: len(data.chunk_graph.chunks) == EXPECTED_CHUNK_COUNT,
1444:             error_message=f"chunk_count must be {EXPECTED_CHUNK_COUNT}",
1445:         )
1446:
1447:         self.addInvariant(
1448:             name="all_chunks_have_pa",
1449:             description="All chunks must have policy_area_id (PA01-PA10)",
1450:             check=lambda data: all(
1451:                 chunk.policy_area_id in POLICY AREAS
1452:                 for chunk in data.chunk_graph.chunks.values()
1453:             ),
1454:             error_message="All chunks must have valid policy_area_id",
1455:         )
1456:
```

```
1457:         self.add_invariant(
1458:             name="all_chunks_have_dim",
1459:             description="All chunks must have dimension_id (DIM01-DIM06)",
1460:             check=lambda data: all(
1461:                 chunk.dimension_id in DIMENSIONS
1462:                 for chunk in data.chunk_graph.chunks.values()
1463:             ),
1464:             error_message="All chunks must have valid dimension_id",
1465:         )
1466:
1467:         self.add_invariant(
1468:             name="provenance_threshold",
1469:             description="Provenance completeness >= 0.8",
1470:             check=lambda data: (
1471:                 data.quality_metrics is not None
1472:                 and data.quality_metrics.provenance_completeness >= 0.8
1473:             ),
1474:             error_message="provenance_completeness must be >= 0.8",
1475:         )
1476:
1477:         self.add_invariant(
1478:             name="structural_threshold",
1479:             description="Structural consistency >= 0.85",
1480:             check=lambda data: (
1481:                 data.quality_metrics is not None
1482:                 and data.quality_metrics.structural_consistency >= 0.85
1483:             ),
1484:             error_message="structural_consistency must be >= 0.85",
1485:         )
1486:
1487:     def validate_input(self, input_data: Any) -> ContractValidationResult:
1488:         """
1489:             Validate CanonicalInput contract.
1490:
1491:             Args:
1492:                 input_data: Input to validate
1493:
1494:             Returns:
1495:                 ContractValidationResult
1496:         """
1497:         errors = []
1498:         warnings = []
1499:
1500:         # Type check
1501:         if not isinstance(input_data, CanonicalInput):
1502:             errors.append(
1503:                 f"Expected CanonicalInput, got {type(input_data).__name__}"
1504:             )
1505:         return ContractValidationResult(
1506:             passed=False,
1507:             contract_type="input",
1508:             phase_name=self.phase_name,
1509:             errors=errors,
1510:         )
1511:
1512:         # Validate fields
```

```
1513:         if not input_data.validation_passed:
1514:             errors.append(
1515:                 f"CanonicalInput has validation_passed=False: {input_data.validation_errors}"
1516:             )
1517:
1518:         if not input_data.pdf_path.exists():
1519:             errors.append(f"PDF path does not exist: {input_data.pdf_path}")
1520:
1521:         if input_data.pdf_page_count <= 0:
1522:             errors.append(
1523:                 f"Invalid pdf_page_count: {input_data.pdf_page_count}"
1524:             )
1525:
1526:     return ContractValidationResult(
1527:         passed=len(errors) == 0,
1528:         contract_type="input",
1529:         phase_name=self.phase_name,
1530:         errors=errors,
1531:         warnings=warnings,
1532:     )
1533:
1534: def validate_output(self, output_data: Any) -> ContractValidationResult:
1535:     """
1536:         Validate CanonPolicyPackage contract.
1537:
1538:     Args:
1539:         output_data: Output to validate
1540:
1541:     Returns:
1542:         ContractValidationResult
1543:     """
1544:     errors = []
1545:     warnings = []
1546:
1547:     # Type check
1548:     if not isinstance(output_data, CanonPolicyPackage):
1549:         errors.append(
1550:             f"Expected CanonPolicyPackage, got {type(output_data).__name__}"
1551:         )
1552:     return ContractValidationResult(
1553:         passed=False,
1554:         contract_type="output",
1555:         phase_name=self.phase_name,
1556:         errors=errors,
1557:     )
1558:
1559:     # Validate using Pydantic
1560:     try:
1561:         CanonPolicyPackageValidator(
1562:             schema_version=output_data.schema_version,
1563:             document_id=output_data.metadata.get("document_id", ""),
1564:             chunk_count=len(output_data.chunk_graph.chunks),
1565:             has_policy_manifest=output_data.policy_manifest is not None,
1566:             has_quality_metrics=output_data.quality_metrics is not None,
1567:             has_integrity_index=output_data.integrity_index is not None,
1568:             provenance_completeness=(
```

```
1569:             output_data.quality_metrics.provenance_completeness
1570:             if output_data.quality_metrics
1571:                 else 0.0
1572:             ),
1573:             structural_consistency=(
1574:                 output_data.quality_metrics.structural_consistency
1575:                 if output_data.quality_metrics
1576:                     else 0.0
1577:                 ),
1578:             )
1579:         except Exception as e:
1580:             errors.append(f"Pydantic validation failed: {e}")
1581:
1582:     return ContractValidationResult(
1583:         passed=len(errors) == 0,
1584:         contract_type="output",
1585:         phase_name=self.phase_name,
1586:         errors=errors,
1587:         warnings=warnings,
1588:     )
1589:
1590:     async def execute(self, input_data: CanonicalInput) -> CanonPolicyPackage:
1591:         """
1592:             Execute Phase 1: SPC Ingestion with 15 subfases.
1593:
1594:             Args:
1595:                 input_data: CanonicalInput from Phase 0
1596:
1597:             Returns:
1598:                 CanonPolicyPackage with 60 chunks
1599:
1600:             Raises:
1601:                 ImportError: If SPC pipeline not available
1602:                 ValueError: If ingestion fails
1603:         """
1604:         logger.info(f"Starting Phase 1: SPC Ingestion for {input_data.document_id}")
1605:
1606:         # Import SPC pipeline
1607:         try:
1608:             from farfan_pipeline.processing.spc_ingestion import CPPIngestionPipeline
1609:         except ImportError as e:
1610:             raise ImportError(
1611:                 "SPC ingestion pipeline not available. "
1612:                 "Ensure farfan_core.processing.spc_ingestion is installed."
1613:             ) from e
1614:
1615:         # Initialize pipeline with questionnaire
1616:         pipeline = CPPIngestionPipeline(
1617:             questionnaire_path=input_data.questionnaire_path,
1618:             enable_runtime_validation=True,
1619:         )
1620:
1621:         # The pipeline.process() method internally executes all 15 subfases:
1622:         # Subfase 0: Language detection (in generate_smart_chunks)
1623:         # Subfase 1: Advanced preprocessing
1624:         # Subfase 2: Structural analysis
```

```
1625:     # Subfase 3: Topic modeling & KG
1626:     # Subfase 4: PAÄ\227DIM segmentation (60 chunks)
1627:     # Subfase 5: Causal chain extraction
1628:     # Subfase 6: Causal integration
1629:     # Subfase 7: Argumentative analysis
1630:     # Subfase 8: Temporal analysis
1631:     # Subfase 9: Discourse analysis
1632:     # Subfase 10: Strategic integration
1633:     # Subfase 11: Chunk generation
1634:     # Subfase 12: Inter-chunk enrichment
1635:     # Subfase 13: Integrity validation
1636:     # Subfase 14: Deduplication
1637:     # Subfase 15: Strategic ranking
1638:
1639:     logger.info("Executing 15 subfases of Strategic Chunking System...")
1640:
1641:     cpp = await pipeline.process(
1642:         document_path=input_data.pdf_path,
1643:         document_id=input_data.document_id,
1644:         title=input_data.pdf_path.name,
1645:         max_chunks=EXPECTED_CHUNK_COUNT,
1646:     )
1647:
1648:     logger.info(
1649:         f"Phase 1 complete: {len(cpp.chunk_graph.chunks)} chunks generated"
1650:     )
1651:
1652:     # Validate chunk count
1653:     actual_count = len(cpp.chunk_graph.chunks)
1654:     if actual_count != EXPECTED_CHUNK_COUNT:
1655:         logger.warning(
1656:             f"Expected {EXPECTED_CHUNK_COUNT} chunks, got {actual_count}. "
1657:             f"PAÄ\227DIM structure may be incomplete."
1658:         )
1659:
1660:     return cpp
1661:
1662:
1663: __all__ = [
1664:     "Phase1SPCIgestionContract",
1665:     "PHASE1_VERSION",
1666:     "EXPECTED_CHUNK_COUNT",
1667:     "POLICY AREAS",
1668:     "DIMENSIONS",
1669:     "SubfaseMetadata",
1670: ]
1671:
1672:
1673:
1674: =====
1675: FILE: src/farfan_pipeline/core/phases/phase1_spc_ingestion_full.py
1676: =====
1677:
1678: """
1679: Phase 1 SPC Ingestion - Full Execution Contract
1680: =====
```

```
1681:
1682: Implementation of the strict Phase 1 contract with zero ambiguity.
1683: """
1684:
1685: from __future__ import annotations
1686:
1687: import hashlib
1688: import json
1689: import logging
1690: import unicodedata
1691: import warnings
1692: from datetime import datetime
1693: from typing import Any, Dict, List, Optional, Tuple, Set
1694:
1695: from farfan_pipeline.core.phases.phase0_input_validation import CanonicalInput
1696: from farfan_pipeline.processing.cpp_ingestion.models import CanonPolicyPackage, ChunkGraph, PolicyManifest, QualityMetrics, IntegrityIndex
1697: from farfan_pipeline.core.phases.phase1_models import (
1698:     LanguageData, PreprocessedDoc, StructureData, KnowledgeGraph, KGNode, KGEEdge,
1699:     Chunk, CausalChains, IntegratedCausal, Arguments, Temporal, Discourse, Strategic,
1700:     SmartChunk, ValidationResult, CausalGraph
1701: )
1702:
1703: logger = logging.getLogger(__name__)
1704:
1705: class Phase1FatalError(Exception):
1706:     """Fatal error in Phase 1 execution."""
1707:     pass
1708:
1709: class Phase1MissionContract:
1710:     """
1711:         CRITICAL WEIGHT: 10000
1712:         FAILURE TO MEET ANY REQUIREMENT = IMMEDIATE PIPELINE TERMINATION
1713:         NO EXCEPTIONS, NO FALLBACKS, NO PARTIAL SUCCESS
1714:     """
1715:     # ... (Constants defined in spec, implicit in logic)
1716:
1717: class PADimGridSpecification:
1718:     """
1719:         WEIGHT: 10000 - NON-NEGOTIABLE GRID STRUCTURE
1720:         ANY DEVIATION = IMMEDIATE FAILURE
1721:     """
1722:
1723:     # IMMUTABLE CONSTANTS - DO NOT MODIFY
1724:     POLICY_AREAS = tuple([
1725:         "PA01",      # Index 0 - Economic policy domain (Using short codes for compatibility)
1726:         "PA02",      # Index 1 - Social policy domain
1727:         "PA03",      # Index 2 - Environmental policy domain
1728:         "PA04",      # Index 3 - Governance policy domain
1729:         "PA05",      # Index 4 - Infrastructure policy domain
1730:         "PA06",      # Index 5 - Security policy domain
1731:         "PA07",      # Index 6 - Technology policy domain
1732:         "PA08",      # Index 7 - Health policy domain
1733:         "PA09",      # Index 8 - Education policy domain
1734:         "PA10"       # Index 9 - Cultural policy domain
1735:     ])
1736:
```

```

1737:     DIMENSIONS = tuple([
1738:         "DIM01",          # What the policy aims to achieve
1739:         "DIM02",          # Tools and mechanisms used
1740:         "DIM03",          # How the policy is executed
1741:         "DIM04",          # Tracking and measurement
1742:         "DIM05",          # Risk assessment and mitigation
1743:         "DIM06"           # Expected and actual results
1744:     ])
1745:
1746:     # COMPUTED INVARIANTS
1747:     TOTAL_COMBINATIONS = len(POLICY AREAS) * len(DIMENSIONS) # MUST BE 60
1748:
1749:     @classmethod
1750:     def validate_chunk(cls, chunk: Any) -> None:
1751:         """
1752:             HARD VALIDATION - WEIGHT: 10000
1753:             EVERY CHECK MUST PASS OR PIPELINE DIES
1754:         """
1755:         # MANDATORY FIELD PRESENCE
1756:         assert hasattr(chunk, 'policy_area_id'), "FATAL: Missing policy_area_id"
1757:         assert hasattr(chunk, 'dimension_id'), "FATAL: Missing dimension_id"
1758:         assert hasattr(chunk, 'chunk_index'), "FATAL: Missing chunk_index"
1759:
1760:         # VALID VALUES
1761:         assert chunk.policy_area_id in cls.POLICY AREAS, \
1762:             f"FATAL: Invalid PA {chunk.policy_area_id}"
1763:         assert chunk.dimension_id in cls.DIMENSIONS, \
1764:             f"FATAL: Invalid DIM {chunk.dimension_id}"
1765:         assert 0 <= chunk.chunk_index < 60, \
1766:             f"FATAL: Invalid index {chunk.chunk_index}"
1767:
1768:         # MANDATORY METADATA - ALL MUST EXIST
1769:         REQUIRED_METADATA = [
1770:             'causal_graph',      # Causal relationships
1771:             'temporal_markers', # Time-based information
1772:             'arguments',        # Argumentative structure
1773:             'discourse_mode',   # Discourse classification
1774:             'strategic_rank',   # Strategic importance
1775:             'irrigation_links', # Inter-chunk connections
1776:             'signal_tags',      # Applied signals
1777:             'signal_scores',    # Signal strengths
1778:             'signal_version'    # Signal catalog version
1779:         ]
1780:
1781:         for field in REQUIRED_METADATA:
1782:             assert hasattr(chunk, field), f"FATAL: Missing {field}"
1783:             assert getattr(chunk, field) is not None, f"FATAL: Null {field}"
1784:
1785:     @classmethod
1786:     def validate_chunk_set(cls, chunks: List[Any]) -> None:
1787:         """
1788:             SET-LEVEL VALIDATION - WEIGHT: 10000
1789:         """
1790:         # EXACT COUNT
1791:         assert len(chunks) == 60, f"FATAL: Got {len(chunks)} chunks, need EXACTLY 60"
1792:

```

```
1793:     # UNIQUE COVERAGE
1794:     seen_combinations = set()
1795:     for chunk in chunks:
1796:         combo = (chunk.policy_area_id, chunk.dimension_id)
1797:         assert combo not in seen_combinations, f"FATAL: Duplicate {combo}"
1798:         seen_combinations.add(combo)
1799:
1800:     # COMPLETE COVERAGE
1801:     for pa in cls.POLICY AREAS:
1802:         for dim in cls.DIMENSIONS:
1803:             assert (pa, dim) in seen_combinations, f"FATAL: Missing {pa}\227{dim}"
1804:
1805: class Phase1FailureHandler:
1806: """
1807:     COMPREHENSIVE FAILURE HANDLING
1808:     NO SILENT FAILURES - EVERY ERROR MUST BE LOUD AND CLEAR
1809: """
1810:
1811:     @staticmethod
1812:     def handle_subphase_failure(sp_num: int, error: Exception) -> None:
1813:         """
1814:             HANDLE SUBPHASE FAILURE - ALWAYS FATAL
1815:         """
1816:         error_report = {
1817:             'phase': 'PHASE_1_SPC_INGESTION',
1818:             'subphase': f'SP{sp_num}',
1819:             'error_type': type(error).__name__,
1820:             'error_message': str(error),
1821:             'timestamp': datetime.utcnow().isoformat(),
1822:             'fatal': True,
1823:             'recovery_possible': False
1824:         }
1825:
1826:         # LOG TO ALL CHANNELS
1827:         logger.critical(f"FATAL ERROR IN PHASE 1, SUBPHASE {sp_num}")
1828:         logger.critical(f"ERROR TYPE: {error_report['error_type']}")
1829:         logger.critical(f"MESSAGE: {error_report['error_message']}")
1830:         logger.critical("PIPELINE TERMINATED")
1831:
1832:         # WRITE ERROR MANIFEST
1833:         try:
1834:             with open('phase1_error_manifest.json', 'w') as f:
1835:                 json.dump(error_report, f, indent=2)
1836:         except Exception:
1837:             pass # Best effort
1838:
1839:         # RAISE WITH FULL CONTEXT
1840:         raise Phase1FatalError(
1841:             f"Phase 1 failed at SP{sp_num}: {error}"
1842:         ) from error
1843:
1844:     @staticmethod
1845:     def validate_final_state(cpp: CanonPolicyPackage) -> bool:
1846: """
1847:     FINAL STATE VALIDATION - RETURN FALSE = PIPELINE DIES
1848: """
```

```

1849:     # Convert chunk_graph back to list for validation if needed, or iterate values
1850:     chunks = list(cpp.chunk_graph.chunks.values())
1851:
1852:     validations = {
1853:         'chunk_count_60': len(chunks) == 60,
1854:         '# mode_chunked': cpp.processing_mode == 'chunked', # Not in current CanonPolicyPackage model, skipping
1855:         'trace_complete': len(cpp.metadata.get('execution_trace', [])) == 16,
1856:         'results_complete': len(cpp.metadata.get('subphase_results', {})) == 16,
1857:         'chunks_valid': all(
1858:             hasattr(c, 'policy_area_id') and
1859:             hasattr(c, 'dimension_id')
1860:             # hasattr(c, 'strategic_rank') # Not in current Chunk model, stored in metadata or SmartChunk
1861:             for c in chunks
1862:         ),
1863:         'pa_dim_complete': len(set(
1864:             (c.policy_area_id, c.dimension_id)
1865:             for c in chunks
1866:         )) == 60
1867:     }
1868:
1869:     all_valid = all(validations.values())
1870:
1871:     if not all_valid:
1872:         logger.critical("PHASE 1 FINAL VALIDATION FAILED:")
1873:         for check, passed in validations.items():
1874:             if not passed:
1875:                 logger.critical(f" \u2192\234\227 {check} FAILED")
1876:
1877:     return all_valid
1878:
1879: class Phase1SPCIgestionFullContract:
1880:     """
1881:         CRITICAL EXECUTION CONTRACT - WEIGHT: 10000
1882:         EVERY LINE IS MANDATORY. NO SHORTCUTS. NO ASSUMPTIONS.
1883:     """
1884:
1885:     def __init__(self):
1886:         self.MANDATORY_SUBPHASES = list(range(16)) # SPO through SP15
1887:         self.execution_trace: List[Tuple[str, str, str]] = []
1888:         self.subphase_results: Dict[int, Any] = {}
1889:         self.error_log: List[Dict[str, Any]] = []
1890:         self.invariant_checks: Dict[str, bool] = {}
1891:
1892:     def _deterministic_serialize(self, output: Any) -> str:
1893:         """Helper to serialize output for hashing."""
1894:         # Simple string representation for now, can be improved
1895:         return str(output)
1896:
1897:     def _validate_canonical_input(self, canonical_input: CanonicalInput):
1898:         assert canonical_input.validation_passed, "Input validation failed"
1899:
1900:     def _assert_chunk_count(self, chunks: List[Any], count: int):
1901:         assert len(chunks) == count, f"Expected {count} chunks, got {len(chunks)}"
1902:
1903:     def _assert_smart_chunk_invariants(self, chunks: List[SmartChunk]):
1904:         PADimGridSpecification.validate_chunk_set(chunks)

```

```
1905:         for chunk in chunks:
1906:             PADimGridSpecification.validate_chunk(chunk)
1907:
1908:     def _assert_validation_pass(self, result: ValidationResult):
1909:         assert result.status == "VALID", f"Validation failed: {result.violations}"
1910:
1911:     def _handle_fatal_error(self, e: Exception):
1912:         Phase1FailureHandler.handle_subphase_failure(len(self.execution_trace), e)
1913:
1914:     def run(self, canonical_input: CanonicalInput) -> CanonPolicyPackage:
1915:         """
1916:             CRITICAL PATH - NO DEVIATIONS ALLOWED
1917:         """
1918:         # PRE-EXECUTION VALIDATION
1919:         self._validate_canonical_input(canonical_input) # WEIGHT: 1000
1920:
1921:         # SUBPHASE EXECUTION - EXACT ORDER MANDATORY
1922:         try:
1923:             # SP0: Language Detection - WEIGHT: 900
1924:             lang_data = self._execute_sp0_language_detection(canonical_input)
1925:             self._record_subphase(0, lang_data)
1926:
1927:             # SP1: Advanced Preprocessing - WEIGHT: 950
1928:             preprocessed = self._execute_sp1_preprocessing(canonical_input, lang_data)
1929:             self._record_subphase(1, preprocessed)
1930:
1931:             # SP2: Structural Analysis - WEIGHT: 950
1932:             structure = self._execute_sp2_structural(preprocessed)
1933:             self._record_subphase(2, structure)
1934:
1935:             # SP3: Topic Modeling & KG - WEIGHT: 980
1936:             knowledge_graph = self._execute_sp3_knowledge_graph(preprocessed, structure)
1937:             self._record_subphase(3, knowledge_graph)
1938:
1939:             # SP4: PA\227DIM Segmentation [CRITICAL: 60 CHUNKS] - WEIGHT: 10000
1940:             pa_dim_chunks = self._execute_sp4_segmentation(
1941:                 preprocessed, structure, knowledge_graph
1942:             )
1943:             self._assert_chunk_count(pa_dim_chunks, 60) # HARD STOP IF FAILS
1944:             self._record_subphase(4, pa_dim_chunks)
1945:
1946:             # SP5: Causal Chain Extraction - WEIGHT: 970
1947:             causal_chains = self._execute_sp5_causal_extraction(pa_dim_chunks)
1948:             self._record_subphase(5, causal_chains)
1949:
1950:             # SP6: Causal Integration - WEIGHT: 970
1951:             integrated_causal = self._execute_sp6_causal_integration(
1952:                 pa_dim_chunks, causal_chains
1953:             )
1954:             self._record_subphase(6, integrated_causal)
1955:
1956:             # SP7: Argumentative Analysis - WEIGHT: 960
1957:             arguments = self._execute_sp7_arguments(pa_dim_chunks, integrated_causal)
1958:             self._record_subphase(7, arguments)
1959:
1960:             # SP8: Temporal Analysis - WEIGHT: 960
```

