

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 17
3: =====
4: Generated: 2025-12-07T06:17:23.668686
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: src/farfan_pipeline/core/phases/phase_orchestrator.py
11: =====
12:
13: """
14: Phase Orchestrator - Constitutional Sequence Enforcement
15: =====
16:
17: This module implements the PhaseOrchestrator which GUARANTEES that:
18:
19: 1. Phases execute in STRICT sequence (0 à\206\222 1 à\206\222 Adapter à\206\222 2)
20: 2. Each phase's output becomes the NEXT phase's input
21: 3. NO phase can be bypassed
22: 4. ALL contracts are validated at boundaries
23: 5. ALL invariants are checked
24: 6. FULL traceability in manifest
25:
26: The orchestrator is the SINGLE point of entry for pipeline execution.
27: It is IMPOSSIBLE to run phases out of order or skip validation.
28:
29: Design Principles:
30: -----
31: - **Single Entry Point**: Only `run_pipeline()` executes the full sequence
32: - **No Bypass**: Phases cannot be called directly from outside
33: - **Contract Enforcement**: All inputs/outputs validated
34: - **Deterministic**: Same Phase0Input à\206\222 same outputs
35: - **Auditable**: Full manifest with all phase boundaries
36:
37: Phase Sequence (IMMUTABLE):
38: -----
39: Phase 0: input_validation
40:     Input: Phase0Input (pdf_path, run_id, questionnaire_path)
41:     Output: CanonicalInput
42:     à\206\223
43: Phase 1: spc_ingestion
44:     Input: CanonicalInput
45:     Output: CanonPolicyPackage
46:     à\206\223
47: Adapter: phase1_to_phase2
48:     Input: CanonPolicyPackage
49:     Output: PreprocessedDocument
50:     à\206\223
51: Phase 2: microquestions
52:     Input: PreprocessedDocument
53:     Output: Phase2Result
54:
55: Author: F.A.R.F.A.N Architecture Team
56: Date: 2025-01-19
```

```
57: """
58:
59: from __future__ import annotations
60:
61: import logging
62: from dataclasses import dataclass, field
63: from pathlib import Path
64: from typing import Any
65:
66: from farfan_pipeline.core.orchestrator.factory import build_processor
67: from farfan_pipeline.core.phases.phase_protocol import (
68:     ContractValidationResult,
69:     PhaseManifestBuilder,
70:     PhaseMetadata,
71: )
72: from farfan_pipeline.core.phases.phase0_input_validation import (
73:     CanonicalInput,
74:     Phase0Input,
75:     Phase0ValidationContract,
76: )
77: from farfan_pipeline.core.phases.phase1_spc_ingestion import (
78:     Phase1SPCIngestionContract,
79: )
80: from farfan_pipeline.core.phases.phase2_types import validate_phase2_result
81: from farfan_pipeline.core.phases.phase3_chunk_routing import (
82:     Phase3ChunkRoutingContract,
83:     Phase3Input,
84: )
85:
86: logger = logging.getLogger(__name__)
87:
88:
89: @dataclass
90: class PipelineResult:
91:     """
92:         Complete result of pipeline execution.
93:
94:         This is the ONLY output of PhaseOrchestrator.run_pipeline().
95:     """
96:
97:     success: bool
98:     run_id: str
99:
100:    # Phase outputs (populated if phase succeeded)
101:    canonical_input: CanonicalInput | None = None
102:    canon_policy_package: Any | None = None # CanonPolicyPackage
103:    preprocessed_document: Any | None = None # PreprocessedDocument
104:    phase2_result: Any | None = None # Phase2Result
105:    phase3_result: Any | None = None # Phase3Result
106:
107:    # Execution metadata
108:    phases_completed: int = 0
109:    phases_failed: int = 0
110:    total_duration_ms: float = 0.0
111:
112:    # Error tracking
```