```

1961:         temporal = self._execute_sp8_temporal(pa_dim_chunks, integrated_causal)
1962:         self._record_subphase(8, temporal)
1963:
1964:         # SP9: Discourse Analysis - WEIGHT: 950
1965:         discourse = self._execute_sp9_discourse(pa_dim_chunks, arguments)
1966:         self._record_subphase(9, discourse)
1967:
1968:         # SP10: Strategic Integration - WEIGHT: 990
1969:         strategic = self._execute_sp10_strategic(
1970:             pa_dim_chunks, integrated_causal, arguments, temporal, discourse
1971:         )
1972:         self._record_subphase(10, strategic)
1973:
1974:         # SP11: Smart Chunk Generation [CRITICAL: 60 CHUNKS] - WEIGHT: 10000
1975:         smart_chunks = self._execute_sp11_smart_chunks(
1976:             pa_dim_chunks, self.subphase_results
1977:         )
1978:         self._assert_smart_chunk_invariants(smart_chunks) # HARD STOP IF FAILS
1979:         self._record_subphase(11, smart_chunks)
1980:
1981:         # SP12: Inter-Chunk Enrichment - WEIGHT: 980
1982:         irrigated = self._execute_sp12_irrigation(smart_chunks)
1983:         self._record_subphase(12, irrigated)
1984:
1985:         # SP13: Integrity Validation [CRITICAL GATE] - WEIGHT: 10000
1986:         validated = self._execute_sp13_validation(irrigated)
1987:         self._assert_validation_pass(validated) # HARD STOP IF FAILS
1988:         self._record_subphase(13, validated)
1989:
1990:         # SP14: Deduplication - WEIGHT: 970
1991:         deduplicated = self._execute_sp14_deduplication(irrigated)
1992:         self._assert_chunk_count(deduplicated, 60) # HARD STOP IF FAILS
1993:         self._record_subphase(14, deduplicated)
1994:
1995:         # SP15: Strategic Ranking - WEIGHT: 990
1996:         ranked = self._execute_sp15_ranking(deduplicated)
1997:         self._record_subphase(15, ranked)
1998:
1999:         # FINAL CPP CONSTRUCTION WITH FULL VERIFICATION
2000:         canon_package = self._construct_cpp_with_verification(ranked)
2001:
2002:         # POSTCONDITION VERIFICATION - WEIGHT: 10000
2003:         self._verify_all_postconditions(canon_package)
2004:
2005:         return canon_package
2006:
2007:     except Exception as e:
2008:         self._handle_fatal_error(e)
2009:         raise Phase1FatalError(f"Phase 1 FAILED: {e}")
2010:
2011:     def _record_subphase(self, sp_num: int, output: Any):
2012:         """MANDATORY RECORDING - NO EXCEPTIONS"""
2013:         timestamp = datetime.utcnow().isoformat()
2014:         serialized = self._deterministic_serialize(output)
2015:         hash_value = hashlib.sha256(serialized.encode()).hexdigest()
2016:

```

```

2017:     self.execution_trace.append((f"SP{sp_num}", timestamp, hash_value))
2018:     self.subphase_results[sp_num] = output
2019:
2020:     # VERIFY RECORDING
2021:     assert len(self.execution_trace) == sp_num + 1
2022:     assert sp_num in self.subphase_results
2023:
2024:     # --- SUBPHASE IMPLEMENTATIONS ---
2025:
2026:     def _execute_sp0_language_detection(self, canonical_input: CanonicalInput) -> LanguageData:
2027:         # Placeholder logic - in real implementation, use langdetect or similar
2028:         return LanguageData(
2029:             primary_language="ES",
2030:             secondary_languages=[],
2031:             confidence_scores={"ES": 0.99},
2032:             detection_method="deterministic_mock",
2033:             _sealed=True
2034:         )
2035:
2036:     def _execute_sp1_preprocessing(self, canonical_input: CanonicalInput, lang_data: LanguageData) -> PreprocessedDoc:
2037:         # Placeholder logic
2038:         text = canonical_input.pdf_path.read_text(errors='ignore') if canonical_input.pdf_path.exists() else "MOCK TEXT"
2039:         normalized = unicodedata.normalize('NFC', text)
2040:         return PreprocessedDoc(
2041:             tokens=normalized.split(),
2042:             sentences=[s for s in normalized.split('.') if s],
2043:             paragraphs=[p for p in normalized.split('\n\n') if p],
2044:             normalized_text=normalized,
2045:             _hash=hashlib.sha256(normalized.encode()).hexdigest()
2046:         )
2047:
2048:     def _execute_sp2_structural(self, preprocessed: PreprocessedDoc) -> StructureData:
2049:         # Placeholder logic
2050:         return StructureData(
2051:             sections=["Section 1"],
2052:             hierarchy={"Section 1": None},
2053:             paragraph_mapping={i: "Section 1" for i in range(len(preprocessed.paragraphs))})
2054:     )
2055:
2056:     def _execute_sp3_knowledge_graph(self, preprocessed: PreprocessedDoc, structure: StructureData) -> KnowledgeGraph:
2057:         # Placeholder logic
2058:         return KnowledgeGraph(
2059:             nodes=[KGNode(id="node1", type="concept", text="policy")],
2060:             edges=[]
2061:         )
2062:
2063:     def _execute_sp4_segmentation(self, preprocessed: PreprocessedDoc, structure: StructureData, kg: KnowledgeGraph) -> List[Chunk]:
2064:         # CRITICAL: Generate exactly 60 chunks
2065:         chunks = []
2066:         idx = 0
2067:         for pa in PADimGridSpecification.POLICY AREAS:
2068:             for dim in PADimGridSpecification.DIMENSIONS:
2069:                 chunks.append(Chunk(
2070:                     chunk_id=f"{pa}_{dim}_CHUNK",
2071:                     policy_area_id=pa,
2072:                     dimension_id=dim,

```

```

2073:             chunk_index=idx,
2074:             signal_tags=[], 
2075:             signal_scores={}
2076:         ))
2077:         idx += 1
2078:     return chunks
2079:
2080: def _execute_sp5_causal_extraction(self, chunks: List[Chunk]) -> CausalChains:
2081:     for chunk in chunks:
2082:         chunk.causal_graph = CausalGraph()
2083:     return CausalChains()
2084:
2085: def _execute_sp6_causal_integration(self, chunks: List[Chunk], chains: CausalChains) -> IntegratedCausal:
2086:     return IntegratedCausal()
2087:
2088: def _execute_sp7_arguments(self, chunks: List[Chunk], integrated: IntegratedCausal) -> Arguments:
2089:     for chunk in chunks:
2090:         chunk.arguments = {}
2091:     return Arguments()
2092:
2093: def _execute_sp8_temporal(self, chunks: List[Chunk], integrated: IntegratedCausal) -> Temporal:
2094:     for chunk in chunks:
2095:         chunk.temporal_markers = {}
2096:     return Temporal()
2097:
2098: def _execute_sp9_discourse(self, chunks: List[Chunk], arguments: Arguments) -> Discourse:
2099:     for chunk in chunks:
2100:         chunk.discourse_mode = "narrative"
2101:     return Discourse()
2102:
2103: def _execute_sp10_strategic(self, chunks: List[Chunk], integrated: IntegratedCausal, arguments: Arguments, temporal: Temporal, discourse: Discourse) ->
Strategic:
2104:     for chunk in chunks:
2105:         chunk.strategic_rank = 0 # Placeholder
2106:     return Strategic()
2107:
2108: def _execute_sp11_smart_chunks(self, chunks: List[Chunk], enrichments: Dict[int, Any]) -> List[SmartChunk]:
2109:     smart_chunks = []
2110:     for chunk in chunks:
2111:         smart_chunks.append(SmartChunk(
2112:             policy_area_id=chunk.policy_area_id,
2113:             dimension_id=chunk.dimension_id,
2114:             chunk_index=chunk.chunk_index,
2115:             causal_graph=chunk.causal_graph or CausalGraph(),
2116:             temporal_markers=chunk.temporal_markers or {},
2117:             arguments=chunk.arguments or {},
2118:             discourse_mode=chunk.discourse_mode,
2119:             strategic_rank=0, # Will be updated in SP15
2120:             signal_tags=chunk.signal_tags,
2121:             signal_scores=chunk.signal_scores
2122:         ))
2123:     return smart_chunks
2124:
2125: def _execute_sp12_irrigation(self, chunks: List[SmartChunk]) -> List[SmartChunk]:
2126:     for chunk in chunks:
2127:         chunk.irrigation_links = []

```

```
2128:         return chunks
2129:
2130:     def _execute_sp13_validation(self, chunks: List[SmartChunk]) -> ValidationResult:
2131:         # Logic to validate chunks
2132:         return ValidationResult(
2133:             status="VALID",
2134:             chunk_count=len(chunks),
2135:             pa_dim_coverage="COMPLETE"
2136:         )
2137:
2138:     def _execute_sp14_deduplication(self, chunks: List[SmartChunk]) -> List[SmartChunk]:
2139:         return chunks # No-op for now as we generated unique chunks
2140:
2141:     def _execute_sp15_ranking(self, chunks: List[SmartChunk]) -> List[SmartChunk]:
2142:         for idx, chunk in enumerate(chunks):
2143:             chunk.strategic_rank = idx
2144:         return chunks
2145:
2146:     def _construct_cpp_with_verification(self, ranked: List[SmartChunk]) -> CanonPolicyPackage:
2147:         # Convert SmartChunks to internal Chunk model for CanonPolicyPackage
2148:         # This is a mapping step to satisfy the existing CanonPolicyPackage structure
2149:         from farfan_pipeline.processing.cpp_ingestion.models import Chunk as LegacyChunk, ChunkResolution, TextSpan
2150:
2151:         chunk_graph = ChunkGraph()
2152:         for sc in ranked:
2153:             legacy_chunk = LegacyChunk(
2154:                 id=f"{sc.policy_area_id}_{sc.dimension_id}",
2155:                 text="[CONTENT]", # Placeholder
2156:                 text_span=TextSpan(0, 0),
2157:                 resolution=ChunkResolution.MACRO,
2158:                 bytes_hash="hash",
2159:                 policy_area_id=sc.policy_area_id,
2160:                 dimension_id=sc.dimension_id
2161:             )
2162:             # Directly add to chunks dict to avoid calibration decorator issues
2163:             chunk_graph.chunks[legacy_chunk.id] = legacy_chunk
2164:
2165:         cpp = CanonPolicyPackage(
2166:             schema_version="SPC-2025.1",
2167:             chunk_graph=chunk_graph,
2168:             metadata={
2169:                 "execution_trace": self.execution_trace,
2170:                 "subphase_results": self.subphase_results # Note: This might be too large for metadata in prod
2171:             }
2172:         )
2173:         return cpp
2174:
2175:     def _verify_all_postconditions(self, cpp: CanonPolicyPackage):
2176:         assert len(cpp.chunk_graph.chunks) == 60
2177:         # Add more checks as needed
2178:
2179: def execute_phase_1_with_full_contract(canonical_input: CanonicalInput) -> CanonPolicyPackage:
2180:     """
2181:     EXECUTE PHASE 1 WITH COMPLETE CONTRACT ENFORCEMENT
2182:     THIS IS THE ONLY ACCEPTABLE WAY TO RUN PHASE 1
2183:     """
```

```
2184:     try:
2185:         # INITIALIZE EXECUTOR WITH FULL TRACKING
2186:         executor = Phase1SPCIgestionFullContract()
2187:
2188:         # RUN WITH COMPLETE VERIFICATION
2189:         cpp = executor.run(canonical_input)
2190:
2191:         # VALIDATE FINAL STATE
2192:         if not Phase1FailureHandler.validate_final_state(cpp):
2193:             raise Phase1FatalError("Final validation failed")
2194:
2195:         # SUCCESS - RETURN CPP
2196:         print(f"PHASE 1 COMPLETED: {len(cpp.chunk_graph.chunks)} chunks, "
2197:               f"{len(executor.execution_trace)} subphases")
2198:         return cpp
2199:
2200:     except Exception as e:
2201:         # NO RECOVERY - FAIL LOUD
2202:         print(f"PHASE 1 FATAL ERROR: {e}")
2203:         raise
2204:
2205:
2206:
2207:
2208: =====
2209: FILE: src/farfan_pipeline/core/phases/phase1_to_phase2_adapter/__init__.py
2210: =====
2211:
2212: """
2213: Phase 1 \206\222 Phase 2 Adapter Contract
2214: =====
2215:
2216: This module implements the adapter contract that transforms CanonPolicyPackage
2217: (Phase 1 output) into PreprocessedDocument (Phase 2 input).
2218:
2219: Responsibilities:
2220: -----
2221: 1. Convert 60 PA\227DIM chunks to sentences
2222: 2. Preserve chunk_id, policy_area_id, dimension_id in sentence_metadata.extra
2223: 3. Maintain chunk graph edges
2224: 4. Preserve all facets (policy, time, geo, entity, budget, KPI)
2225: 5. Validate preservation of critical metadata
2226:
2227: Input Contract:
2228: -----
2229: CanonPolicyPackage (from Phase 1):
2230:   - chunk_graph with 60 chunks
2231:   - Each chunk has policy_area_id (PA01-PA10)
2232:   - Each chunk has dimension_id (DIM01-DIM06)
2233:   - policy_manifest, quality_metrics, integrity_index
2234:
2235: Output Contract:
2236: -----
2237: PreprocessedDocument (for Phase 2):
2238:   - sentences: tuple[str] (one per chunk)
2239:   - sentence_metadata: tuple[SentenceMetadata]
```

```
2240:     - sentence_metadata[i].extra MUST contain:
2241:         - chunk_id: str
2242:         - policy_area_id: str (PA01-PA10)
2243:         - dimension_id: str (DIM01-DIM06)
2244:         - resolution: str
2245:         - policy_facets: dict
2246:         - time_facets: dict
2247:         - geo_facets: dict
2248:     - metadata: dict with quality_metrics, policy_manifest
2249:
2250: Invariants:
2251: -----
2252: 1. len(sentences) == 60 (one per chunk)
2253: 2. All sentence_metadata have chunk_id in extra
2254: 3. All sentence_metadata have policy_area_id in extra
2255: 4. All sentence_metadata have dimension_id in extra
2256: 5. processing_mode == "chunked"
2257:
2258: Author: F.A.R.F.A.N Architecture Team
2259: Date: 2025-01-19
2260: """
2261:
2262: from __future__ import annotations
2263:
2264: import logging
2265: from typing import Any
2266:
2267: from farfan_pipeline.core.types import PreprocessedDocument
2268: from farfan_pipeline.core.phases.phase_protocol import (
2269:     ContractValidationResult,
2270:     PhaseContract,
2271: )
2272: from farfan_pipeline.processing.cpp_ingestion.models import CanonPolicyPackage
2273:
2274: logger = logging.getLogger(__name__)
2275:
2276:
2277: def _meta_extra(meta: Any) -> dict[str, Any]:
2278:     """Extract extra metadata from sentence metadata entries."""
2279:     if isinstance(meta, dict):
2280:         return meta.get("extra", {}) or {}
2281:     if hasattr(meta, "extra"):
2282:         return getattr(meta, "extra") or {}
2283:     return {}
2284:
2285:
2286: class AdapterContract(PhaseContract[CanonPolicyPackage, PreprocessedDocument]):
2287:     """
2288:     Adapter contract enforcing PA\227DIM metadata preservation.
2289:
2290:     This contract ensures that the transformation from CanonPolicyPackage
2291:     to PreprocessedDocument preserves all critical chunk metadata needed
2292:     for Phase 2 question routing.
2293:     """
2294:
2295:     def __init__(self):
```

```
2296:     """Initialize adapter contract with invariants."""
2297:     super().__init__(phase_name="phasel_to_phase2_adapter")
2298:
2299:     # Invariant: All chunks preserved as sentences
2300:     self.addInvariant(
2301:         name="chunk_count_preserved",
2302:         description="All chunks must be preserved as sentences",
2303:         check=lambda data: len(data.sentences) > 0,
2304:         error_message="No sentences in PreprocessedDocument",
2305:     )
2306:
2307:     # Invariant: Processing mode is chunked
2308:     self.addInvariant(
2309:         name="processing_mode_chunked",
2310:         description="processing_mode must be 'chunked'",
2311:         check=lambda data: data.processing_mode == "chunked",
2312:         error_message="processing_mode must be 'chunked' for SPC adapter",
2313:     )
2314:
2315:     # Invariant: All sentence_metadata have chunk_id
2316:     self.addInvariant(
2317:         name="chunk_id_preserved",
2318:         description="All sentence_metadata must have chunk_id in extra",
2319:         check=lambda data: all(
2320:             "chunk_id" in _meta_extra(meta) for meta in data.sentence_metadata
2321:         ),
2322:         error_message="Missing chunk_id in sentence_metadata.extra",
2323:     )
2324:
2325:     # Invariant: All sentence_metadata have policy_area_id
2326:     self.addInvariant(
2327:         name="policy_area_id_preserved",
2328:         description="All sentence_metadata must have policy_area_id in extra",
2329:         check=lambda data: all(
2330:             "policy_area_id" in _meta_extra(meta) for meta in data.sentence_metadata
2331:         ),
2332:         error_message="Missing policy_area_id in sentence_metadata.extra - CRITICAL for Phase 2",
2333:     )
2334:
2335:     # Invariant: All sentence_metadata have dimension_id
2336:     self.addInvariant(
2337:         name="dimension_id_preserved",
2338:         description="All sentence_metadata must have dimension_id in extra",
2339:         check=lambda data: all(
2340:             "dimension_id" in _meta_extra(meta) for meta in data.sentence_metadata
2341:         ),
2342:         error_message="Missing dimension_id in sentence_metadata.extra - CRITICAL for Phase 2",
2343:     )
2344:
2345:     def validate_input(self, input_data: Any) -> ContractValidationResult:
2346:         """
2347:             Validate CanonPolicyPackage input.
2348:
2349:             Args:
2350:                 input_data: Input to validate
2351:
```

```
2352:     Returns:
2353:         ContractValidationResult
2354:     """
2355:     errors = []
2356:     warnings = []
2357:
2358:     # Type check
2359:     if not isinstance(input_data, CanonPolicyPackage):
2360:         errors.append(
2361:             f"Expected CanonPolicyPackage, got {type(input_data).__name__}")
2362:     )
2363:     return ContractValidationResult(
2364:         passed=False,
2365:         contract_type="input",
2366:         phase_name=self.phase_name,
2367:         errors=errors,
2368:     )
2369:
2370:     # Validate chunk_graph exists
2371:     if not hasattr(input_data, "chunk_graph") or not input_data.chunk_graph:
2372:         errors.append("CanonPolicyPackage missing chunk_graph")
2373:
2374:     # Validate chunks exist
2375:     if hasattr(input_data, "chunk_graph") and input_data.chunk_graph:
2376:         chunk_count = len(input_data.chunk_graph.chunks)
2377:         if chunk_count == 0:
2378:             errors.append("chunk_graph.chunks is empty")
2379:         elif chunk_count != 60:
2380:             warnings.append(
2381:                 f"Expected 60 chunks (10 PA \227 6 DIM), got {chunk_count}")
2382:         )
2383:
2384:     # Validate PA\227DIM tags present
2385:     missing_pa_dim = []
2386:     for chunk_id, chunk in input_data.chunk_graph.chunks.items():
2387:         if not hasattr(chunk, "policy_area_id") or not chunk.policy_area_id:
2388:             missing_pa_dim.append(f"{chunk_id}: missing policy_area_id")
2389:         if not hasattr(chunk, "dimension_id") or not chunk.dimension_id:
2390:             missing_pa_dim.append(f"{chunk_id}: missing dimension_id")
2391:
2392:         if missing_pa_dim:
2393:             errors.append(f"Chunks missing PA\227DIM tags: {missing_pa_dim[:5]$")
2394:
2395:     return ContractValidationResult(
2396:         passed=len(errors) == 0,
2397:         contract_type="input",
2398:         phase_name=self.phase_name,
2399:         errors=errors,
2400:         warnings=warnings,
2401:     )
2402:
2403: def validate_output(self, output_data: Any) -> ContractValidationResult:
2404:     """
2405:     Validate PreprocessedDocument output.
2406:
2407:     Args:
```

```
2408:         output_data: Output to validate
2409:
2410:     Returns:
2411:         ContractValidationResult
2412:     """
2413:     errors = []
2414:     warnings = []
2415:
2416:     # Type check
2417:     if not isinstance(output_data, PreprocessedDocument):
2418:         errors.append(
2419:             f"Expected PreprocessedDocument, got {type(output_data).__name__}"
2420:         )
2421:     return ContractValidationResult(
2422:         passed=False,
2423:         contract_type="output",
2424:         phase_name=self.phase_name,
2425:         errors=errors,
2426:     )
2427:
2428:     # Validate sentences exist
2429:     if not hasattr(output_data, "sentences") or not output_data.sentences:
2430:         errors.append("PreprocessedDocument.sentences is empty")
2431:
2432:     # Validate processing_mode
2433:     if (
2434:         not hasattr(output_data, "processing_mode")
2435:         or output_data.processing_mode != "chunked"
2436:     ):
2437:         errors.append(
2438:             f"processing_mode must be 'chunked', got '{getattr(output_data, 'processing_mode', None)}'"
2439:         )
2440:
2441:     # Validate sentence_metadata exists and matches sentences
2442:     if hasattr(output_data, "sentences") and hasattr(
2443:         output_data, "sentence_metadata"
2444:     ):
2445:         if len(output_data.sentence_metadata) != len(output_data.sentences):
2446:             errors.append(
2447:                 f"sentence_metadata count ({len(output_data.sentence_metadata)}) != "
2448:                 f"sentences count ({len(output_data.sentences)})"
2449:             )
2450:
2451:     # Validate PA\227DIM preservation in sentence_metadata
2452:     missing_metadata = []
2453:     for idx, meta in enumerate(output_data.sentence_metadata):
2454:         extra = _meta_extra(meta)
2455:         if not extra:
2456:             missing_metadata.append(f"sentence[{idx}]: no extra field")
2457:             continue
2458:
2459:         if "chunk_id" not in extra:
2460:             missing_metadata.append(f"sentence[{idx}]: missing chunk_id")
2461:         if "policy_area_id" not in extra:
2462:             missing_metadata.append(f"sentence[{idx}]: missing policy_area_id")
2463:         if "dimension_id" not in extra:
```

```
2464:             missing_metadata.append(f"sentence[{idx}]: missing dimension_id")
2465:
2466:         if missing_metadata:
2467:             errors.append(f"Metadata preservation failed: {missing_metadata[:5]}")
2468:
2469:     return ContractValidationResult(
2470:         passed=len(errors) == 0,
2471:         contract_type="output",
2472:         phase_name=self.phase_name,
2473:         errors=errors,
2474:         warnings=warnings,
2475:     )
2476:
2477:     async def execute(self, input_data: CanonPolicyPackage) -> PreprocessedDocument:
2478:         """
2479:             Execute adapter transformation.
2480:
2481:             Args:
2482:                 input_data: CanonPolicyPackage from Phase 1
2483:
2484:             Returns:
2485:                 PreprocessedDocument for Phase 2
2486:
2487:             Raises:
2488:                 ImportError: If SPCAdapter not available
2489:                 ValueError: If transformation fails
2490:         """
2491:         logger.info(f"Starting adapter: CanonPolicyPackage \u2061\u2062\u2062 PreprocessedDocument")
2492:
2493:         # Use existing SPCAdapter implementation
2494:         from farfan_pipeline.utils.spc_adapter import SPCAdapter
2495:
2496:         adapter = SPCAdapter(enable_runtime_validation=False) # We validate here
2497:
2498:         # Get document_id from metadata
2499:         document_id = input_data.metadata.get("document_id", "unknown")
2500:
2501:         # Transform
2502:         preprocessed = adapter.to_preprocessed_document(
2503:             input_data, document_id=document_id
2504:         )
2505:
2506:         logger.info(
2507:             f"Adapter complete: {len(preprocessed.sentences)} sentences, "
2508:             f"mode={preprocessed.processing_mode}"
2509:         )
2510:
2511:         return preprocessed
2512:
2513:
2514:     __all__ = [
2515:         "AdapterContract",
2516:     ]
```

```
2520: =====
2521: FILE: src/farfán_pipeline/core/phases/phase2_models.py
2522: =====
2523:
2524: import json
2525: from dataclasses import dataclass, field
2526: from typing import List, Dict, Any, Optional, Tuple
2527: from datetime import datetime
2528:
2529: # Import Phase 1 models for reference/typing
2530: from farfan_pipeline.core.phases.phase1_models import SmartChunk
2531: from farfan_pipeline.processing.cpp_ingestion.models import CanonPolicyPackage
2532:
2533: class FatalPhase2Error(Exception):
2534:     """Irrecoverable fatal error in Phase 2."""
2535:     pass
2536:
2537: @dataclass
2538: class Phase2MissionContract:
2539:     """
2540:         CRITICAL WEIGHT: 10000
2541:         PHASE 2 PROCESSES EXACTLY 300 MICRO QUESTIONS
2542:         ANY DEVIATION = IMMEDIATE PIPELINE TERMINATION
2543:     """
2544:
2545:     IMMUTABLE_CONSTANTS = {
2546:         "TOTAL_QUESTIONS": 300,
2547:         "CHUNKS_AVAILABLE": 60,
2548:         "DIMENSIONS": 6,
2549:         "POLICY AREAS": 10,
2550:         "QUESTIONS_PER_DIMENSION": 50, # 300/6
2551:         "QUESTIONS_PER_PA_PER_DIM": 5 # 50/10
2552:     }
2553:
2554:     PRIMARY_OBJECTIVES = {
2555:         "QUESTION_ANSWERING": {
2556:             "requirement": "Answer ALL 300 micro questions using 60 PA\227DIM chunks",
2557:             "hard_constraints": [
2558:                 "EXACTLY 300 questions answered",
2559:                 "STRICT dimension-first ordering",
2560:                 "STRICT policy-area scoping within dimension",
2561:                 "ALL evidence from Phase-1 CPP chunks",
2562:                 "ZERO re-computation of Phase-1 outputs",
2563:                 "FULL signal integration",
2564:                 "COMPLETE execution traceability"
2565:             ],
2566:             "weight": 10000,
2567:             "failure_mode": "FATAL_PIPELINE_ABORT"
2568:         },
2569:
2570:         "EVIDENCE_PIPELINE": {
2571:             "requirement": "Use canonical evidence from CPP chunks",
2572:             "hard_constraints": [
2573:                 "Evidence ONLY from PreprocessedDocument",
2574:                 "Evidence ONLY from 60 SmartPolicyChunks",
2575:                 "PA\227DIM isolation enforced",
```

```
2576:             "Signal-driven selection",
2577:             "Full provenance tracking",
2578:             "Zero hallucination"
2579:         ],
2580:         "weight": 10000,
2581:         "failure_mode": "FATAL_EVIDENCE_CORRUPTION"
2582:     },
2583:
2584:     "EXECUTION_ORDER": {
2585:         "requirement": "Dimension-first, PA-scoped processing",
2586:         "execution_sequence": [
2587:             "DIM1: PA1-10 questions (50 total)",
2588:             "DIM2: PA1-10 questions (50 total)",
2589:             "DIM3: PA1-10 questions (50 total)",
2590:             "DIM4: PA1-10 questions (50 total)",
2591:             "DIM5: PA1-10 questions (50 total)",
2592:             "DIM6: PA1-10 questions (50 total)"
2593:         ],
2594:         "weight": 10000,
2595:         "failure_mode": "FATAL_ORDER_VIOLATION"
2596:     }
2597: }
2598:
2599: FAILURE_CONDITIONS = {
2600:     "wrong_question_count": "questions_answered != 300 \u2061\u2062 PHASE 2 FAILED",
2601:     "orderViolation": "dimension_sequence broken \u2061\u2062 PHASE 2 FAILED",
2602:     "pa_contamination": "cross-PA evidence leak \u2061\u2062 PHASE 2 FAILED",
2603:     "phase1_recomputation": "ANY Phase-1 re-derivation \u2061\u2062 PHASE 2 FAILED",
2604:     "missing_evidence": "unanswerable question \u2061\u2062 PHASE 2 FAILED",
2605:     "signal_failure": "signal routing broken \u2061\u2062 PHASE 2 FAILED"
2606: }
2607:
2608: class Phase2OperatingContext:
2609: """
2610:     STRICT DEFINITION OF WHAT PHASE 2 CAN ACCESS
2611:     WEIGHT: 10000 - NO UNAUTHORIZED DATA ACCESS
2612: """
2613:
2614: AUTHORIZED_INPUTS = {
2615:     "cpp_from_phase1": {
2616:         "type": "CanonPolicyPackage",
2617:         "contents": [
2618:             "60 SmartPolicyChunk objects",
2619:             "chunk metadata (causal, temporal, args, discourse)",
2620:             "execution_trace from Phase 1",
2621:             "subphase_results[0:16]",
2622:             "signals_summary"
2623:         ],
2624:         "access_mode": "READ_ONLY",
2625:         "modification": "FORBIDDEN"
2626:     },
2627:
2628:     "preprocessed_document": {
2629:         "type": "PreprocessedDocument",
2630:         "contents": [
2631:             "sentence-level tokenization",
```

```
2632:             "paragraph structure",
2633:             "span mappings",
2634:             "linguistic annotations"
2635:         ],
2636:         "access_mode": "READ_ONLY",
2637:         "usage": "sentence-level evidence extraction"
2638:     },
2639:
2640:     "canonical_json": {
2641:         "type": "JSON Factory outputs",
2642:         "contents": [
2643:             "serialized chunks",
2644:             "metadata registries",
2645:             "signal states",
2646:             "evidence mappings"
2647:         ],
2648:         "access_mode": "READ_ONLY",
2649:         "requirement": "MUST use Factory, NEVER parse manually"
2650:     }
2651: }
2652:
2653: FORBIDDEN_OPERATIONS = [
2654:     "Re-running ANY Phase-1 subphase",
2655:     "Re-computing causal graphs",
2656:     "Re-extracting arguments",
2657:     "Re-doing temporal analysis",
2658:     "Re-calculating strategic ranks",
2659:     "Modifying chunk boundaries",
2660:     "Creating new chunks",
2661:     "Deleting existing chunks"
2662: ]
2663:
2664: @staticmethod
2665: def validate_data_access(operation: str, data_source: str) -> bool:
2666:     """
2667:     ENFORCE DATA ACCESS RULES
2668:     """
2669:     if operation in Phase2OperatingContext.FORBIDDEN_OPERATIONS:
2670:         raise FatalPhase2Error(f"FORBIDDEN: {operation}")
2671:
2672:     if data_source not in Phase2OperatingContext.AUTHORIZED_INPUTS:
2673:         raise FatalPhase2Error(f"UNAUTHORIZED DATA: {data_source}")
2674:
2675:     return True
2676:
2677: @dataclass
2678: class MicroQuestion:
2679:     question_id: str          # Unique identifier
2680:     dimension_id: str          # D1-D6
2681:     policy_area_id: str        # PA01-PA10
2682:     question_text: str         # Actual question
2683:     question_type: str         # Type/template
2684:     required_evidence: List[str] # Evidence types needed
2685:     signal_requirements: Dict[str, float] # Min signal thresholds
2686:
2687: @dataclass
```

```
2688: class Evidence:
2689:     evidence_id: str
2690:     source_chunk_id: str
2691:     policy_area: str
2692:     dimension: str
2693:     evidence_type: str
2694:     content: Any
2695:     provenance: Dict[str, Any]
2696:
2697: @dataclass
2698: class Signal:
2699:     signal_id: str
2700:     signal_type: str
2701:     value: float = 0.0
2702:     metadata: Dict = field(default_factory=dict)
2703:     computation_trace: List[str] = field(default_factory=list)
2704:
2705:     def __init__(self, signal_id: str, signal_type: str):
2706:         self.signal_id = signal_id
2707:         self.signal_type = signal_type
2708:         self.value = 0.0
2709:         self.metadata = {}
2710:         self.computation_trace = []
2711:
2712: @dataclass
2713: class ExecutorConfig:
2714:     executor_id: str
2715:     executor_type: str
2716:     supported_dimensions: List[str]
2717:     supported_policy_areas: List[str]
2718:     required_evidence_types: List[str]
2719:     method_whitelist: List[str] # From MethodRegistry
2720:     signal_requirements: Dict[str, float]
2721:     version: str
2722:
2723: @dataclass
2724: class ExecutedContract:
2725:     """
2726:     RUNTIME RECORD OF EXECUTION
2727:     """
2728:     contract_id: str
2729:     executor_id: str
2730:     question_id: str
2731:     input_evidence_ids: List[str]
2732:     input_signals: Dict[str, float]
2733:     methods_called: List[str]
2734:     output_answer: str
2735:     output_confidence: float
2736:     output_metrics: Dict[str, Any]
2737:     execution_timestamp: str
2738:     execution_duration_ms: int
2739:
2740: @dataclass
2741: class Phase2Results:
2742:     total_questions_answered: int = 0
2743:     execution_log: List[Dict] = field(default_factory=list)
```

```
2744:     answers: Dict[str, str] = field(default_factory=dict)
2745:     dimension_coverage: Dict[str, int] = field(default_factory=dict)
2746:     pa_coverage: Dict[str, int] = field(default_factory=dict)
2747:     total_evidence_used: int = 0
2748:     avg_confidence: float = 0.0
2749:     verification: Dict[str, Any] = field(default_factory=dict)
2750:     verification_manifest: Dict[str, Any] = field(default_factory=dict)
2751:
2752:
2753:
2754: =====
2755: FILE: src/farfan_pipeline/core/phases/phase2_types.py
2756: =====
2757:
2758: """
2759: Types for Phase 2 (Microquestions)
2760: =====
2761:
2762: This module defines the canonical data structures for the output of
2763: Phase 2, which involves processing the PreprocessedDocument to generate
2764: a series of microquestions and their answers.
2765:
2766: These types are used by the PhaseOrchestrator to validate and record
2767: the results of the core orchestrator's execution.
2768:
2769: Author: F.A.R.F.A.N Architecture Team
2770: Date: 2025-11-21
2771: """
2772:
2773: from __future__ import annotations
2774:
2775: from dataclasses import dataclass
2776: from typing import Any, List, Tuple
2777:
2778:
2779: @dataclass
2780: class Phase2QuestionResult:
2781:     """
2782:         Represents the result for a single microquestion generated
2783:         and answered during Phase 2.
2784:     """
2785:
2786:     base_slot: str
2787:     question_id: str
2788:     question_global: int | None
2789:     policy_area_id: str | None = None
2790:     dimension_id: str | None = None
2791:     cluster_id: str | None = None
2792:     evidence: dict[str, Any] | None = None
2793:     validation: dict[str, Any] | None = None
2794:     trace: dict[str, Any] | None = None
2795:     metadata: dict[str, Any] | None = None
2796:     human_readable_output: str | None = None # Added for v3 contracts
2797:
2798:
2799: @dataclass
```

```
2800: class Phase2Result:
2801:     """
2802:         Represents the complete output of Phase 2, containing all
2803:         the generated microquestions.
2804:         """
2805:
2806:     questions: List[Phase2QuestionResult]
2807:
2808:
2809: def _extract_questions(result: Any) -> tuple[List[Any] | None, list[str]]:
2810:     """Normalize Phase 2 result into a list of question dicts if possible."""
2811:     errors: list[str] = []
2812:     if result is None:
2813:         errors.append("Phase 2 returned no data")
2814:     return None, errors
2815:
2816:     if isinstance(result, dict) and "questions" in result:
2817:         questions = result.get("questions")
2818:     elif hasattr(result, "questions"):
2819:         questions = getattr(result, "questions")
2820:     else:
2821:         errors.append("Phase 2 result missing 'questions'")
2822:     return None, errors
2823:
2824:     if not isinstance(questions, list):
2825:         errors.append("Phase 2 questions must be a list")
2826:     return None, errors
2827:
2828:     return questions, errors
2829:
2830:
2831: def validate_phase2_result(result: Any) -> Tuple[bool, list[str], List[dict[str, Any]] | None]:
2832:     """
2833:         Validate the structure of a Phase 2 result.
2834:
2835:         Returns:
2836:             Tuple of (is_valid, errors, normalized_questions)
2837:             """
2838:     questions, errors = _extract_questions(result)
2839:     if questions is None:
2840:         return False, errors, None
2841:
2842:     if len(questions) == 0:
2843:         errors.append("Phase 2 questions list is empty or missing")
2844:     return False, errors, questions
2845:
2846:     normalized: list[dict[str, Any]] = []
2847:     for idx, q in enumerate(questions):
2848:         if not isinstance(q, dict):
2849:             errors.append(f"Question {idx} must be a dict")
2850:             continue
2851:
2852:         required_keys = ["base_slot", "question_id", "question_global", "evidence", "validation"]
2853:         missing = [key for key in required_keys if q.get(key) is None]
2854:         if missing:
2855:             errors.append(f"Question {idx} missing keys: {', '.join(missing)}")
```

```
2856:  
2857:         normalized.append(q)  
2858:  
2859:     return len(errors) == 0, errors, normalized  
2860:  
2861:  
2862: __all__ = [  
2863:     "Phase2QuestionResult",  
2864:     "Phase2Result",  
2865:     "validate_phase2_result",  
2866: ]  
2867:  
2868:  
2869:  
2870: ======  
2871: FILE: src/farfan_pipeline/core/phases/phase3_chunk_routing.py  
2872: ======  
2873:  
2874: """  
2875: Phase 3: Chunk Routing  
2876: ======  
2877:  
2878: This module implements Phase 3 of the F.A.R.F.A.N pipeline, which routes  
2879: questions to their corresponding policy area and dimension chunks.  
2880:  
2881: Phase Structure:  
2882: -----  
2883: Phase 3 follows the established hierarchical structure:  
2884:  
2885: 1. Sequential Dependency Root (Input Extraction):  
2886:     - Extract questions from Phase 2 result  
2887:     - Extract chunk matrix from PreprocessedDocument  
2888:     - Validate input contracts  
2889:  
2890: 2. Validation Stage:  
2891:     - Verify question structure (policy_area_id, dimension_id presence)  
2892:     - Verify chunk matrix completeness (60 chunks, PA\227DIM coverage)  
2893:     - Validate lookup key formats  
2894:  
2895: 3. Transformation Stage:  
2896:     - Convert question dimension to DIM_ID format (D1 \206\222 DIM01)  
2897:     - Construct lookup keys (policy_area_id, dimension_id)  
2898:     - Perform chunk routing via matrix lookup  
2899:  
2900: 4. Error Conditions (Leaf Nodes):  
2901:     - Missing policy_area_id or dimension_id in question \206\222 ValueError  
2902:     - Chunk not found for (PA, DIM) key \206\222 ValueError with descriptive message  
2903:     - Chunk metadata mismatch \206\222 ValueError  
2904:  
2905: 5. Observability:  
2906:     - Log routing outcomes (match counts)  
2907:     - Log policy area distribution  
2908:     - Record routing failures  
2909:  
2910: Design Principles:  
2911: -----
```

```
2912: - **Strict Equality Enforcement**: policy_area_id and dimension_id must match exactly
2913: - **Complete Field Population**: All 7 canonical ChunkRoutingResult fields populated
2914: - **Descriptive Errors**: ValueError exceptions identify question and failure reason
2915: - **No Task Creep**: Focus solely on core routing correctness
2916: - **Deterministic**: Same inputs produce same routing outcomes
2917:
2918: Author: F.A.R.F.A.N Architecture Team
2919: Date: 2025-01-22
2920: """
2921:
2922: from __future__ import annotations
2923:
2924: import logging
2925: from dataclasses import dataclass, field
2926: from typing import Any
2927:
2928: from farfan_pipeline.core.phases.phase_protocol import (
2929:     ContractValidationResult,
2930:     PhaseContract,
2931: )
2932: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
2933:
2934: logger = logging.getLogger(__name__)
2935:
2936:
2937: @dataclass
2938: class ChunkRoutingResult:
2939:     """Result of routing a single question to its target chunk.
2940:
2941:     This dataclass contains all seven canonical fields required for
2942:     Phase 3 chunk routing verification.
2943:     """
2944:
2945:     target_chunk: ChunkData
2946:     chunk_id: str
2947:     policy_area_id: str
2948:     dimension_id: str
2949:     text_content: str
2950:     expected_elements: list[dict[str, Any]]
2951:     document_position: tuple[int, int] | None
2952:
2953:     def __post_init__(self) -> None:
2954:         """Validate that all required fields are properly populated."""
2955:         if self.target_chunk is None:
2956:             raise ValueError("target_chunk cannot be None")
2957:         if not self.chunk_id:
2958:             raise ValueError("chunk_id cannot be empty")
2959:         if not self.policy_area_id:
2960:             raise ValueError("policy_area_id cannot be empty")
2961:         if not self.dimension_id:
2962:             raise ValueError("dimension_id cannot be empty")
2963:         if not self.text_content:
2964:             raise ValueError("text_content cannot be empty")
2965:         if self.expected_elements is None:
2966:             raise ValueError("expected_elements cannot be None (use empty list [])")
2967:
```