```
113:     errors: list[str] = field(default_factory=list)
114:
115:     # Manifest
116:     manifest: dict[str, Any] = field(default_factory=dict)
117:
118:
119: class PhaseOrchestrator:
120:     """
121:         Orchestrator that enforces the canonical phase sequence.
122:
123:         This class is the CONSTITUTIONAL GUARANTEE that phases execute
124:         in order with full contract validation.
125:
126:     Usage:
127:     -----
128:     ```python
129:     orchestrator = PhaseOrchestrator()
130:     result = await orchestrator.run_pipeline(
131:         pdf_path=Path("plan.pdf"),
132:         run_id="plan1",
133:         questionnaire_path=Path("questionnaire.json"),
134:         artifacts_dir=Path("artifacts/plan1"),
135:     )
136:
137:     if result.success:
138:         print(f"Pipeline succeeded: {result.phases_completed} phases")
139:     else:
140:         print(f"Pipeline failed: {result.errors}")
141:     ```
142:     """
143:
144:     def __init__(self):
145:         """Initialize orchestrator with phase contracts."""
146:         logger.info("Initializing PhaseOrchestrator with constitutional constraints")
147:
148:         # Initialize phase contracts
149:         self.phase0 = Phase0ValidationContract()
150:         self.phase1 = Phase1SPCIgestionContract()
151:
152:         # Import and initialize adapter contract
153:         from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract
154:         self.adapter = AdapterContract()
155:
156:         # Initialize Phase 3 contract
157:         self.phase3 = Phase3ChunkRoutingContract()
158:
159:         # self.phase2 = Phase2Contract()      # To be implemented
160:
161:         # Initialize manifest builder
162:         self.manifest_builder = PhaseManifestBuilder()
163:
164:         logger.info("PhaseOrchestrator initialized successfully")
165:
166:     async def run_pipeline(
167:         self,
168:         pdf_path: Path,
```

```
169:         run_id: str,
170:         questionnaire_path: Path | None = None,
171:         artifacts_dir: Path | None = None,
172:     ) -> PipelineResult:
173:     """
174:         Execute the COMPLETE canonical pipeline in STRICT sequence.
175:
176:         This is the ONLY way to run the pipeline. It enforces:
177:         1. Phase 0 \206\222 Phase 1 \206\222 Adapter \206\222 Phase 2
178:         2. Contract validation at ALL boundaries
179:         3. Invariant checking for ALL phases
180:         4. Full manifest generation
181:
182:     Args:
183:         pdf_path: Path to input PDF
184:         run_id: Unique run identifier
185:         questionnaire_path: Optional questionnaire path
186:         artifacts_dir: Optional directory for artifacts
187:
188:     Returns:
189:         PipelineResult with success status and all phase outputs
190:
191:     Raises:
192:         This method does NOT raise exceptions. All errors are captured
193:         in PipelineResult.errors and PipelineResult.success = False.
194:     """
195:     logger.info(f"Starting pipeline execution: run_id={run_id}")
196:
197:     result = PipelineResult(
198:         success=False, # Will be set to True only if ALL phases succeed
199:         run_id=run_id,
200:     )
201:
202:     # Create artifacts directory if provided
203:     if artifacts_dir:
204:         artifacts_dir.mkdir(parents=True, exist_ok=True)
205:
206:     try:
207:         # =====
208:         # PHASE 0: Input Validation
209:         # =====
210:         logger.info("=" * 70)
211:         logger.info("PHASE 0: Input Validation")
212:         logger.info("=" * 70)
213:
214:         phase0_input = Phase0Input(
215:             pdf_path=pdf_path,
216:             run_id=run_id,
217:             questionnaire_path=questionnaire_path,
218:         )
219:
220:         canonical_input, phase0_metadata = await self.phase0.run(phase0_input)
221:
222:         # Record Phase 0 in manifest
223:         self.manifest_builder.record_phase(
224:             phase_name="phase0_input_validation",
```