```
2968:
2969: @dataclass
2970: class Phase3Input:
2971:     """Input contract for Phase 3: chunk routing.
2972:
2973:     Contains the preprocessed document with chunk matrix and
2974:     the questions from Phase 2 that need routing.
2975:     """
2976:
2977:     preprocessed_document: PreprocessedDocument
2978:     questions: list[dict[str, Any]]
2979:
2980:
2981: @dataclass
2982: class Phase3Result:
2983:     """Output contract for Phase 3: chunk routing results.
2984:
2985:     Contains routing results for all questions and observability metrics.
2986:     """
2987:
2988:     routing_results: list[ChunkRoutingResult]
2989:     total_questions: int
2990:     successful_routes: int
2991:     failed_routes: int
2992:     policy_area_distribution: dict[str, int] = field(default_factory=dict)
2993:     dimension_distribution: dict[str, int] = field(default_factory=dict)
2994:     routing_errors: list[str] = field(default_factory=list)
2995:
2996:
2997: class Phase3ChunkRoutingContract(PhaseContract[Phase3Input, Phase3Result]):
2998:     """
2999:     Phase 3 Contract: Chunk Routing with Strict PA\227DIM Enforcement.
3000:
3001:     This phase routes each question to its corresponding chunk in the
3002:     60-chunk PA\227DIM matrix, enforcing strict equality between question
3003:     and chunk identifiers.
3004:     """
3005:
3006:     def __init__(self):
3007:         """Initialize Phase 3 contract with validation invariants."""
3008:         super().__init__("phase3_chunk_routing")
3009:
3010:         # Add invariants for Phase 3
3011:         self.addInvariant(
3012:             name="routing_completeness",
3013:             description="All questions must be either successfully routed or have an error recorded",
3014:             check=lambda result: result.successful_routes + result.failed_routes == result.total_questions,
3015:             error_message="Routing count mismatch: some questions were neither routed nor recorded as failures"
3016:         )
3017:
3018:         self.addInvariant(
3019:             name="routing_results_match_success",
3020:             description="Number of routing results must match successful routes",
3021:             check=lambda result: len(result.routing_results) == result.successful_routes,
3022:             error_message="Routing results count does not match successful_routes count"
3023:         )
```

```
3024:  
3025:         self.add_invariant(  
3026:             name="policy_area_distribution_sum",  
3027:             description="Policy area distribution must sum to successful routes",  
3028:             check=lambda result: sum(result.policy_area_distribution.values()) == result.successful_routes,  
3029:             error_message="Policy area distribution counts do not sum to successful_routes"  
3030:         )  
3031:  
3032:     def validate_input(self, input_data: Any) -> ContractValidationResult:  
3033:         """Validate Phase 3 input contract.  
3034:  
3035:         Args:  
3036:             input_data: Input to validate (should be Phase3Input)  
3037:  
3038:         Returns:  
3039:             ContractValidationResult with validation status  
3040:         """  
3041:         errors = []  
3042:         warnings = []  
3043:  
3044:         if not isinstance(input_data, Phase3Input):  
3045:             errors.append(f"Input must be Phase3Input, got {type(input_data).__name__}")  
3046:         return ContractValidationResult(  
3047:             passed=False,  
3048:             contract_type="input",  
3049:             phase_name=self.phase_name,  
3050:             errors=errors  
3051:         )  
3052:  
3053:         # Validate preprocessed document  
3054:         if not isinstance(input_data.preprocessed_document, PreprocessedDocument):  
3055:             errors.append("preprocessed_document must be PreprocessedDocument instance")  
3056:  
3057:         if not input_data.preprocessed_document.chunks:  
3058:             errors.append("preprocessed_document.chunks cannot be empty")  
3059:  
3060:         chunk_count = len(input_data.preprocessed_document.chunks)  
3061:         if chunk_count != 60:  
3062:             warnings.append(f"Expected 60 chunks, found {chunk_count}")  
3063:  
3064:         # Validate questions list  
3065:         if not isinstance(input_data.questions, list):  
3066:             errors.append("questions must be a list")  
3067:  
3068:         if not input_data.questions:  
3069:             errors.append("questions list cannot be empty")  
3070:  
3071:         # Validate question structure  
3072:         for idx, question in enumerate(input_data.questions[:5]): # Sample first 5  
3073:             if not isinstance(question, dict):  
3074:                 errors.append(f"Question {idx} must be a dict")  
3075:                 continue  
3076:  
3077:             if "policy_area_id" not in question:  
3078:                 errors.append(f"Question {idx} missing policy_area_id")  
3079:
```

```
3080:         if "dimension_id" not in question:
3081:             errors.append(f"Question {idx} missing dimension_id")
3082:
3083:     passed = len(errors) == 0
3084:
3085:     return ContractValidationResult(
3086:         passed=passed,
3087:         contract_type="input",
3088:         phase_name=self.phase_name,
3089:         errors=errors,
3090:         warnings=warnings
3091:     )
3092:
3093: def validate_output(self, output_data: Any) -> ContractValidationResult:
3094:     """Validate Phase 3 output contract.
3095:
3096:     Args:
3097:         output_data: Output to validate (should be Phase3Result)
3098:
3099:     Returns:
3100:         ContractValidationResult with validation status
3101:     """
3102:     errors = []
3103:     warnings = []
3104:
3105:     if not isinstance(output_data, Phase3Result):
3106:         errors.append(f"Output must be Phase3Result, got {type(output_data).__name__}")
3107:     return ContractValidationResult(
3108:         passed=False,
3109:         contract_type="output",
3110:         phase_name=self.phase_name,
3111:         errors=errors
3112:     )
3113:
3114:     # Validate routing results structure
3115:     if not isinstance(output_data.routing_results, list):
3116:         errors.append("routing_results must be a list")
3117:
3118:     # Validate counts
3119:     if output_data.total_questions < 0:
3120:         errors.append("total_questions cannot be negative")
3121:
3122:     if output_data.successful_routes < 0:
3123:         errors.append("successful_routes cannot be negative")
3124:
3125:     if output_data.failed_routes < 0:
3126:         errors.append("failed_routes cannot be negative")
3127:
3128:     # Validate routing result objects
3129:     for idx, result in enumerate(output_data.routing_results[:5]): # Sample first 5
3130:         if not isinstance(result, ChunkRoutingResult):
3131:             errors.append(f"routing_results[{idx}] must be ChunkRoutingResult")
3132:             continue
3133:
3134:         if not result.chunk_id:
3135:             errors.append(f"routing_results[{idx}] has empty chunk_id")
```

```
3136:             if not result.policy_area_id:
3137:                 errors.append(f"routing_results[{idx}] has empty policy_area_id")
3138:
3139:             if not result.dimension_id:
3140:                 errors.append(f"routing_results[{idx}] has empty dimension_id")
3141:
3142:
3143:             # Check for failures
3144:             if output_data.failed_routes > 0:
3145:                 warnings.append(f"{output_data.failed_routes} questions failed routing")
3146:
3147:             passed = len(errors) == 0
3148:
3149:         return ContractValidationResult(
3150:             passed=passed,
3151:             contract_type="output",
3152:             phase_name=self.phase_name,
3153:             errors=errors,
3154:             warnings=warnings
3155:         )
3156:
3157:     async def execute(self, input_data: Phase3Input) -> Phase3Result:
3158:         """Execute Phase 3: chunk routing logic.
3159:
3160:             Routes each question to its corresponding chunk by matching
3161:             policy_area_id and dimension_id between the question and the
3162:             chunk matrix.
3163:
3164:             Args:
3165:                 input_data: Phase3Input with preprocessed document and questions
3166:
3167:             Returns:
3168:                 Phase3Result with routing outcomes and observability metrics
3169:
3170:             Raises:
3171:                 ValueError: If routing logic encounters invalid data
3172: """
3173:     logger.info("=" * 70)
3174:     logger.info("PHASE 3: Chunk Routing - Starting Execution")
3175:     logger.info("=" * 70)
3176:
3177:     # Stage 1: Input Extraction (Sequential Dependency Root)
3178:     logger.info("Stage 1: Extracting inputs")
3179:     preprocessed_doc = input_data.preprocessed_document
3180:     questions = input_data.questions
3181:
3182:     logger.info(f"Loaded {len(questions)} questions")
3183:     logger.info(f"Loaded {len(preprocessed_doc.chunks)} chunks")
3184:
3185:     # Stage 2: Validation
3186:     logger.info("Stage 2: Validating chunk matrix")
3187:     chunk_matrix = self._build_chunk_matrix(preprocessed_doc)
3188:     logger.info(f"Built chunk matrix with {len(chunk_matrix)} entries")
3189:
3190:     # Stage 3: Transformation (Routing)
3191:     logger.info("Stage 3: Performing chunk routing")
```

```
3192:     routing_results: list[ChunkRoutingResult] = []
3193:     routing_errors: list[str] = []
3194:     policy_area_dist: dict[str, int] = {}
3195:     dimension_dist: dict[str, int] = {}
3196:
3197:     for question in questions:
3198:         try:
3199:             routing_result = self._route_question_to_chunk(
3200:                 question=question,
3201:                 chunk_matrix=chunk_matrix
3202:             )
3203:             routing_results.append(routing_result)
3204:
3205:             # Update distributions
3206:             pa_id = routing_result.policy_area_id
3207:             dim_id = routing_result.dimension_id
3208:             policy_area_dist[pa_id] = policy_area_dist.get(pa_id, 0) + 1
3209:             dimension_dist[dim_id] = dimension_dist.get(dim_id, 0) + 1
3210:
3211:         except ValueError as e:
3212:             # Stage 4: Error Conditions (Leaf Nodes)
3213:             error_msg = str(e)
3214:             routing_errors.append(error_msg)
3215:             logger.error(f"Routing failed: {error_msg}")
3216:
3217:     # Stage 5: Observability
3218:     successful_routes = len(routing_results)
3219:     failed_routes = len(routing_errors)
3220:     total_questions = len(questions)
3221:
3222:     logger.info("=" * 70)
3223:     logger.info("PHASE 3: Chunk Routing - Execution Complete")
3224:     logger.info(f"Total Questions: {total_questions}")
3225:     logger.info(f"Successful Routes: {successful_routes}")
3226:     logger.info(f"Failed Routes: {failed_routes}")
3227:     logger.info("Policy Area Distribution:")
3228:     for pa_id in sorted(policy_area_dist.keys()):
3229:         logger.info(f" {pa_id}: {policy_area_dist[pa_id]} questions")
3230:     logger.info("Dimension Distribution:")
3231:     for dim_id in sorted(dimension_dist.keys()):
3232:         logger.info(f" {dim_id}: {dimension_dist[dim_id]} questions")
3233:     logger.info("=" * 70)
3234:
3235:     return Phase3Result(
3236:         routing_results=routing_results,
3237:         total_questions=total_questions,
3238:         successful_routes=successful_routes,
3239:         failed_routes=failed_routes,
3240:         policy_area_distribution=policy_area_dist,
3241:         dimension_distribution=dimension_dist,
3242:         routing_errors=routing_errors
3243:     )
3244:
3245:     def _build_chunk_matrix(
3246:         self,
3247:         preprocessed_doc: PreprocessedDocument
```

```
3248:     ) -> dict[tuple[str, str], ChunkData]:
3249:         """Build chunk matrix keyed by (policy_area_id, dimension_id).
3250:
3251:         Args:
3252:             preprocessed_doc: PreprocessedDocument with chunks
3253:
3254:         Returns:
3255:             Dictionary mapping (PA, DIM) tuples to ChunkData
3256:
3257:         Raises:
3258:             ValueError: If chunk matrix is invalid
3259:         """
3260:         chunk_matrix: dict[tuple[str, str], ChunkData] = {}
3261:
3262:         for chunk in preprocessed_doc.chunks:
3263:             if chunk.policy_area_id is None:
3264:                 raise ValueError(
3265:                     f"Chunk {chunk.id} has null policy_area_id"
3266:                 )
3267:
3268:             if chunk.dimension_id is None:
3269:                 raise ValueError(
3270:                     f"Chunk {chunk.id} has null dimension_id"
3271:                 )
3272:
3273:             key = (chunk.policy_area_id, chunk.dimension_id)
3274:
3275:             if key in chunk_matrix:
3276:                 raise ValueError(
3277:                     f"Duplicate chunk for {chunk.policy_area_id}-{chunk.dimension_id}"
3278:                 )
3279:
3280:             chunk_matrix[key] = chunk
3281:
3282: # Validate we have 60 chunks (10 PA \227 6 DIM)
3283: if len(chunk_matrix) != 60:
3284:     raise ValueError(
3285:         f"Expected 60 chunks in matrix, found {len(chunk_matrix)}"
3286:     )
3287:
3288:     return chunk_matrix
3289:
3290: def _route_question_to_chunk(
3291:     self,
3292:     question: dict[str, Any],
3293:     chunk_matrix: dict[tuple[str, str], ChunkData]
3294: ) -> ChunkRoutingResult:
3295:     """Route a single question to its corresponding chunk.
3296:
3297:     This method implements the core routing logic with strict
3298:     policy_area_id and dimension_id equality enforcement.
3299:
3300:     Args:
3301:         question: Question dictionary with policy_area_id and dimension_id
3302:         chunk_matrix: Matrix of chunks keyed by (PA, DIM)
3303:
```

```
3304:     Returns:
3305:         ChunkRoutingResult with all seven canonical fields populated
3306:
3307:     Raises:
3308:         ValueError: If question is missing required fields or no matching chunk found
3309:         """
3310:     # Extract question identifiers
3311:     question_id = question.get("question_id", "UNKNOWN")
3312:     policy_area_id = question.get("policy_area_id")
3313:     dimension_id = question.get("dimension_id")
3314:
3315:     # Validate question has required fields
3316:     if policy_area_id is None:
3317:         raise ValueError(
3318:             f"Question {question_id} missing required field 'policy_area_id'"
3319:         )
3320:
3321:     if dimension_id is None:
3322:         raise ValueError(
3323:             f"Question {question_id} missing required field 'dimension_id'"
3324:         )
3325:
3326:     # Convert dimension format if needed (D1 \206\222 DIM01)
3327:     if isinstance(dimension_id, str) and dimension_id.startswith("D") and not dimension_id.startswith("DIM"):
3328:         # Extract number from D1, D2, etc.
3329:         try:
3330:             dim_num = int(dimension_id[1:])
3331:             dimension_id = f"DIM{dim_num:02d}"
3332:         except (ValueError, IndexError):
3333:             raise ValueError(
3334:                 f"Question {question_id} has invalid dimension_id format: {dimension_id}"
3335:             )
3336:
3337:     # Construct lookup key
3338:     lookup_key = (policy_area_id, dimension_id)
3339:
3340:     # Perform chunk lookup
3341:     if lookup_key not in chunk_matrix:
3342:         raise ValueError(
3343:             f"Question {question_id} routing failed: "
3344:             f"No matching chunk found for policy_area_id={policy_area_id}, "
3345:             f"dimension_id={dimension_id}. "
3346:             f"Required chunk {policy_area_id}-{dimension_id} is missing from the chunk matrix."
3347:         )
3348:
3349:     target_chunk = chunk_matrix[lookup_key]
3350:
3351:     # Verify strict equality between question and chunk identifiers
3352:     if target_chunk.policy_area_id != policy_area_id:
3353:         raise ValueError(
3354:             f"Question {question_id} routing verification failed: "
3355:             f"Chunk policy_area_id mismatch. "
3356:             f"Question expects {policy_area_id}, chunk has {target_chunk.policy_area_id}"
3357:         )
3358:
3359:     if target_chunk.dimension_id != dimension_id:
```

```
3360:             raise ValueError(
3361:                 f"Question {question_id} routing verification failed: "
3362:                 f"Chunk dimension_id mismatch. "
3363:                 f"Question expects {dimension_id}, chunk has {target_chunk.dimension_id}"
3364:             )
3365:
3366:     # Extract chunk_id (guaranteed to be present after validation)
3367:     chunk_id = target_chunk.chunk_id or f"{policy_area_id}-{dimension_id}"
3368:
3369:     # Extract text content
3370:     text_content = target_chunk.text
3371:
3372:     # Extract expected_elements (ensure it's never None)
3373:     expected_elements = target_chunk.expected_elements or []
3374:
3375:     # Extract document_position (can be None)
3376:     document_position = target_chunk.document_position
3377:
3378:     # Construct and return ChunkRoutingResult with all 7 canonical fields
3379:     return ChunkRoutingResult(
3380:         target_chunk=target_chunk,
3381:         chunk_id=chunk_id,
3382:         policy_area_id=policy_area_id,
3383:         dimension_id=dimension_id,
3384:         text_content=text_content,
3385:         expected_elements=expected_elements,
3386:         document_position=document_position
3387:     )
3388:
3389:
3390: __all__ = [
3391:     "ChunkRoutingResult",
3392:     "Phase3Input",
3393:     "Phase3Result",
3394:     "Phase3ChunkRoutingContract",
3395: ]
3396:
3397:
3398:
3399: =====
3400: FILE: src/farfan_pipeline/core/phases/phase5_signal_resolution.py
3401: =====
3402:
3403: """Phase 5: Signal Resolution
3404:
3405: This module implements Phase 5 signal resolution that accepts question dictionaries
3406: and resolves required signals by looking them up in a signal registry.
3407:
3408: Key Features:
3409: - Iterates over question's signal_requirements list
3410: - Looks up each signal type in registry (maps types to handler instances)
3411: - Raises ValueError if any signal type is missing (hard-stop)
3412: - Validates signal_requirements entries contain signal_type string field via dict get
3413: - Returns immutable tuple of resolved signal objects
3414: - Handles missing/empty signal_requirements by returning empty tuple
3415: - Structured debug logging with question ID, counts, and correlation ID
```

```
3416:  
3417: Design:  
3418: - Signal registry passed as constructor parameter or stored as instance state  
3419: - Signal handlers cannot be mutated after resolution completes (immutable tuple)  
3420: - No fallbacks or degraded modes - missing signals cause immediate failure  
3421: - Registry is a dict-like mapping or object where signal types are lookup keys  
3422:  
3423: Example Usage:  
3424:     >>> # Create a signal registry with handler instances  
3425:     >>> class SignalRegistry:  
3426:         ...     def __init__(self):  
3427:             ...         self.budget_signal = BudgetSignalHandler()  
3428:             ...         self.actor_signal = ActorSignalHandler()  
3429:             ...         self.timeline_signal = TimelineSignalHandler()  
3430:     >>>  
3431:     >>> registry = SignalRegistry()  
3432:     >>> resolver = SignalResolver(registry)  
3433:     >>>  
3434:     >>> # Question with signal requirements  
3435:     >>> question = {  
3436:         ...     "question_id": "Q001",  
3437:         ...     "signal_requirements": [  
3438:             {"signal_type": "budget_signal"},  
3439:             {"signal_type": "actor_signal"}  
3440:         ...     ]  
3441:     ... }  
3442:     >>>  
3443:     >>> # Resolve signals - returns immutable tuple  
3444:     >>> signals = resolver.resolve_signals_for_question(  
3445:         ...     question=question,  
3446:         ...     correlation_id="corr-123"  
3447:         ... )  
3448:     >>>  
3449:     >>> # signals is now an immutable tuple of handlers  
3450:     >>> assert isinstance(signals, tuple)  
3451:     >>> assert len(signals) == 2  
3452:     >>>  
3453:     >>> # Missing signal raises ValueError  
3454:     >>> question_missing = {  
3455:         ...     "question_id": "Q002",  
3456:         ...     "signal_requirements": [{"signal_type": "missing_signal"}]  
3457:         ... }  
3458:     >>> try:  
3459:         ...     resolver.resolve_signals_for_question(question_missing, "corr-124")  
3460:     ... except ValueError as e:  
3461:         ...     print(f"Hard-stop error: {e}")  
3462:     ...  
3463: Hard-stop error: Signal type 'missing_signal' missing from registry...  
3464: >>>  
3465:     >>> # Question without requirements returns empty tuple  
3466:     >>> question_no_signals = {"question_id": "Q003"}  
3467:     >>> result = resolver.resolve_signals_for_question(question_no_signals, "corr-125")  
3468:     >>> assert result == ()  
3469:  
3470: Alternative Usage with Dict Registry:  
3471:     >>> # Registry as a dict
```

```
3472:     >>> registry_dict = {
3473:         ...         "budget_signal": BudgetSignalHandler(),
3474:         ...         "actor_signal": ActorSignalHandler()
3475:         ...
3476:     >>> resolver = SignalResolver(registry_dict)
3477:     >>>
3478:     >>> # Signal requirements as simple list of strings
3479:     >>> question = {
3480:         ...         "question_id": "Q004",
3481:         ...         "signal_requirements": ["budget_signal", "actor_signal"]
3482:         ...
3483:     >>> signals = resolver.resolve_signals_for_question(question, "corr-126")
3484: """
3485:
3486: from __future__ import annotations
3487:
3488: import contextlib
3489: from typing import Any
3490:
3491: try:
3492:     import structlog
3493:
3494:     logger = structlog.get_logger(__name__)
3495: except ImportError:
3496:     import logging
3497:
3498:     logger = logging.getLogger(__name__)
3499:
3500:
3501: class SignalResolver:
3502:     """Phase 5 signal resolver with registry-based signal lookup.
3503:
3504:     This class encapsulates the signal resolution logic for Phase 5,
3505:     accepting a signal registry at construction time and using it to
3506:     resolve signals for questions during execution.
3507:
3508:     The signal registry can be:
3509:     - A dict mapping signal types to handler instances
3510:     - An object with signal types as attributes
3511:     - Any object that supports item access or attribute access
3512:
3513:     Attributes:
3514:         signal_registry: Registry that maps signal types to handler instances
3515:     """
3516:
3517:     def __init__(self, signal_registry: Any) -> None:
3518:         """Initialize signal resolver with registry.
3519:
3520:         Args:
3521:             signal_registry: Registry mapping signal types to handler instances.
3522:                         Can be a dict, object with attributes, or similar.
3523:         """
3524:         self.signal_registry = signal_registry
3525:
3526:     def resolve_signals_for_question(
3527:         self,
```

```
3528:         question: dict[str, Any],  
3529:         correlation_id: str,  
3530:     ) -> tuple[Any, ...]:  
3531:         """Resolve signals for a question by iterating over signal_requirements.  
3532:  
3533:         This method:  
3534:             1. Extracts signal_requirements from question dictionary via get()  
3535:             2. Returns empty tuple if signal_requirements is missing or empty  
3536:             3. Validates each entry contains signal_type string field via dict get with None-checking  
3537:             4. Iterates over signal_requirements list  
3538:             5. Looks up each signal type in registry as a key  
3539:             6. Raises ValueError if any signal type is missing (hard-stop, no fallback)  
3540:             7. Collects resolved signal handlers into a list  
3541:             8. Converts list to immutable tuple before returning  
3542:             9. Logs debug info with question ID, required count, resolved count, correlation ID  
3543:  
3544:     Args:  
3545:         question: Question dict with optional signal_requirements field  
3546:         correlation_id: Correlation ID for tracing and logging  
3547:  
3548:     Returns:  
3549:         Immutable tuple of resolved signal handler objects. Empty tuple if  
3550:         signal_requirements is missing or empty.  
3551:  
3552:     Raises:  
3553:         ValueError: When any signal type is missing from registry, since unresolved  
3554:                         signals would cause executor failures. Also raised when  
3555:                         signal_requirements entry does not contain signal_type string field.  
3556:     """  
3557:     question_id = question.get("question_id", "UNKNOWN")  
3558:  
3559:     signal_requirements = question.get("signal_requirements")  
3560:  
3561:     if signal_requirements is None or not signal_requirements:  
3562:         return ()  
3563:  
3564:     if not isinstance(signal_requirements, list):  
3565:         if isinstance(signal_requirements, set | tuple):  
3566:             signal_requirements = list(signal_requirements)  
3567:         elif isinstance(signal_requirements, dict):  
3568:             signal_requirements = list(signal_requirements.keys())  
3569:         else:  
3570:             return ()  
3571:  
3572:     if not signal_requirements:  
3573:         return ()  
3574:  
3575:     resolved_signals_list: list[Any] = []  
3576:     required_types_list: list[str] = []  
3577:  
3578:     for idx, requirement in enumerate(signal_requirements):  
3579:         signal_type: str | None = None  
3580:  
3581:         if isinstance(requirement, dict):  
3582:             signal_type = requirement.get("signal_type")  
3583:             if signal_type is None:
```

```
3584:             raise ValueError(
3585:                 f"Signal requirement at index {idx} missing signal_type field "
3586:                 f"for question {question_id}"
3587:             )
3588:         if not isinstance(signal_type, str):
3589:             raise ValueError(
3590:                 f"Signal requirement at index {idx} has non-string signal_type "
3591:                 f"for question {question_id}: {type(signal_type).__name__}"
3592:             )
3593:     elif isinstance(requirement, str):
3594:         signal_type = requirement
3595:     else:
3596:         raise ValueError(
3597:             f"Signal requirement at index {idx} has invalid type "
3598:             f"for question {question_id}: {type(requirement).__name__}"
3599:         )
3600:
3601:     required_types_list.append(signal_type)
3602:
3603:     signal_handler = None
3604:
3605:     if isinstance(self.signal_registry, dict):
3606:         signal_handler = self.signal_registry.get(signal_type)
3607:     elif hasattr(self.signal_registry, signal_type):
3608:         signal_handler = getattr(self.signal_registry, signal_type)
3609:     elif hasattr(self.signal_registry, "__getitem__"):
3610:         with contextlib.suppress(KeyError, TypeError):
3611:             signal_handler = self.signal_registry[signal_type]
3612:
3613:     if signal_handler is None:
3614:         raise ValueError(
3615:             f"Signal type '{signal_type}' missing from registry "
3616:             f"for question {question_id}. "
3617:             f"Unresolved signals would cause executor failures."
3618:         )
3619:
3620:     resolved_signals_list.append(signal_handler)
3621:
3622:     resolved_count = len(resolved_signals_list)
3623:     required_count = len(required_types_list)
3624:
3625:     logger.debug(
3626:         "signals_resolved_for_question",
3627:         question_id=question_id,
3628:         required_signals=required_count,
3629:         resolved_signals=resolved_count,
3630:         correlation_id=correlation_id,
3631:     )
3632:
3633:     return tuple(resolved_signals_list)
3634:
3635:
3636: def resolve_signals_for_question(
3637:     question: dict[str, Any],
3638:     signal_registry: Any,
3639:     correlation_id: str,
```

```
3640: ) -> tuple[Any, ...]:  
3641:     """Standalone function for resolving signals for a question.  
3642:  
3643:     This is a convenience function that creates a SignalResolver and calls  
3644:     its resolve_signals_for_question method. Use this when you don't need  
3645:     to maintain resolver state across multiple calls.  
3646:  
3647:     Args:  
3648:         question: Question dict with optional signal_requirements field  
3649:         signal_registry: Registry that maps signal types to handler instances  
3650:         correlation_id: Correlation ID for tracing and logging  
3651:  
3652:     Returns:  
3653:         Immutable tuple of resolved signal handler objects  
3654:  
3655:     Raises:  
3656:         ValueError: When any signal type is missing from registry or  
3657:                     when signal_requirements entry is invalid  
3658:     """  
3659:     resolver = SignalResolver(signal_registry)  
3660:     return resolver.resolve_signals_for_question(question, correlation_id)  
3661:  
3662:  
3663: __all__ = [  
3664:     "SignalResolver",  
3665:     "resolve_signals_for_question",  
3666: ]  
3667:  
3668:  
3669:  
3670: =====  
3671: FILE: src/farfan_pipeline/core/phases/phase6_schema_validation.py  
3672: =====  
3673:  
3674: """  
3675: Phase 6: Schema Validation - Comprehensive Input Validation for Task Construction  
3676: =====  
3677:  
3678: This module implements Phase 6 of the canonical pipeline:  
3679:     Raw Question/Chunk Data \206\222 Validated Schemas \206\222 Phase 7 Task Construction  
3680:  
3681: Responsibilities:  
3682: -----  
3683: 1. Validate question schema structure and field types  
3684: 2. Validate chunk schema structure and field types  
3685: 3. Enforce non-null constraints on mandatory fields  
3686: 4. Validate data type correctness before downstream use  
3687: 5. Check for missing required fields with explicit diagnostics  
3688: 6. Ensure policy area and dimension ID consistency  
3689: 7. Validate question_global range and format  
3690:  
3691: Input Contract:  
3692: -----  
3693: Phase5AggregationOutput (conceptual - from aggregation phase):  
3694:     - raw_questions: List[Dict[str, Any]]  
3695:     - raw_chunks: List[Dict[str, Any]]
```

```
3696:     - aggregation_metadata: Dict[str, Any]
3697:
3698: Output Contract (to Phase 7):
3699: -----
3700: Phase6SchemaValidationOutput:
3701:     - validated_questions: List[ValidatedQuestionSchema]
3702:     - validated_chunks: List[ValidatedChunkSchema]
3703:     - schema_validation_passed: bool
3704:     - validation_errors: List[str]
3705:     - validation_warnings: List[str]
3706:     - question_count: int
3707:     - chunk_count: int
3708:     - validation_timestamp: str
3709:
3710: ValidatedQuestionSchema:
3711:     - question_id: str (non-null, non-empty)
3712:     - question_global: int (0-999)
3713:     - dimension_id: str (non-null, non-empty)
3714:     - policy_area_id: str (non-null, non-empty)
3715:     - base_slot: str (non-null)
3716:     - cluster_id: str (non-null)
3717:     - patterns: List[Dict[str, Any]] (validated list type)
3718:     - signals: Dict[str, Any] (validated dict type)
3719:     - expected_elements: List[Dict[str, Any]] (validated list type)
3720:     - metadata: Dict[str, Any] (validated dict type)
3721:
3722: ValidatedChunkSchema:
3723:     - chunk_id: str (non-null, non-empty)
3724:     - policy_area_id: str (non-null, non-empty)
3725:     - dimension_id: str (non-null, non-empty)
3726:     - document_position: int (non-negative)
3727:     - content: str (non-empty)
3728:     - metadata: Dict[str, Any] (validated dict type)
3729:
3730: Error Propagation Semantics:
3731: -----
3732: 1. Schema validation failures prevent Phase 7 execution
3733: 2. Type mismatches are caught BEFORE Phase 7 receives data
3734: 3. Missing required fields are explicitly reported with field name
3735: 4. All errors are accumulated and returned in validation_errors list
3736:
3737: Integration Points:
3738: -----
3739: Phase 5 \206\222 Phase 6:
3740:     - Contract: Raw question and chunk dictionaries
3741:     - Precondition: None (Phase 6 validates everything)
3742:     - Error: Schema violations \206\222 schema_validation_passed=False
3743:
3744: Phase 6 \206\222 Phase 7:
3745:     - Contract: Phase6SchemaValidationOutput
3746:     - Postcondition: schema_validation_passed must be True for Phase 7 to proceed
3747:     - Error: If False, Phase 7 returns empty task list with propagated errors
3748:
3749: Author: F.A.R.F.A.N Architecture Team
3750: Date: 2025-01-19
3751: Version: 1.0.0
```

```
3752: """
3753:
3754: from __future__ import annotations
3755:
3756: import logging
3757: from dataclasses import dataclass
3758: from datetime import datetime, timezone
3759: from typing import Any
3760:
3761: logger = logging.getLogger(__name__)
3762:
3763: PHASE6_VERSION = "1.0.0"
3764: MAX_QUESTION_GLOBAL = 999
3765:
3766:
3767: @dataclass
3768: class ValidatedQuestionSchema:
3769:     """Validated question schema output."""
3770:
3771:     question_id: str
3772:     question_global: int
3773:     dimension_id: str
3774:     policy_area_id: str
3775:     base_slot: str
3776:     cluster_id: str
3777:     patterns: list[dict[str, Any]]
3778:     signals: dict[str, Any]
3779:     expected_elements: list[dict[str, Any]]
3780:     metadata: dict[str, Any]
3781:
3782:
3783: @dataclass
3784: class ValidatedChunkSchema:
3785:     """Validated chunk schema output."""
3786:
3787:     chunk_id: str
3788:     policy_area_id: str
3789:     dimension_id: str
3790:     document_position: int
3791:     content: str
3792:     metadata: dict[str, Any]
3793:
3794:
3795: @dataclass
3796: class Phase6SchemaValidationOutput:
3797:     """Output contract for Phase 6."""
3798:
3799:     validated_questions: list[ValidatedQuestionSchema]
3800:     validated_chunks: list[ValidatedChunkSchema]
3801:     schema_validation_passed: bool
3802:     validation_errors: list[str]
3803:     validation_warnings: list[str]
3804:     question_count: int
3805:     chunk_count: int
3806:     validation_timestamp: str
3807:     phase6_version: str = PHASE6_VERSION
```

```
3808:  
3809:  
3810: def _validate_question_schema( # noqa: PLR0911, PLR0912, PLR0915  
3811:     question: dict[str, Any],  
3812:     index: int,  
3813:     errors: list[str],  
3814:     warnings: list[str],  
3815: ) -> ValidatedQuestionSchema | None:  
3816:     """  
3817:         Validate a single question schema with explicit error handling.  
3818:  
3819:         Validates fields in order:  
3820:             1. question_id: str (non-null, non-empty)  
3821:             2. question_global: int (0-999)  
3822:             3. dimension_id: str (non-null, non-empty)  
3823:             4. policy_area_id: str (non-null, non-empty)  
3824:             5. base_slot: str (non-null)  
3825:             6. cluster_id: str (non-null)  
3826:             7. patterns: List[Dict] (type validation)  
3827:             8. signals: Dict (type validation)  
3828:             9. expected_elements: List[Dict] (type validation)  
3829:             10. metadata: Dict (type validation)  
3830:  
3831:         Args:  
3832:             question: Raw question dictionary  
3833:             index: Index in question list for error reporting  
3834:             errors: List to accumulate errors  
3835:             warnings: List to accumulate warnings  
3836:  
3837:         Returns:  
3838:             ValidatedQuestionSchema or None if validation failed  
3839:     """  
3840:     context = f"question[{index}]"  
3841:  
3842:     try:  
3843:         question_id = question["question_id"]  
3844:     except KeyError:  
3845:         errors.append(f"Field 'question_id' not found in {context}: KeyError")  
3846:         return None  
3847:  
3848:     if not isinstance(question_id, str):  
3849:         errors.append(  
3850:             f"Field 'question_id' has invalid type in {context}: "  
3851:             f"expected str, got {type(question_id).__name__}")  
3852:     )  
3853:     return None  
3854:  
3855:     if not question_id:  
3856:         errors.append(f"Field 'question_id' is empty string in {context}")  
3857:         return None  
3858:  
3859:     try:  
3860:         question_global = question["question_global"]  
3861:     except KeyError:  
3862:         errors.append(f"Field 'question_global' not found in {context}: KeyError")  
3863:         return None
```

```
3864:  
3865:     if not isinstance(question_global, int):  
3866:         errors.append(  
3867:             f"Field 'question_global' has invalid type in {context}: "  
3868:             f"expected int, got {type(question_global).__name__}"  
3869:         )  
3870:     return None  
3871:  
3872:     if not (0 <= question_global <= MAX_QUESTION_GLOBAL):  
3873:         errors.append(  
3874:             f"Field 'question_global' has invalid value in {context}: "  
3875:             f"{question_global} not in range [0, {MAX_QUESTION_GLOBAL}]"  
3876:         )  
3877:     return None  
3878:  
3879:     try:  
3880:         dimension_id = question["dimension_id"]  
3881:     except KeyError:  
3882:         errors.append(f"Field 'dimension_id' not found in {context}: KeyError")  
3883:     return None  
3884:  
3885:     if not isinstance(dimension_id, str):  
3886:         errors.append(  
3887:             f"Field 'dimension_id' has invalid type in {context}: "  
3888:             f"expected str, got {type(dimension_id).__name__}"  
3889:         )  
3890:     return None  
3891:  
3892:     if not dimension_id:  
3893:         errors.append(f"Field 'dimension_id' is empty string in {context}")  
3894:     return None  
3895:  
3896:     try:  
3897:         policy_area_id = question["policy_area_id"]  
3898:     except KeyError:  
3899:         errors.append(f"Field 'policy_area_id' not found in {context}: KeyError")  
3900:     return None  
3901:  
3902:     if not isinstance(policy_area_id, str):  
3903:         errors.append(  
3904:             f"Field 'policy_area_id' has invalid type in {context}: "  
3905:             f"expected str, got {type(policy_area_id).__name__}"  
3906:         )  
3907:     return None  
3908:  
3909:     if not policy_area_id:  
3910:         errors.append(f"Field 'policy_area_id' is empty string in {context}")  
3911:     return None  
3912:  
3913:     try:  
3914:         base_slot = question["base_slot"]  
3915:     except KeyError:  
3916:         base_slot = ""  
3917:         warnings.append(f"Field 'base_slot' not found in {context}; using empty string")  
3918:  
3919:     if not isinstance(base_slot, str):
```

```
3920:         errors.append(
3921:             f"Field 'base_slot' has invalid type in {context}: "
3922:             f"expected str, got {type(base_slot).__name__}"
3923:         )
3924:     return None
3925:
3926:     try:
3927:         cluster_id = question["cluster_id"]
3928:     except KeyError:
3929:         cluster_id = ""
3930:     warnings.append(
3931:         f"Field 'cluster_id' not found in {context}; using empty string"
3932:     )
3933:
3934:     if not isinstance(cluster_id, str):
3935:         errors.append(
3936:             f"Field 'cluster_id' has invalid type in {context}: "
3937:             f"expected str, got {type(cluster_id).__name__}"
3938:         )
3939:     return None
3940:
3941:     try:
3942:         patterns = question["patterns"]
3943:     except KeyError:
3944:         errors.append(f"Field 'patterns' not found in {context}: KeyError")
3945:     return None
3946:
3947:     if not isinstance(patterns, list):
3948:         errors.append(
3949:             f"Field 'patterns' has invalid type in {context}: "
3950:             f"expected list, got {type(patterns).__name__}"
3951:         )
3952:     return None
3953:
3954:     try:
3955:         signals = question["signals"]
3956:     except KeyError:
3957:         errors.append(f"Field 'signals' not found in {context}: KeyError")
3958:     return None
3959:
3960:     if not isinstance(signals, dict):
3961:         errors.append(
3962:             f"Field 'signals' has invalid type in {context}: "
3963:             f"expected dict, got {type(signals).__name__}"
3964:         )
3965:     return None
3966:
3967:     try:
3968:         expected_elements = question["expected_elements"]
3969:     except KeyError:
3970:         expected_elements = []
3971:         warnings.append(
3972:             f"Field 'expected_elements' not found in {context}; using empty list"
3973:         )
3974:
3975:     if not isinstance(expected_elements, list):
```

```
3976:         errors.append(
3977:             f"Field 'expected_elements' has invalid type in {context}: "
3978:             f"expected list, got {type(expected_elements).__name__}"
3979:         )
3980:     return None
3981:
3982:     try:
3983:         metadata = question["metadata"]
3984:     except KeyError:
3985:         metadata = {}
3986:     warnings.append(f"Field 'metadata' not found in {context}; using empty dict")
3987:
3988:     if not isinstance(metadata, dict):
3989:         errors.append(
3990:             f"Field 'metadata' has invalid type in {context}: "
3991:             f"expected dict, got {type(metadata).__name__}"
3992:         )
3993:     return None
3994:
3995:     return ValidatedQuestionSchema(
3996:         question_id=question_id,
3997:         question_global=question_global,
3998:         dimension_id=dimension_id,
3999:         policy_area_id=policy_area_id,
4000:         base_slot=base_slot,
4001:         cluster_id=cluster_id,
4002:         patterns=patterns,
4003:         signals=signals,
4004:         expected_elements=expected_elements,
4005:         metadata=metadata,
4006:     )
4007:
4008:
4009: def _validate_chunk_schema( # noqa: PLR0911, PLR0912
4010:     chunk: dict[str, Any],
4011:     index: int,
4012:     errors: list[str],
4013:     warnings: list[str],
4014: ) -> ValidatedChunkSchema | None:
4015: """
4016:     Validate a single chunk schema with explicit error handling.
4017:
4018:     Validates fields in order:
4019:     1. chunk_id: str (non-null, non-empty)
4020:     2. policy_area_id: str (non-null, non-empty)
4021:     3. dimension_id: str (non-null, non-empty)
4022:     4. document_position: int (non-negative)
4023:     5. content: str (non-empty)
4024:     6. metadata: Dict (type validation)
4025:
4026:     Args:
4027:         chunk: Raw chunk dictionary
4028:         index: Index in chunk list for error reporting
4029:         errors: List to accumulate errors
4030:         warnings: List to accumulate warnings
4031:
```

```
4032:     Returns:  
4033:         ValidatedChunkSchema or None if validation failed  
4034:     """  
4035:     context = f"chunk[{index}]"  
4036:  
4037:     try:  
4038:         chunk_id = chunk["chunk_id"]  
4039:     except KeyError:  
4040:         errors.append(f"Field 'chunk_id' not found in {context}: KeyError")  
4041:         return None  
4042:  
4043:     if not isinstance(chunk_id, str):  
4044:         errors.append(  
4045:             f"Field 'chunk_id' has invalid type in {context}: "  
4046:             f"expected str, got {type(chunk_id).__name__}"  
4047:         )  
4048:         return None  
4049:  
4050:     if not chunk_id:  
4051:         errors.append(f"Field 'chunk_id' is empty string in {context}")  
4052:         return None  
4053:  
4054:     try:  
4055:         policy_area_id = chunk["policy_area_id"]  
4056:     except KeyError:  
4057:         errors.append(f"Field 'policy_area_id' not found in {context}: KeyError")  
4058:         return None  
4059:  
4060:     if not isinstance(policy_area_id, str):  
4061:         errors.append(  
4062:             f"Field 'policy_area_id' has invalid type in {context}: "  
4063:             f"expected str, got {type(policy_area_id).__name__}"  
4064:         )  
4065:         return None  
4066:  
4067:     if not policy_area_id:  
4068:         errors.append(f"Field 'policy_area_id' is empty string in {context}")  
4069:         return None  
4070:  
4071:     try:  
4072:         dimension_id = chunk["dimension_id"]  
4073:     except KeyError:  
4074:         errors.append(f"Field 'dimension_id' not found in {context}: KeyError")  
4075:         return None  
4076:  
4077:     if not isinstance(dimension_id, str):  
4078:         errors.append(  
4079:             f"Field 'dimension_id' has invalid type in {context}: "  
4080:             f"expected str, got {type(dimension_id).__name__}"  
4081:         )  
4082:         return None  
4083:  
4084:     if not dimension_id:  
4085:         errors.append(f"Field 'dimension_id' is empty string in {context}")  
4086:         return None  
4087:
```