```
225:         metadata=phase0_metadata,
226:         input_validation=self.phase0.validate_input(phase0_input),
227:         output_validation=self.phase0.validate_output(canonical_input),
228:         invariants_checked=[inv.name for inv in self.phase0.invariants],
229:         artifacts=[], # No artifacts for Phase 0
230:     )
231:
232:     result.canonical_input = canonical_input
233:     result.phases_completed += 1
234:     result.total_duration_ms += phase0_metadata.duration_ms or 0.0
235:
236:     logger.info(
237:         f"Phase 0 completed successfully in {phase0_metadata.duration_ms:.0f}ms"
238:     )
239:
240:     # =====
241:     # PHASE 1: SPC Ingestion
242:     # =====
243:     logger.info("=" * 70)
244:     logger.info("PHASE 1: SPC Ingestion (15 subfases)")
245:     logger.info("=" * 70)
246:
247:     # Phase 1 input is Phase 0 output (guaranteed by type system)
248:     cpp, phase1_metadata = await self.phase1.run(canonical_input)
249:
250:     # Record Phase 1 in manifest
251:     self.manifest_builder.record_phase(
252:         phase_name="phase1_spcl_ingestion",
253:         metadata=phase1_metadata,
254:         input_validation=self.phase1.validate_input(canonical_input),
255:         output_validation=self.phase1.validate_output(cpp),
256:         invariants_checked=[inv.name for inv in self.phase1.invariants],
257:         artifacts=[], # Artifacts tracked separately
258:     )
259:
260:     result.cannon_policy_package = cpp
261:     result.phases_completed += 1
262:     result.total_duration_ms += phase1_metadata.duration_ms or 0.0
263:
264:     logger.info(
265:         f"Phase 1 completed successfully in {phase1_metadata.duration_ms:.0f}ms"
266:     )
267:     logger.info(f"Generated {len(cpp.chunk_graph.chunks)} chunks")
268:
269:     # =====
270:     # ADAPTER: Phase 1 \206\222 Phase 2
271:     # =====
272:     logger.info("=" * 70)
273:     logger.info("ADAPTER: CanonPolicyPackage \206\222 PreprocessedDocument")
274:     logger.info("=" * 70)
275:
276:     # Run adapter with contract enforcement
277:     preprocessed, adapter_metadata = await self.adapter.run(cpp)
278:
279:     # Record Adapter in manifest
280:     self.manifest_builder.record_phase(
```

```
281:         phase_name="phase1_to_phase2_adapter",
282:         metadata=adapter_metadata,
283:         input_validation=self.adapter.validate_input(cpp),
284:         output_validation=self.adapter.validate_output(preprocessed),
285:         invariants_checked=[inv.name for inv in self.adapter.invariants],
286:         artifacts=[],
287:     )
288:
289:     result.preprocessed_document = preprocessed
290:     result.phases_completed += 1
291:     result.total_duration_ms += adapter_metadata.duration_ms or 0.0
292:
293:     logger.info(
294:         f"Adapter completed successfully in {adapter_metadata.duration_ms:.0f}ms"
295:     )
296:     logger.info(
297:         f"PreprocessedDocument: {len(preprocessed.sentences)} sentences, "
298:         f"mode={preprocessed.processing_mode}"
299:     )
300:
301: # =====
302: # CORE ORCHESTRATOR: Phases 0-10 (Includes Micro-Questions)
303: # =====
304: logger.info("=" * 70)
305: logger.info("CORE ORCHESTRATOR: Executing Phases 0-10")
306: logger.info("=" * 70)
307:
308: # --- Imports for Phase 2 Integration ---
309: from datetime import datetime, timedelta, timezone
310:
311: # --- Execute Core Orchestrator ---
312: processor = build_processor()
313: p2_block_started_at = datetime.now(timezone.utc)
314: core_results = await processor.orchestrator.process_development_plan_async(
315:     pdf_path=str(pdf_path),
316:     preprocessed_document=preprocessed,
317: )
318: p2_block_finished_at = datetime.now(timezone.utc)
319:
320: # --- Process and Record Phase 2 ---
321: phase2_success = False
322: phase2_errors: list[str] = []
323: phase2_questions: list[dict[str, Any]] | None = None
324:
325: if len(core_results) >= 3:
326:     phase2_core = core_results[2] # FASE 2 - Micro Preguntas
327:     result.phase2_result = phase2_core.data if phase2_core.success else None
328:
329:     if phase2_core.success:
330:         is_valid, validation_errors, normalized_questions = validate_phase2_result(
331:             phase2_core.data
332:         )
333:         phase2_questions = normalized_questions
334:         if not is_valid:
335:             phase2_errors.extend(validation_errors)
336:             phase2_errors.append(
```