```
4088:     try:
4089:         document_position = chunk["document_position"]
4090:     except KeyError:
4091:         document_position = 0
4092:         warnings.append(f"Field 'document_position' not found in {context}; using 0")
4093:
4094:     if not isinstance(document_position, int):
4095:         errors.append(
4096:             f"Field 'document_position' has invalid type in {context}: "
4097:             f"expected int, got {type(document_position).__name__}"
4098:         )
4099:         return None
4100:
4101:     if document_position < 0:
4102:         errors.append(
4103:             f"Field 'document_position' has invalid value in {context}: "
4104:             f"{document_position} must be non-negative"
4105:         )
4106:         return None
4107:
4108:     try:
4109:         content = chunk["content"]
4110:     except KeyError:
4111:         errors.append(f"Field 'content' not found in {context}: KeyError")
4112:         return None
4113:
4114:     if not isinstance(content, str):
4115:         errors.append(
4116:             f"Field 'content' has invalid type in {context}: "
4117:             f"expected str, got {type(content).__name__}"
4118:         )
4119:         return None
4120:
4121:     if not content:
4122:         errors.append(f"Field 'content' is empty string in {context}")
4123:         return None
4124:
4125:     try:
4126:         metadata = chunk["metadata"]
4127:     except KeyError:
4128:         metadata = {}
4129:         warnings.append(f"Field 'metadata' not found in {context}; using empty dict")
4130:
4131:     if not isinstance(metadata, dict):
4132:         errors.append(
4133:             f"Field 'metadata' has invalid type in {context}: "
4134:             f"expected dict, got {type(metadata).__name__}"
4135:         )
4136:         return None
4137:
4138:     return ValidatedChunkSchema(
4139:         chunk_id=chunk_id,
4140:         policy_area_id=policy_area_id,
4141:         dimension_id=dimension_id,
4142:         document_position=document_position,
4143:         content=content,
```

```
4144:         metadata=metadata,
4145:     )
4146:
4147:
4148: def phase6_schema_validation(
4149:     raw_questions: list[dict[str, Any]],
4150:     raw_chunks: list[dict[str, Any]],
4151: ) -> Phase6SchemaValidationOutput:
4152:
4153:     """Execute Phase 6: Schema Validation.
4154:
4155:     Validates all questions and chunks with explicit error handling for:
4156:     - KeyError: Missing required fields
4157:     - TypeError: Incorrect field types
4158:     - ValueError: Invalid field values
4159:
4160:     All validation occurs BEFORE returning to ensure Phase 7 receives
4161:     clean, type-validated data.
4162:
4163:     Args:
4164:         raw_questions: List of raw question dictionaries
4165:         raw_chunks: List of raw chunk dictionaries
4166:
4167:     Returns:
4168:         Phase6SchemaValidationOutput with validated schemas or error details
4169:     """
4170:     logger.info("Phase 6: Schema Validation started")
4171:
4172:     validation_errors: list[str] = []
4173:     validation_warnings: list[str] = []
4174:     validated_questions: list[ValidatedQuestionSchema] = []
4175:     validated_chunks: list[ValidatedChunkSchema] = []
4176:
4177:     if not isinstance(raw_questions, list):
4178:         validation_errors.append(
4179:             f"raw_questions has invalid type: expected list, got {type(raw_questions).__name__}"
4180:         )
4181:         raw_questions = []
4182:
4183:     if not isinstance(raw_chunks, list):
4184:         validation_errors.append(
4185:             f"raw_chunks has invalid type: expected list, got {type(raw_chunks).__name__}"
4186:         )
4187:         raw_chunks = []
4188:
4189:     logger.info(f"Validating {len(raw_questions)} questions")
4190:
4191:     for index, question in enumerate(raw_questions):
4192:         if not isinstance(question, dict):
4193:             validation_errors.append(
4194:                 f"question[{index}] has invalid type: expected dict, got {type(question).__name__}"
4195:             )
4196:             continue
4197:
4198:         validated_question = _validate_question_schema(
4199:             question, index, validation_errors, validation_warnings
```

```
4200:         )
4201:
4202:     if validated_question:
4203:         validated_questions.append(validated_question)
4204:
4205:     logger.info(
4206:         f"Validated {len(validated_questions)}/{len(raw_questions)} questions "
4207:         f"(errors: {len([e for e in validation_errors if 'question' in e])})"
4208:     )
4209:
4210:     logger.info(f"Validating {len(raw_chunks)} chunks")
4211:
4212:     for index, chunk in enumerate(raw_chunks):
4213:         if not isinstance(chunk, dict):
4214:             validation_errors.append(
4215:                 f"chunk[{index}] has invalid type: expected dict, got {type(chunk).__name__}"
4216:             )
4217:             continue
4218:
4219:         validated_chunk = _validate_chunk_schema(
4220:             chunk, index, validation_errors, validation_warnings
4221:         )
4222:
4223:         if validated_chunk:
4224:             validated_chunks.append(validated_chunk)
4225:
4226:     logger.info(
4227:         f"Validated {len(validated_chunks)}/{len(raw_chunks)} chunks "
4228:         f"(errors: {len([e for e in validation_errors if 'chunk' in e])})"
4229:     )
4230:
4231:     schema_validation_passed = len(validation_errors) == 0
4232:
4233:     validation_timestamp = datetime.now(timezone.utc).isoformat()
4234:
4235:     output = Phase6SchemaValidationOutput(
4236:         validated_questions=validated_questions,
4237:         validated_chunks=validated_chunks,
4238:         schema_validation_passed=schema_validation_passed,
4239:         validation_errors=validation_errors,
4240:         validation_warnings=validation_warnings,
4241:         question_count=len(validated_questions),
4242:         chunk_count=len(validated_chunks),
4243:         validation_timestamp=validation_timestamp,
4244:     )
4245:
4246:     logger.info(
4247:         f"Phase 6: Schema Validation completed "
4248:         f"(passed={schema_validation_passed}, "
4249:         f"questions={len(validated_questions)}, chunks={len(validated_chunks)}, "
4250:         f"errors={len(validation_errors)}, warnings={len(validation_warnings)})"
4251:     )
4252:
4253:     return output
4254:
4255:
```

```
4256: __all__ = [
4257:     "Phase6SchemaValidationOutput",
4258:     "ValidatedQuestionSchema",
4259:     "ValidatedChunkSchema",
4260:     "phase6_schema_validation",
4261:     "PHASE6_VERSION",
4262:     "MAX_QUESTION_GLOBAL",
4263: ]
4264:
4265:
4266:
4267: =====
4268: FILE: src/farfan_pipeline/core/phases/phase7_task_construction.py
4269: =====
4270:
4271: """
4272: Phase 7: Task Construction - Deterministic Task Generation with Strict Validation
4273: =====
4274:
4275: This module implements Phase 7 of the canonical pipeline:
4276:     Phase 6 Schema Validation Output \206\222 ExecutableTask Set \206\222 Phase 8 Execution Plan
4277:
4278: Responsibilities:
4279: -----
4280: 1. Generate unique task identifiers with zero-padded three-character format
4281: 2. Construct ExecutableTask instances with complete field validation
4282: 3. Enforce non-null constraints on all mandatory fields
4283: 4. Validate type correctness before any operations assuming type
4284: 5. Handle KeyError and AttributeError explicitly with diagnostic messages
4285: 6. Establish sequential dependencies for hierarchical decomposition
4286: 7. Validate integration points with Phase 6 and Phase 8
4287:
4288: Input Contract (from Phase 6):
4289: -----
4290: Phase6SchemaValidationOutput:
4291:     - validated_questions: List[ValidatedQuestionSchema]
4292:     - validated_chunks: List[ValidatedChunkSchema]
4293:     - schema_validation_passed: bool
4294:     - validation_errors: List[str]
4295:     - validation_warnings: List[str]
4296:     - question_count: int
4297:     - chunk_count: int
4298:     - validation_timestamp: str
4299:
4300: ValidatedQuestionSchema:
4301:     - question_id: str (non-null, validated format)
4302:     - question_global: int (0-999, validated)
4303:     - dimension_id: str (non-null)
4304:     - policy_area_id: str (non-null)
4305:     - base_slot: str (non-null)
4306:     - cluster_id: str (non-null)
4307:     - patterns: List[Dict[str, Any]] (validated schema)
4308:     - signals: Dict[str, Any] (validated schema)
4309:     - expected_elements: List[Dict[str, Any]] (validated schema)
4310:     - metadata: Dict[str, Any]
4311:
```

```
4312: ValidatedChunkSchema:  
4313:     - chunk_id: str (non-null, validated format)  
4314:     - policy_area_id: str (non-null)  
4315:     - dimension_id: str (non-null)  
4316:     - document_position: int (non-negative)  
4317:     - content: str (non-empty)  
4318:     - metadata: Dict[str, Any]  
4319:  
4320: Output Contract (to Phase 8):  
4321: -----  
4322: Phase7TaskConstructionOutput:  
4323:     - tasks: List[ExecutableTask] (length=300)  
4324:     - task_count: int (must equal 300)  
4325:     - construction_passed: bool  
4326:     - construction_errors: List[str]  
4327:     - construction_warnings: List[str]  
4328:     - duplicate_task_ids: List[str]  
4329:     - missing_fields_by_task: Dict[str, List[str]]  
4330:     - construction_timestamp: str  
4331:     - metadata: Dict[str, Any]  
4332:  
4333: ExecutableTask (from task_planner.py):  
4334:     - task_id: str (format: "MQC-{question_global:03d}_{policy_area_id}")  
4335:     - question_id: str (non-empty)  
4336:     - question_global: int (0-999)  
4337:     - policy_area_id: str (non-empty)  
4338:     - dimension_id: str (non-empty)  
4339:     - chunk_id: str (non-empty)  
4340:     - patterns: List[Dict[str, Any]]  
4341:     - signals: Dict[str, Any]  
4342:     - creation_timestamp: str (ISO 8601)  
4343:     - expected_elements: List[Dict[str, Any]]  
4344:     - metadata: Dict[str, Any]  
4345:  
4346: Phase 7 Sub-phases:  
4347: -----  
4348: 7.1 Identifier Generation:  
4349:     - Input: validated_questions (from Phase 6)  
4350:     - Process: Generate task_id with format "MQC-{question_global:03d}_{policy_area_id}"  
4351:     - Validation: Enforce three-character zero-padding, check duplicates  
4352:     - Output: task_id_map: Dict[str, str]  
4353:     - Error Propagation: Duplicate IDs à\206\222 construction_errors  
4354:  
4355: 7.2 Task Construction:  
4356:     - Input: validated_questions, validated_chunks, task_id_map (from 7.1)  
4357:     - Process: Construct ExecutableTask for each question-chunk pair  
4358:     - Validation: Validate ALL mandatory fields before dataclass constructor  
4359:     - Field Processing Order (deterministic):  
4360:         1. task_id (from task_id_map)  
4361:         2. question_id (from question)  
4362:         3. question_global (validate int type)  
4363:         4. policy_area_id (from question)  
4364:         5. dimension_id (from question)  
4365:         6. chunk_id (from chunk)  
4366:         7. patterns (validate list type)  
4367:         8. signals (validate dict type)
```

```
4368:         9. creation_timestamp (generate ISO 8601)
4369:         10. expected_elements (validate list type)
4370:         11. metadata (construct dict)
4371:     - Output: List[ExecutableTask]
4372:     - Error Propagation: Field validation failures à\206\222 construction_errors
4373:
4374: Integration Points:
4375: -----
4376: Phase 6 à\206\222 Phase 7:
4377:     - Contract: Phase6SchemaValidationOutput
4378:     - Precondition: schema_validation_passed must be True
4379:     - Error: If False, Phase 7 returns empty task list with propagated errors
4380:
4381: Phase 7 à\206\222 Phase 8:
4382:     - Contract: Phase7TaskConstructionOutput
4383:     - Precondition: construction_passed must be True
4384:     - Postcondition: task_count must equal 300
4385:     - Error: If False, Phase 8 aborts with diagnostic error message
4386:
4387: Error Handling:
4388: -----
4389: 1. KeyError Handling:
4390:     - All dictionary access wrapped in try-except
4391:     - Error message format: "Field '{field}' not found in {source}: KeyError"
4392:     - Accumulate errors rather than fail-fast
4393:
4394: 2. AttributeError Handling:
4395:     - All attribute access validated before use
4396:     - Error message format: "Attribute '{attr}' not found in {object}: AttributeError"
4397:     - Accumulate errors rather than fail-fast
4398:
4399: 3. TypeError Handling:
4400:     - Type validation occurs BEFORE operations
4401:     - Error message format: "Field '{field}' has invalid type: expected {expected}, got {actual}"
4402:     - Accumulate errors rather than fail-fast
4403:
4404: 4. ValueError Handling:
4405:     - Range/format validation before use
4406:     - Error message format: "Field '{field}' has invalid value: {value} ({reason})"
4407:     - Accumulate errors rather than fail-fast
4408:
4409: Determinism Guarantees:
4410: -----
4411: 1. Field processing order is fixed and documented
4412: 2. Error messages are consistent and deterministic
4413: 3. Task ID generation is deterministic from input
4414: 4. Metadata construction follows fixed key order
4415: 5. All operations are pure functions with no side effects
4416:
4417: Author: F.A.R.F.A.N Architecture Team
4418: Date: 2025-01-19
4419: Version: 1.0.0
4420: """
4421:
4422: from __future__ import annotations
4423:
```

```
4424: import logging
4425: from dataclasses import dataclass
4426: from datetime import datetime, timezone
4427: from typing import Any
4428:
4429: from farfan_pipeline.core.orchestrator.task_planner import ExecutableTask
4430:
4431: logger = logging.getLogger(__name__)
4432:
4433: PHASE7_VERSION = "1.0.0"
4434: REQUIRED_TASK_COUNT = 300
4435: MAX_QUESTION_GLOBAL = 999
4436: TASK_ID_FORMAT = "MQC-{question_global:03d}_{policy_area_id}"
4437:
4438:
4439: @dataclass
4440: class ValidatedQuestionSchema:
4441:     """Validated question schema from Phase 6."""
4442:
4443:     question_id: str
4444:     question_global: int
4445:     dimension_id: str
4446:     policy_area_id: str
4447:     base_slot: str
4448:     cluster_id: str
4449:     patterns: list[dict[str, Any]]
4450:     signals: dict[str, Any]
4451:     expected_elements: list[dict[str, Any]]
4452:     metadata: dict[str, Any]
4453:
4454:
4455: @dataclass
4456: class ValidatedChunkSchema:
4457:     """Validated chunk schema from Phase 6."""
4458:
4459:     chunk_id: str
4460:     policy_area_id: str
4461:     dimension_id: str
4462:     document_position: int
4463:     content: str
4464:     metadata: dict[str, Any]
4465:
4466:
4467: @dataclass
4468: class Phase6SchemaValidationOutput:
4469:     """Output contract from Phase 6."""
4470:
4471:     validated_questions: list[ValidatedQuestionSchema]
4472:     validated_chunks: list[ValidatedChunkSchema]
4473:     schema_validation_passed: bool
4474:     validation_errors: list[str]
4475:     validation_warnings: list[str]
4476:     question_count: int
4477:     chunk_count: int
4478:     validation_timestamp: str
4479:
```

```
4480:
4481: @dataclass
4482: class Phase7TaskConstructionOutput:
4483:     """Output contract for Phase 7."""
4484:
4485:     tasks: list[ExecutableTask]
4486:     task_count: int
4487:     construction_passed: bool
4488:     construction_errors: list[str]
4489:     construction_warnings: list[str]
4490:     duplicate_task_ids: list[str]
4491:     missing_fields_by_task: dict[str, list[str]]
4492:     construction_timestamp: str
4493:     metadata: dict[str, Any]
4494:     phase7_version: str = PHASE7_VERSION
4495:
4496:
4497: class FieldValidationError(Exception):
4498:     """Raised when field validation fails during task construction."""
4499:
4500:     pass
4501:
4502:
4503: class TaskConstructionError(Exception):
4504:     """Raised when task construction fails."""
4505:
4506:     pass
4507:
4508:
4509: def _validate_field_non_null(
4510:     field_name: str,
4511:     field_value: Any,
4512:     source: str,
4513:     errors: list[str],
4514: ) -> bool:
4515:     """
4516:         Validate that a field is non-null.
4517:
4518:     Args:
4519:         field_name: Name of the field being validated
4520:         field_value: Value to validate
4521:         source: Source context (e.g., "question", "chunk")
4522:         errors: List to accumulate errors
4523:
4524:     Returns:
4525:         True if validation passed, False otherwise
4526:     """
4527:     if field_value is None:
4528:         errors.append(
4529:             f"Field '{field_name}' is None in {source}: non-null constraint violated"
4530:         )
4531:     return False
4532: return True
4533:
4534:
4535: def _validate_field_non_empty_string(
```

```
4536:     field_name: str,
4537:     field_value: Any,
4538:     source: str,
4539:     errors: list[str],
4540: ) -> bool:
4541: """
4542:     Validate that a field is a non-empty string.
4543:
4544:     Type validation occurs BEFORE empty check.
4545:
4546:     Args:
4547:         field_name: Name of the field being validated
4548:         field_value: Value to validate
4549:         source: Source context (e.g., "question", "chunk")
4550:         errors: List to accumulate errors
4551:
4552:     Returns:
4553:         True if validation passed, False otherwise
4554: """
4555: if not isinstance(field_value, str):
4556:     errors.append(
4557:         f"Field '{field_name}' has invalid type in {source}: "
4558:         f"expected str, got {type(field_value).__name__}"
4559:     )
4560: return False
4561:
4562: if not field_value:
4563:     errors.append(
4564:         f"Field '{field_name}' is empty string in {source}: non-empty constraint violated"
4565:     )
4566: return False
4567:
4568: return True
4569:
4570:
4571: def _validate_field_integer_range(
4572:     field_name: str,
4573:     field_value: Any,
4574:     source: str,
4575:     min_value: int,
4576:     max_value: int,
4577:     errors: list[str],
4578: ) -> bool:
4579: """
4580:     Validate that a field is an integer within specified range.
4581:
4582:     Type validation occurs BEFORE range check.
4583:
4584:     Args:
4585:         field_name: Name of the field being validated
4586:         field_value: Value to validate
4587:         source: Source context
4588:         min_value: Minimum allowed value (inclusive)
4589:         max_value: Maximum allowed value (inclusive)
4590:         errors: List to accumulate errors
4591:
```

```
4592:     Returns:
4593:         True if validation passed, False otherwise
4594:     """
4595:     if not isinstance(field_value, int):
4596:         errors.append(
4597:             f"Field '{field_name}' has invalid type in {source}: "
4598:             f"expected int, got {type(field_value).__name__}")
4599:     )
4600:     return False
4601:
4602:     if not (min_value <= field_value <= max_value):
4603:         errors.append(
4604:             f"Field '{field_name}' has invalid value in {source}: "
4605:             f"{field_value} not in range [{min_value}, {max_value}]"
4606:         )
4607:     return False
4608:
4609:     return True
4610:
4611:
4612: def _validate_field_list_type(
4613:     field_name: str,
4614:     field_value: Any,
4615:     source: str,
4616:     errors: list[str],
4617: ) -> bool:
4618:     """
4619:         Validate that a field is a list.
4620:
4621:         Type validation only; does not check contents.
4622:
4623:         Args:
4624:             field_name: Name of the field being validated
4625:             field_value: Value to validate
4626:             source: Source context
4627:             errors: List to accumulate errors
4628:
4629:         Returns:
4630:             True if validation passed, False otherwise
4631:     """
4632:     if not isinstance(field_value, list):
4633:         errors.append(
4634:             f"Field '{field_name}' has invalid type in {source}: "
4635:             f"expected list, got {type(field_value).__name__}"
4636:         )
4637:     return False
4638:
4639:     return True
4640:
4641:
4642: def _validate_field_dict_type(
4643:     field_name: str,
4644:     field_value: Any,
4645:     source: str,
4646:     errors: list[str],
4647: ) -> bool:
```

```
4648: """
4649:     Validate that a field is a dict.
4650:
4651:     Type validation only; does not check contents.
4652:
4653:     Args:
4654:         field_name: Name of the field being validated
4655:         field_value: Value to validate
4656:         source: Source context
4657:         errors: List to accumulate errors
4658:
4659:     Returns:
4660:         True if validation passed, False otherwise
4661: """
4662:     if not isinstance(field_value, dict):
4663:         errors.append(
4664:             f"Field '{field_name}' has invalid type in {source}: "
4665:             f"expected dict, got {type(field_value).__name__}"
4666:         )
4667:     return False
4668:
4669:     return True
4670:
4671:
4672: def _safe_get_dict_field(
4673:     data: dict[str, Any],
4674:     field_name: str,
4675:     source: str,
4676:     errors: list[str],
4677:     default: Any = None,
4678: ) -> Any:
4679: """
4680:     Safely extract field from dictionary with KeyError handling.
4681:
4682:     Args:
4683:         data: Dictionary to extract from
4684:         field_name: Key to extract
4685:         source: Source context for error message
4686:         errors: List to accumulate errors
4687:         default: Default value if key not found and error not appended
4688:
4689:     Returns:
4690:         Field value or default
4691:
4692:     Raises:
4693:         Never raises; accumulates errors instead
4694: """
4695:     try:
4696:         return data[field_name]
4697:     except KeyError:
4698:         errors.append(f"Field '{field_name}' not found in {source}: KeyError")
4699:     return default
4700:
4701:
4702: def _safe_get_attribute(
4703:     obj: Any,
```

```
4704:     attr_name: str,
4705:     source: str,
4706:     errors: list[str],
4707:     default: Any = None,
4708: ) -> Any:
4709: """
4710:     Safely extract attribute from object with AttributeError handling.
4711:
4712:     Args:
4713:         obj: Object to extract from
4714:         attr_name: Attribute to extract
4715:         source: Source context for error message
4716:         errors: List to accumulate errors
4717:         default: Default value if attribute not found
4718:
4719:     Returns:
4720:         Attribute value or default
4721:
4722:     Raises:
4723:         Never raises; accumulates errors instead
4724: """
4725: try:
4726:     return getattr(obj, attr_name)
4727: except AttributeError:
4728:     errors.append(f"Attribute '{attr_name}' not found in {source}: AttributeError")
4729:     return default
4730:
4731:
4732: def _generate_task_id(
4733:     question_global: int,
4734:     policy_area_id: str,
4735:     errors: list[str],
4736: ) -> str:
4737: """
4738:     Generate task ID with three-character zero-padded format.
4739:
4740:     Phase 7.1: Identifier Generation
4741:
4742:     Args:
4743:         question_global: Question number (0-999)
4744:         policy_area_id: Policy area identifier
4745:         errors: List to accumulate errors
4746:
4747:     Returns:
4748:         Task ID in format "MQC-{question_global:03d}_{policy_area_id}"
4749:         Empty string if validation fails
4750: """
4751: if not _validate_field_integer_range(
4752:     "question_global",
4753:     question_global,
4754:     "task_id_generation",
4755:     0,
4756:     MAX_QUESTION_GLOBAL,
4757:     errors,
4758: ):
4759:     return ""
```

```
4760:
4761:     if not _validate_field_non_empty_string(
4762:         "policy_area_id",
4763:         policy_area_id,
4764:         "task_id_generation",
4765:         errors,
4766:     ):
4767:         return ""
4768:
4769:     try:
4770:         task_id = TASK_ID_FORMAT.format(
4771:             question_global=question_global,
4772:             policy_area_id=policy_area_id,
4773:         )
4774:         return task_id
4775:     except (ValueError, KeyError) as e:
4776:         errors.append(f"Task ID formatting failed: {type(e).__name__}: {e}")
4777:         return ""
4778:
4779:
4780: def _check_duplicate_task_ids(
4781:     task_ids: list[str],
4782: ) -> tuple[bool, list[str]]:
4783:     """
4784:         Check for duplicate task IDs.
4785:
4786:         Phase 7.1: Identifier Generation - Duplicate Detection
4787:
4788:         Args:
4789:             task_ids: List of task IDs to check
4790:
4791:         Returns:
4792:             Tuple of (has_duplicates, list_of_duplicate_ids)
4793:     """
4794:     seen: set[str] = set()
4795:     duplicates: set[str] = set()
4796:
4797:     for task_id in task_ids:
4798:         if task_id in seen:
4799:             duplicates.add(task_id)
4800:             seen.add(task_id)
4801:
4802:     return len(duplicates) > 0, sorted(duplicates)
4803:
4804:
4805: def _construct_single_task( # noqa: PLR0911
4806:     question: ValidatedQuestionSchema,
4807:     chunk: ValidatedChunkSchema,
4808:     task_id: str,
4809:     errors: list[str],
4810:     warnings: list[str], # noqa: ARG001
4811: ) -> ExecutableTask | None:
4812:     """
4813:         Construct a single ExecutableTask with complete field validation.
4814:
4815:         Phase 7.2: Task Construction
```

```
4816:  
4817:     Field Processing Order (deterministic):  
4818:         1. task_id (validated)  
4819:         2. question_id (validate non-empty string)  
4820:         3. question_global (validate int type and range)  
4821:         4. policy_area_id (validate non-empty string)  
4822:         5. dimension_id (validate non-empty string)  
4823:         6. chunk_id (validate non-empty string)  
4824:         7. patterns (validate list type)  
4825:         8. signals (validate dict type)  
4826:         9. creation_timestamp (generate ISO 8601)  
4827:        10. expected_elements (validate list type)  
4828:        11. metadata (construct dict with fixed key order)  
4829:  
4830:    All validations occur BEFORE dataclass constructor invocation.  
4831:  
4832:    Args:  
4833:        question: Validated question schema from Phase 6  
4834:        chunk: Validated chunk schema from Phase 6  
4835:        task_id: Pre-generated task ID from Phase 7.1  
4836:        errors: List to accumulate errors  
4837:        warnings: List to accumulate warnings  
4838:  
4839:    Returns:  
4840:        ExecutableTask instance or None if validation failed  
4841:    """  
4842:    task_errors: list[str] = []  
4843:    task_context = f"task[{task_id}]"  
4844:  
4845:    if not _validate_field_non_empty_string(  
4846:        "task_id", task_id, task_context, task_errors  
4847:    ):  
4848:        errors.extend(task_errors)  
4849:    return None  
4850:  
4851:    question_id = _safe_get_attribute(  
4852:        question, "question_id", task_context, task_errors, ""  
4853:    )  
4854:    if not _validate_field_non_empty_string(  
4855:        "question_id", question_id, task_context, task_errors  
4856:    ):  
4857:        errors.extend(task_errors)  
4858:    return None  
4859:  
4860:    question_global = _safe_get_attribute(  
4861:        question, "question_global", task_context, task_errors, -1  
4862:    )  
4863:    if not _validate_field_integer_range(  
4864:        "question_global",  
4865:        question_global,  
4866:        task_context,  
4867:        0,  
4868:        MAX_QUESTION_GLOBAL,  
4869:        task_errors,  
4870:    ):  
4871:        errors.extend(task_errors)
```

```
4872:         return None
4873:
4874:     policy_area_id = _safe_get_attribute(
4875:         question, "policy_area_id", task_context, task_errors, ""
4876:     )
4877:     if not _validate_field_non_empty_string(
4878:         "policy_area_id", policy_area_id, task_context, task_errors
4879:     ):
4880:         errors.extend(task_errors)
4881:     return None
4882:
4883:     dimension_id = _safe_get_attribute(
4884:         question, "dimension_id", task_context, task_errors, ""
4885:     )
4886:     if not _validate_field_non_empty_string(
4887:         "dimension_id", dimension_id, task_context, task_errors
4888:     ):
4889:         errors.extend(task_errors)
4890:     return None
4891:
4892:     chunk_id = _safe_get_attribute(chunk, "chunk_id", task_context, task_errors, "")
4893:     if not _validate_field_non_empty_string(
4894:         "chunk_id", chunk_id, task_context, task_errors
4895:     ):
4896:         errors.extend(task_errors)
4897:     return None
4898:
4899:     patterns = _safe_get_attribute(question, "patterns", task_context, task_errors, [])
4900:     if not _validate_field_list_type("patterns", patterns, task_context, task_errors):
4901:         errors.extend(task_errors)
4902:     return None
4903:
4904:     signals = _safe_get_attribute(question, "signals", task_context, task_errors, {})
4905:     if not _validate_field_dict_type("signals", signals, task_context, task_errors):
4906:         errors.extend(task_errors)
4907:     return None
4908:
4909:     creation_timestamp = datetime.now(timezone.utc).isoformat()
4910:
4911:     expected_elements = _safe_get_attribute(
4912:         question, "expected_elements", task_context, task_errors, []
4913:     )
4914:     if not _validate_field_list_type(
4915:         "expected_elements", expected_elements, task_context, task_errors
4916:     ):
4917:         errors.extend(task_errors)
4918:     return None
4919:
4920:     question_metadata = _safe_get_attribute(
4921:         question, "metadata", task_context, task_errors, {}
4922:     )
4923:     chunk_metadata = _safe_get_attribute(
4924:         chunk, "metadata", task_context, task_errors, {}
4925:     )
4926:
4927:     base_slot = _safe_get_attribute(question, "base_slot", task_context, [], "")
```

```
4928:     cluster_id = _safe_get_attribute(question, "cluster_id", task_context, [], "")
4929:     document_position = _safe_get_attribute(
4930:         chunk, "document_position", task_context, [], 0
4931:     )
4932:
4933:     metadata = {
4934:         "base_slot",
4935:         "cluster_id": cluster_id,
4936:         "document_position": document_position,
4937:         "phase7_version": PHASE7_VERSION,
4938:         "question_metadata": (
4939:             question_metadata if isinstance(question_metadata, dict) else {}
4940:         ),
4941:         "chunk_metadata": chunk_metadata if isinstance(chunk_metadata, dict) else {},
4942:         "pattern_count": len(patterns) if isinstance(patterns, list) else 0,
4943:         "signal_count": len(signals) if isinstance(signals, dict) else 0,
4944:         "expected_element_count": (
4945:             len(expected_elements) if isinstance(expected_elements, list) else 0
4946:         ),
4947:     }
4948:
4949:     try:
4950:         task = ExecutableTask(
4951:             task_id=task_id,
4952:             question_id=question_id,
4953:             question_global=question_global,
4954:             policy_area_id=policy_area_id,
4955:             dimension_id=dimension_id,
4956:             chunk_id=chunk_id,
4957:             patterns=patterns,
4958:             signals=signals,
4959:             creation_timestamp=creation_timestamp,
4960:             expected_elements=expected_elements,
4961:             metadata=metadata,
4962:         )
4963:
4964:         logger.debug(
4965:             f"Successfully constructed task: {task_id} "
4966:             f"(question={question_id}, chunk={chunk_id})"
4967:         )
4968:
4969:         return task
4970:
4971:     except (ValueError, TypeError) as e:
4972:         error_msg = (
4973:             f"ExecutableTask dataclass constructor failed for {task_id}: "
4974:             f"{type(e).__name__}: {e}"
4975:         )
4976:         errors.append(error_msg)
4977:         logger.error(error_msg)
4978:         return None
4979:
4980:
4981: def phase7_task_construction( # noqa: PLR0912, PLR0915
4982:     phase6_output: Phase6SchemaValidationOutput,
4983: ) -> Phase7TaskConstructionOutput:
```

```
4984: """
4985: Execute Phase 7: Task Construction.
4986:
4987: Hierarchical Decomposition:
4988: 1. Phase 7.1: Identifier Generation
4989:     - Generate task IDs with three-character zero-padding
4990:     - Check for duplicates
4991:     - Accumulate errors
4992:
4993: 2. Phase 7.2: Task Construction
4994:     - Validate all mandatory fields in deterministic order
4995:     - Construct ExecutableTask instances
4996:     - Accumulate errors and track missing fields
4997:
4998: Integration Points:
4999: - Input: Phase6SchemaValidationOutput (requires schema_validation_passed=True)
5000: - Output: Phase7TaskConstructionOutput (provides tasks for Phase 8)
5001:
5002: Error Propagation:
5003: - Phase 6 failures \206\222 construction_passed=False, empty task list
5004: - Field validation failures \206\222 accumulated in construction_errors
5005: - Duplicate task IDs \206\222 accumulated in duplicate_task_ids
5006: - Missing fields \206\222 tracked in missing_fields_by_task
5007:
5008: Args:
5009:     phase6_output: Validated output from Phase 6
5010:
5011: Returns:
5012:     Phase7TaskConstructionOutput with constructed tasks or error details
5013: """
5014: logger.info("Phase 7: Task Construction started")
5015:
5016: construction_errors: list[str] = []
5017: construction_warnings: list[str] = []
5018: tasks: list[ExecutableTask] = []
5019: duplicate_task_ids: list[str] = []
5020: missing_fields_by_task: dict[str, list[str]] = {}
5021:
5022: if not phase6_output.schema_validation_passed:
5023:     construction_errors.append(
5024:         "Phase 6 schema validation failed; cannot proceed with task construction"
5025:     )
5026:     construction_errors.extend(phase6_output.validation_errors)
5027:
5028: return Phase7TaskConstructionOutput(
5029:     tasks=[],
5030:     task_count=0,
5031:     construction_passed=False,
5032:     construction_errors=construction_errors,
5033:     construction_warnings=construction_warnings,
5034:     duplicate_task_ids=[],
5035:     missing_fields_by_task={},
5036:     construction_timestamp=datetime.now(timezone.utc).isoformat(),
5037:     metadata={
5038:         "phase6_errors_propagated": len(phase6_output.validation_errors),
5039:         "phase6_warnings_propagated": len(phase6_output.validation_warnings),
```

```
5040:         },
5041:     )
5042:
5043:     validated_questions = phase6_output.validated_questions
5044:     validated_chunks = phase6_output.validated_chunks
5045:
5046:     if not validated_questions:
5047:         construction_errors.append("No validated questions provided by Phase 6")
5048:
5049:     if not validated_chunks:
5050:         construction_errors.append("No validated chunks provided by Phase 6")
5051:
5052:     if construction_errors:
5053:         return Phase7TaskConstructionOutput(
5054:             tasks=[],
5055:             task_count=0,
5056:             construction_passed=False,
5057:             construction_errors=construction_errors,
5058:             construction_warnings=construction_warnings,
5059:             duplicate_task_ids=[],
5060:             missing_fields_by_task={},
5061:             construction_timestamp=datetime.now(timezone.utc).isoformat(),
5062:             metadata={}
5063:         )
5064:
5065:     logger.info(
5066:         f"Phase 7.1: Identifier Generation started "
5067:         f"({len(validated_questions)} questions"
5068:     )
5069:
5070:     task_id_map: dict[str, str] = {}
5071:     task_ids_list: list[str] = []
5072:
5073:     for question in validated_questions:
5074:         question_global = _safe_get_attribute(
5075:             question, "question_global", "phase7.1", construction_errors, -1
5076:         )
5077:         policy_area_id = _safe_get_attribute(
5078:             question, "policy_area_id", "phase7.1", construction_errors, ""
5079:         )
5080:
5081:         task_id = _generate_task_id(
5082:             question_global, policy_area_id, construction_errors
5083:         )
5084:
5085:         if task_id:
5086:             question_id = _safe_get_attribute(
5087:                 question, "question_id", "phase7.1", construction_errors, ""
5088:             )
5089:             task_id_map[question_id] = task_id
5090:             task_ids_list.append(task_id)
5091:
5092:     has_duplicates, duplicate_list = _check_duplicate_task_ids(task_ids_list)
5093:     if has_duplicates:
5094:         duplicate_task_ids = duplicate_list
5095:         construction_errors.append(f"Duplicate task IDs detected: {duplicate_task_ids}")
```

```
5096:  
5097:     logger.info(  
5098:         f"Phase 7.1: Generated {len(task_id_map)} task IDs "  
5099:         f"(duplicates: {len(duplicate_task_ids)})"  
5100:     )  
5101:  
5102:     logger.info("Phase 7.2: Task Construction started")  
5103:  
5104:     for question in validated_questions:  
5105:         question_id = _safe_get_attribute(  
5106:             question, "question_id", "phase7.2", construction_errors, "")  
5107:     )  
5108:  
5109:     task_id = task_id_map.get(question_id, "")  
5110:     if not task_id:  
5111:         construction_errors.append(  
5112:             f"Task ID not found in task_id_map for question {question_id}")  
5113:     )  
5114:     continue  
5115:  
5116:     policy_area_id = _safe_get_attribute(  
5117:         question, "policy_area_id", "phase7.2", construction_errors, "")  
5118:     )  
5119:     dimension_id = _safe_get_attribute(  
5120:         question, "dimension_id", "phase7.2", construction_errors, "")  
5121:     )  
5122:  
5123:     matching_chunks = [  
5124:         chunk  
5125:         for chunk in validated_chunks  
5126:         if chunk.policy_area_id == policy_area_id  
5127:         and chunk.dimension_id == dimension_id  
5128:     ]  
5129:  
5130:     if not matching_chunks:  
5131:         construction_warnings.append(  
5132:             f"No matching chunks found for question {question_id} "  
5133:             f"(PA={policy_area_id}, DIM={dimension_id})"  
5134:         )  
5135:         continue  
5136:  
5137:     chunk = matching_chunks[0]  
5138:  
5139:     task = _construct_single_task(  
5140:         question=question,  
5141:         chunk=chunk,  
5142:         task_id=task_id,  
5143:         errors=construction_errors,  
5144:         warnings=construction_warnings,  
5145:     )  
5146:  
5147:     if task:  
5148:         tasks.append(task)  
5149:     else:  
5150:         missing_fields = [  
5151:             err.split("\n")[1]
```

```
5152:             for err in construction_errors
5153:                 if f"task[{task_id}]" in err and "not found" in err
5154:             ]
5155:             if missing_fields:
5156:                 missing_fields_by_task[task_id] = missing_fields
5157:
5158: logger.info(f"Phase 7.2: Constructed {len(tasks)} tasks")
5159:
5160: task_count = len(tasks)
5161: construction_passed =
5162:     len(construction_errors) == 0 and task_count == REQUIRED_TASK_COUNT
5163: )
5164:
5165: if task_count != REQUIRED_TASK_COUNT:
5166:     construction_errors.append(
5167:         f"Task count mismatch: expected {REQUIRED_TASK_COUNT}, got {task_count}"
5168:     )
5169:
5170: construction_timestamp = datetime.now(timezone.utc).isoformat()
5171:
5172: metadata = {
5173:     "question_count": len(validated_questions),
5174:     "chunk_count": len(validated_chunks),
5175:     "task_id_map_size": len(task_id_map),
5176:     "duplicate_count": len(duplicate_task_ids),
5177:     "missing_field_task_count": len(missing_fields_by_task),
5178:     "construction_duration_ms": 0,
5179: }
5180:
5181: output = Phase7TaskConstructionOutput(
5182:     tasks=tasks,
5183:     task_count=task_count,
5184:     construction_passed=construction_passed,
5185:     construction_errors=construction_errors,
5186:     construction_warnings=construction_warnings,
5187:     duplicate_task_ids=duplicate_task_ids,
5188:     missing_fields_by_task=missing_fields_by_task,
5189:     construction_timestamp=construction_timestamp,
5190:     metadata=metadata,
5191: )
5192:
5193: logger.info(
5194:     f"Phase 7: Task Construction completed "
5195:     f"(passed={construction_passed}, tasks={task_count}, "
5196:     f"errors={len(construction_errors)}, warnings={len(construction_warnings)})"
5197: )
5198:
5199: return output
5200:
5201:
5202: __all__ = [
5203:     "Phase6SchemaValidationOutput",
5204:     "Phase7TaskConstructionOutput",
5205:     "ValidatedQuestionSchema",
5206:     "ValidatedChunkSchema",
5207:     "phase7_task_construction",
```

```
5208:     "PHASE7_VERSION",
5209:     "REQUIRED_TASK_COUNT",
5210:     "MAX_QUESTION_GLOBAL",
5211:     "TASK_ID_FORMAT",
5212: ]
5213:
5214:
5215:
5216: =====
5217: FILE: src/farfán_pipeline/core/phases/phase8_execution_plan.py
5218: =====
5219:
5220: """
5221: Phase 8: Execution Plan Assembly - Deterministic Plan Construction with Integrity Verification
5222: =====
5223:
5224: This module implements Phase 8 of the canonical pipeline:
5225:     Phase 7 Task Construction Output → ExecutionPlan → Phase 9+ Execution
5226:
5227: Responsibilities:
5228: -----
5229: 1. Assemble validated ExecutableTask instances into ExecutionPlan
5230: 2. Verify exactly 300 tasks are present (canonical contract)
5231: 3. Compute cryptographic integrity hash (BLAKE3 or SHA256)
5232: 4. Detect duplicate task IDs across the complete plan
5233: 5. Validate execution dependencies and ordering constraints
5234: 6. Generate execution metadata and provenance information
5235: 7. Establish error propagation semantics to downstream phases
5236:
5237: Input Contract (from Phase 7):
5238: -----
5239: Phase7TaskConstructionOutput:
5240:     - tasks: List[ExecutableTask] (validated instances)
5241:     - task_count: int (must equal 300)
5242:     - construction_passed: bool (must be True)
5243:     - construction_errors: List[str]
5244:     - construction_warnings: List[str]
5245:     - duplicate_task_ids: List[str]
5246:     - missing_fields_by_task: Dict[str, List[str]]
5247:     - construction_timestamp: str
5248:     - metadata: Dict[str, Any]
5249:
5250: Output Contract (to Phase 9):
5251: -----
5252: Phase8ExecutionPlanOutput:
5253:     - execution_plan: ExecutionPlan | None
5254:     - plan_id: str
5255:     - task_count: int
5256:     - assembly_passed: bool
5257:     - assembly_errors: List[str]
5258:     - assembly_warnings: List[str]
5259:     - integrity_hash: str (BLAKE3 or SHA256)
5260:     - duplicate_tasks: List[str]
5261:     - assembly_timestamp: str
5262:     - metadata: Dict[str, Any]
5263:
```

```
5264: ExecutionPlan:  
5265:     - plan_id: str (unique identifier)  
5266:     - tasks: Tuple[ExecutableTask, ...] (immutable, length=300)  
5267:     - creation_timestamp: str (ISO 8601)  
5268:     - integrity_hash: str (cryptographic hash)  
5269:     - metadata: Dict[str, Any]  
5270:  
5271: Phase 8 Sub-phases:  
5272: -----  
5273: 8.1 Precondition Validation:  
5274:     - Input: Phase7TaskConstructionOutput  
5275:     - Process: Verify construction_passed=True, task_count=300  
5276:     - Output: validation_result: bool  
5277:     - Error Propagation: Failed precondition à\206\222 assembly_passed=False  
5278:  
5279: 8.2 Duplicate Detection:  
5280:     - Input: tasks (from Phase 7)  
5281:     - Process: Check for duplicate task_ids across all tasks  
5282:     - Output: duplicate_list: List[str]  
5283:     - Error Propagation: Duplicates found à\206\222 assembly_passed=False  
5284:  
5285: 8.3 Execution Plan Construction:  
5286:     - Input: validated tasks, plan_id  
5287:     - Process: Construct immutable ExecutionPlan with tuple conversion  
5288:     - Output: ExecutionPlan instance  
5289:     - Error Propagation: Construction failures à\206\222 assembly_passed=False  
5290:  
5291: 8.4 Integrity Hash Computation:  
5292:     - Input: ExecutionPlan  
5293:     - Process: Compute BLAKE3/SHA256 hash of serialized task data  
5294:     - Output: integrity_hash: str  
5295:     - Error Propagation: Hash computation failures à\206\222 assembly_warning  
5296:  
5297: Integration Points:  
5298: -----  
5299: Phase 7 à\206\222 Phase 8:  
5300:     - Contract: Phase7TaskConstructionOutput  
5301:     - Precondition: construction_passed must be True  
5302:     - Error: If False, Phase 8 returns None execution_plan with propagated errors  
5303:  
5304: Phase 8 à\206\222 Phase 9:  
5305:     - Contract: Phase8ExecutionPlanOutput  
5306:     - Precondition: assembly_passed must be True  
5307:     - Postcondition: execution_plan is not None, task_count=300  
5308:     - Error: If False, Phase 9+ abort with diagnostic error message  
5309:  
5310: Error Propagation Semantics:  
5311: -----  
5312: 1. Phase 7 construction failures à\206\222 Phase 8 returns early with propagated errors  
5313: 2. Task count != 300 à\206\222 assembly_passed=False, execution_plan=None  
5314: 3. Duplicate task IDs à\206\222 assembly_passed=False, execution_plan=None  
5315: 4. ExecutionPlan construction failures à\206\222 assembly_passed=False, execution_plan=None  
5316: 5. Integrity hash failures à\206\222 assembly_warning (plan still usable)  
5317:  
5318: All errors accumulate rather than fail-fast to provide complete diagnostics.  
5319:
```

```
5320: Author: F.A.R.F.A.N Architecture Team
5321: Date: 2025-01-19
5322: Version: 1.0.0
5323: """
5324:
5325: from __future__ import annotations
5326:
5327: import hashlib
5328: import json
5329: import logging
5330: from dataclasses import dataclass
5331: from datetime import datetime, timezone
5332: from typing import Any
5333: from uuid import uuid4
5334:
5335: from farfan_pipeline.core.orchestrator.task_planner import (
5336:     ExecutableTask, # noqa: TC001
5337: )
5338:
5339: logger = logging.getLogger(__name__)
5340:
5341: PHASE8_VERSION = "1.0.0"
5342: REQUIRED_TASK_COUNT = 300
5343:
5344:
5345: @dataclass
5346: class ExecutionPlan:
5347: """
5348:     Immutable execution plan with exactly 300 tasks.
5349:
5350:     Enforces the 300-question analysis contract (D1-D6 dimensions \227 PA01-PA10 areas)
5351:     with cryptographic integrity verification.
5352: """
5353:
5354:     plan_id: str
5355:     tasks: tuple[ExecutableTask, ...]
5356:     creation_timestamp: str
5357:     integrity_hash: str
5358:     metadata: dict[str, Any]
5359:
5360:     def __post_init__(self) -> None:
5361:         if len(self.tasks) != REQUIRED_TASK_COUNT:
5362:             raise ValueError(
5363:                 f"ExecutionPlan requires exactly {REQUIRED_TASK_COUNT} tasks, "
5364:                 f"got {len(self.tasks)}"
5365:             )
5366:
5367:         task_ids = [task.task_id for task in self.tasks]
5368:         if len(task_ids) != len(set(task_ids)):
5369:             seen: set[str] = set()
5370:             duplicates: set[str] = set()
5371:             for task_id in task_ids:
5372:                 if task_id in seen:
5373:                     duplicates.add(task_id)
5374:                     seen.add(task_id)
5375:             raise ValueError(
```

```
5376:             f"ExecutionPlan contains duplicate task_ids: {sorted(duplicates)}"
```

```
5377:         )
```

```
5378:
```

```
5379:
```

```
5380: @dataclass
```

```
5381: class Phase7TaskConstructionOutput:
```

```
5382:     """Input contract from Phase 7."""
```

```
5383:
```

```
5384:     tasks: list[ExecutableTask]
```

```
5385:     task_count: int
```

```
5386:     construction_passed: bool
```

```
5387:     construction_errors: list[str]
```

```
5388:     construction_warnings: list[str]
```

```
5389:     duplicate_task_ids: list[str]
```

```
5390:     missing_fields_by_task: dict[str, list[str]]
```

```
5391:     construction_timestamp: str
```

```
5392:     metadata: dict[str, Any]
```

```
5393:
```

```
5394:
```

```
5395: @dataclass
```

```
5396: class Phase8ExecutionPlanOutput:
```

```
5397:     """Output contract for Phase 8."""
```

```
5398:
```

```
5399:     execution_plan: ExecutionPlan | None
```

```
5400:     plan_id: str
```

```
5401:     task_count: int
```

```
5402:     assembly_passed: bool
```

```
5403:     assembly_errors: list[str]
```

```
5404:     assembly_warnings: list[str]
```

```
5405:     integrity_hash: str
```

```
5406:     duplicate_tasks: list[str]
```

```
5407:     assembly_timestamp: str
```

```
5408:     metadata: dict[str, Any]
```

```
5409:     phase8_version: str = PHASE8_VERSION
```

```
5410:
```

```
5411:
```

```
5412: def _validate_phase7_preconditions(
```

```
5413:     phase7_output: Phase7TaskConstructionOutput,
```

```
5414:     errors: list[str],
```

```
5415: ) -> bool:
```

```
5416:     """
```

```
5417:     Validate Phase 7 preconditions.
```

```
5418:
```

```
5419:     Phase 8.1: Precondition Validation
```

```
5420:
```

```
5421:     Args:
```

```
5422:         phase7_output: Output from Phase 7
```

```
5423:         errors: List to accumulate errors
```

```
5424:
```

```
5425:     Returns:
```

```
5426:         True if preconditions satisfied, False otherwise
```

```
5427:     """
```

```
5428:     preconditions_met = True
```

```
5429:
```

```
5430:     if not phase7_output.construction_passed:
```

```
5431:         errors.append(
```

```
5432:         "Phase 7 construction failed; cannot proceed with execution plan assembly"
5433:     )
5434:     errors.extend(phase7_output.construction_errors)
5435:     preconditions_met = False
5436:
5437: if phase7_output.task_count != REQUIRED_TASK_COUNT:
5438:     errors.append(
5439:         f"Phase 7 task count mismatch: expected {REQUIRED_TASK_COUNT}, "
5440:         f"got {phase7_output.task_count}"
5441:     )
5442:     preconditions_met = False
5443:
5444: if not isinstance(phase7_output.tasks, list):
5445:     errors.append(
5446:         f"Phase 7 tasks has invalid type: expected list, "
5447:         f"got {type(phase7_output.tasks).__name__}"
5448:     )
5449:     preconditions_met = False
5450:
5451: if phase7_output.duplicate_task_ids:
5452:     errors.append(
5453:         f"Phase 7 reported duplicate task IDs: {phase7_output.duplicate_task_ids}"
5454:     )
5455:     preconditions_met = False
5456:
5457: return preconditions_met
5458:
5459:
5460: def _detect_duplicate_tasks(
5461:     tasks: list[ExecutableTask],
5462: ) -> tuple[bool, list[str]]:
5463:     """
5464:     Detect duplicate task IDs in task list.
5465:
5466:     Phase 8.2: Duplicate Detection
5467:
5468:     Args:
5469:         tasks: List of tasks to check
5470:
5471:     Returns:
5472:         Tuple of (has_duplicates, list_of_duplicate_ids)
5473:     """
5474:     seen: set[str] = set()
5475:     duplicates: set[str] = set()
5476:
5477:     for task in tasks:
5478:         task_id = task.task_id
5479:         if task_id in seen:
5480:             duplicates.add(task_id)
5481:             seen.add(task_id)
5482:
5483:     return len(duplicates) > 0, sorted(duplicates)
5484:
5485:
5486: def _compute_integrity_hash(
5487:     tasks: tuple[ExecutableTask, ...],
```

```
5488:     warnings: list[str],
5489: ) -> str:
5490:     """
5491:     Compute cryptographic integrity hash of task data.
5492:
5493:     Phase 8.4: Integrity Hash Computation
5494:
5495:     Uses SHA256 for deterministic serialization of task data.
5496:     Failures produce warning and return empty string.
5497:
5498:     Args:
5499:         tasks: Tuple of tasks to hash
5500:         warnings: List to accumulate warnings
5501:
5502:     Returns:
5503:         Hexadecimal hash string or empty string on failure
5504:     """
5505:     try:
5506:         task_data = [
5507:             {
5508:                 "task_id": task.task_id,
5509:                 "question_id": task.question_id,
5510:                 "question_global": task.question_global,
5511:                 "policy_area_id": task.policy_area_id,
5512:                 "dimension_id": task.dimension_id,
5513:                 "chunk_id": task.chunk_id,
5514:                 "creation_timestamp": task.creation_timestamp,
5515:             }
5516:             for task in tasks
5517:         ]
5518:
5519:         serialized = json.dumps(task_data, sort_keys=True, separators=(", ", ":" ))
5520:         hash_value = hashlib.sha256(serialized.encode("utf-8")).hexdigest()
5521:
5522:         logger.debug(f"Computed integrity hash: {hash_value[:16]}...")
5523:     return hash_value
5524:
5525: except (TypeError, ValueError, AttributeError) as e:
5526:     warning_msg = f"Integrity hash computation failed: {type(e).__name__}: {e}"
5527:     warnings.append(warning_msg)
5528:     logger.warning(warning_msg)
5529:     return ""
5530:
5531:
5532: def _construct_execution_plan(
5533:     plan_id: str,
5534:     tasks: list[ExecutableTask],
5535:     errors: list[str],
5536:     warnings: list[str],
5537: ) -> ExecutionPlan | None:
5538:     """
5539:     Construct immutable ExecutionPlan instance.
5540:
5541:     Phase 8.3: Execution Plan Construction
5542:
5543:     Args:
```

```
5544:     plan_id: Unique plan identifier
5545:     tasks: List of validated tasks
5546:     errors: List to accumulate errors
5547:     warnings: List to accumulate warnings
5548:
5549:     Returns:
5550:         ExecutionPlan instance or None if construction failed
5551:     """
5552:     try:
5553:         tasks_tuple = tuple(tasks)
5554:
5555:         creation_timestamp = datetime.now(timezone.utc).isoformat()
5556:
5557:         integrity_hash = _compute_integrity_hash(tasks_tuple, warnings)
5558:
5559:         metadata = {
5560:             "phase8_version": PHASE8_VERSION,
5561:             "task_count": len(tasks_tuple),
5562:             "has_integrity_hash": bool(integrity_hash),
5563:             "creation_timestamp": creation_timestamp,
5564:         }
5565:
5566:         execution_plan = ExecutionPlan(
5567:             plan_id=plan_id,
5568:             tasks=tasks_tuple,
5569:             creation_timestamp=creation_timestamp,
5570:             integrity_hash=integrity_hash,
5571:             metadata=metadata,
5572:         )
5573:
5574:         logger.debug(f"Successfully constructed execution plan: {plan_id}")
5575:         return execution_plan
5576:
5577:     except (ValueError, TypeError) as e:
5578:         error_msg = f"ExecutionPlan construction failed: {type(e).__name__}: {e}"
5579:         errors.append(error_msg)
5580:         logger.error(error_msg)
5581:         return None
5582:
5583:
5584: def phase8_execution_plan_assembly(
5585:     phase7_output: Phase7TaskConstructionOutput,
5586:     plan_id: str | None = None,
5587: ) -> Phase8ExecutionPlanOutput:
5588:     """
5589:         Execute Phase 8: Execution Plan Assembly.
5590:
5591:         Hierarchical Decomposition:
5592:             1. Phase 8.1: Precondition Validation
5593:                 - Verify Phase 7 construction_passed=True
5594:                 - Verify task_count=300
5595:                 - Check for duplicate_task_ids from Phase 7
5596:
5597:             2. Phase 8.2: Duplicate Detection
5598:                 - Scan all tasks for duplicate task_ids
5599:                 - Accumulate duplicates in error list
```

```
5600:
5601:    3. Phase 8.3: Execution Plan Construction
5602:        - Convert task list to immutable tuple
5603:        - Construct ExecutionPlan instance
5604:        - Handle construction failures
5605:
5606:    4. Phase 8.4: Integrity Hash Computation
5607:        - Compute SHA256 hash of serialized tasks
5608:        - Attach hash to execution plan
5609:        - Warn on hash computation failures
5610:
5611:    Integration Points:
5612:        - Input: Phase7TaskConstructionOutput (requires construction_passed=True)
5613:        - Output: Phase8ExecutionPlanOutput (provides execution_plan for Phase 9+)
5614:
5615:    Error Propagation:
5616:        - Phase 7 failures \206\222 assembly_passed=False, execution_plan=None
5617:        - Precondition failures \206\222 accumulated in assembly_errors
5618:        - Duplicate detection failures \206\222 accumulated in assembly_errors
5619:        - Construction failures \206\222 accumulated in assembly_errors
5620:        - Hash failures \206\222 accumulated in assembly_warnings (non-fatal)
5621:
5622:    Args:
5623:        phase7_output: Validated output from Phase 7
5624:        plan_id: Optional plan identifier (generates UUID if not provided)
5625:
5626:    Returns:
5627:        Phase8ExecutionPlanOutput with execution plan or error details
5628:    """
5629:    logger.info("Phase 8: Execution Plan Assembly started")
5630:
5631:    assembly_errors: list[str] = []
5632:    assembly_warnings: list[str] = []
5633:    execution_plan: ExecutionPlan | None = None
5634:    duplicate_tasks: list[str] = []
5635:    integrity_hash = ""
5636:
5637:    if plan_id is None:
5638:        plan_id = f"plan-{uuid4().hex[:16]}"
5639:
5640:    logger.info(f"Phase 8.1: Precondition Validation (plan_id={plan_id})")
5641:
5642:    preconditions_met = _validate_phase7_preconditions(phase7_output, assembly_errors)
5643:
5644:    if not preconditions_met:
5645:        logger.error(
5646:            f"Phase 8.1: Precondition validation failed "
5647:            f"({len(assembly_errors)} errors)"
5648:        )
5649:
5650:    return Phase8ExecutionPlanOutput(
5651:        execution_plan=None,
5652:        plan_id=plan_id,
5653:        task_count=phase7_output.task_count,
5654:        assembly_passed=False,
5655:        assembly_errors=assembly_errors,
```

```
5656:         assembly_warnings=assembly_warnings,
5657:         integrity_hash="",
5658:         duplicate_tasks=[],
5659:         assembly_timestamp=datetime.now(timezone.utc).isoformat(),
5660:         metadata={}
5661:             "phase7_errors_propagated": len(phase7_output.construction_errors),
5662:             "phase7_warnings_propagated": len(phase7_output.construction_warnings),
5663:         },
5664:     )
5665:
5666:     logger.info(f"Phase 8.2: Duplicate Detection ({len(phase7_output.tasks)} tasks)")
5667:
5668:     has_duplicates, duplicate_list = _detect_duplicate_tasks(phase7_output.tasks)
5669:
5670:     if has_duplicates:
5671:         duplicate_tasks = duplicate_list
5672:         assembly_errors.append(
5673:             f"Duplicate task IDs detected in Phase 8: {duplicate_tasks}"
5674:         )
5675:         logger.error(f"Phase 8.2: Found {len(duplicate_tasks)} duplicate task IDs")
5676:     else:
5677:         logger.info("Phase 8.2: No duplicate task IDs detected")
5678:
5679:     if assembly_errors:
5680:         logger.error(
5681:             f"Phase 8: Cannot proceed with plan construction "
5682:             f"({len(assembly_errors)} errors)"
5683:         )
5684:
5685:     return Phase8ExecutionPlanOutput(
5686:         execution_plan=None,
5687:         plan_id=plan_id,
5688:         task_count=phase7_output.task_count,
5689:         assembly_passed=False,
5690:         assembly_errors=assembly_errors,
5691:         assembly_warnings=assembly_warnings,
5692:         integrity_hash="",
5693:         duplicate_tasks=duplicate_tasks,
5694:         assembly_timestamp=datetime.now(timezone.utc).isoformat(),
5695:         metadata={},
5696:     )
5697:
5698:     logger.info("Phase 8.3: Execution Plan Construction")
5699:
5700:     execution_plan = _construct_execution_plan(
5701:         plan_id=plan_id,
5702:         tasks=phase7_output.tasks,
5703:         errors=assembly_errors,
5704:         warnings=assembly_warnings,
5705:     )
5706:
5707:     if execution_plan is None:
5708:         logger.error("Phase 8.3: Execution plan construction failed")
5709:
5710:     return Phase8ExecutionPlanOutput(
5711:         execution_plan=None,
```

```
5712:         plan_id=plan_id,
5713:         task_count=phase7_output.task_count,
5714:         assembly_passed=False,
5715:         assembly_errors=assembly_errors,
5716:         assembly_warnings=assembly_warnings,
5717:         integrity_hash="",
5718:         duplicate_tasks=duplicate_tasks,
5719:         assembly_timestamp=datetime.now(timezone.utc).isoformat(),
5720:         metadata={},
5721:     )
5722:
5723:     integrity_hash = execution_plan.integrity_hash
5724:
5725:     logger.info(
5726:         f"Phase 8.3: Successfully constructed execution plan "
5727:         f"(integrity_hash={integrity_hash[:16]})...")
5728:     )
5729:
5730:     assembly_passed = len(assembly_errors) == 0
5731:
5732:     assembly_timestamp = datetime.now(timezone.utc).isoformat()
5733:
5734:     metadata = {
5735:         "plan_id": plan_id,
5736:         "task_count": len(execution_plan.tasks),
5737:         "integrity_hash_length": len(integrity_hash),
5738:         "phase7_warnings_count": len(phase7_output.construction_warnings),
5739:         "assembly_duration_ms": 0,
5740:     }
5741:
5742:     output = Phase8ExecutionPlanOutput(
5743:         execution_plan=execution_plan,
5744:         plan_id=plan_id,
5745:         task_count=len(execution_plan.tasks),
5746:         assembly_passed=assembly_passed,
5747:         assembly_errors=assembly_errors,
5748:         assembly_warnings=assembly_warnings,
5749:         integrity_hash=integrity_hash,
5750:         duplicate_tasks=duplicate_tasks,
5751:         assembly_timestamp=assembly_timestamp,
5752:         metadata=metadata,
5753:     )
5754:
5755:     logger.info(
5756:         f"Phase 8: Execution Plan Assembly completed "
5757:         f"(passed={assembly_passed}, tasks={len(execution_plan.tasks)}, "
5758:         f"errors={len(assembly_errors)}, warnings={len(assembly_warnings)})"
5759:     )
5760:
5761:     return output
5762:
5763:
5764: __all__ = [
5765:     "ExecutionPlan",
5766:     "Phase7TaskConstructionOutput",
5767:     "Phase8ExecutionPlanOutput",
```

```
5768:     "phase8_execution_plan_assembly",
5769:     "PHASE8_VERSION",
5770:     "REQUIRED_TASK_COUNT",
5771: ]
5772:
5773:
```