```
337:                     "Phase 2 failed structural invariant: questions list is empty or missing."
338:                 )
339:             phase2_success = phase2_core.success and is_valid
340:         else:
341:             phase2_errors.append(
342:                 f"Core phase 2 returned error: {phase2_core.error}"
343:             )
344:
345:             # --- Create Manifest Entry for Phase 2 ---
346:             p2_error_msg = "; ".join(phase2_errors) if phase2_errors else None
347:
348:             # Approximate start/end times for the manifest metadata
349:             p2_duration = timedelta(milliseconds=phase2_core.duration_ms)
350:             p2_started_at_approx = p2_block_finished_at - p2_duration
351:
352:             p2_metadata = PhaseMetadata(
353:                 phase_name="phase2_microquestions",
354:                 success=phase2_success,
355:                 error=p2_error_msg,
356:                 duration_ms=phase2_core.duration_ms,
357:                 started_at=p2_started_at_approx.isoformat(),
358:                 finished_at=p2_block_finished_at.isoformat(),
359:             )
360:
361:             # Create validation results to satisfy the manifest builder
362:             dummy_input_validation = ContractValidationResult(
363:                 passed=True,
364:                 contract_type="input",
365:                 phase_name="phase2_microquestions",
366:             )
367:             dummy_output_validation = ContractValidationResult(
368:                 passed=phase2_success,
369:                 contract_type="output",
370:                 phase_name="phase2_microquestions",
371:                 errors=phase2_errors,
372:             )
373:
374:             self.manifest_builder.record_phase(
375:                 phase_name="phase2_microquestions",
376:                 metadata=p2_metadata,
377:                 input_validation=dummy_input_validation,
378:                 output_validation=dummy_output_validation,
379:                 invariants_checked=["questions_are_present_and_non_empty"],
380:                 artifacts=[],
381:             )
382:             self.manifest_builder.phases["phase2_microquestions"]["question_count"] = len(phase2_questions or [])
383:             if phase2_errors:
384:                 self.manifest_builder.phases["phase2_microquestions"]["errors"] = list(phase2_errors)
385:
386:             if not phase2_success:
387:                 error_msg = f"Core Orchestrator Phase 2 failed: {p2_error_msg}"
388:                 logger.error(error_msg)
389:                 result.errors.append(error_msg)
390:                 result.phases_failed += 1
391:
392:             else:
393:                 # Only add core result count if Phase 2 was successful
```

```
393:         result.phase2_result = {"questions": phase2_questions or []}
394:         result.phases_completed += len(core_results)
395:         logger.info(
396:             f"Core Orchestrator completed {len(core_results)} phases successfully"
397:         )
398:
399:         # =====
400:         # PHASE 3: Chunk Routing
401:         # =====
402:         logger.info("=" * 70)
403:         logger.info("PHASE 3: Chunk Routing")
404:         logger.info("=" * 70)
405:
406:         # Phase 3 input is preprocessed document + Phase 2 questions
407:         phase3_input = Phase3Input(
408:             preprocessed_document=preprocessed,
409:             questions=phase2_questions or []
410:         )
411:
412:         phase3_result, phase3_metadata = await self.phase3.run(phase3_input)
413:
414:         # Record Phase 3 in manifest
415:         self.manifest_builder.record_phase(
416:             phase_name="phase3_chunk_routing",
417:             metadata=phase3_metadata,
418:             input_validation=self.phase3.validate_input(phase3_input),
419:             output_validation=self.phase3.validate_output(phase3_result),
420:             invariants_checked=[inv.name for inv in self.phase3.invariants],
421:             artifacts=[]
422:         )
423:
424:         result.phase3_result = phase3_result
425:         result.phases_completed += 1
426:         result.total_duration_ms += phase3_metadata.duration_ms or 0.0
427:
428:         logger.info(
429:             f"Phase 3 completed successfully in {phase3_metadata.duration_ms:.0f}ms"
430:         )
431:         logger.info(
432:             f"Routed {phase3_result.successful_routes}/{phase3_result.total_questions} questions"
433:         )
434:
435:     else:
436:         # Phase 2 was not even present in the results
437:         missing_p2_error = "Core Orchestrator did not produce a result for Phase 2."
438:         logger.error(missing_p2_error)
439:         result.errors.append(missing_p2_error)
440:         result.phases_failed += 1
441:         # Create a failure record in the manifest
442:         p2_metadata = PhaseMetadata(
443:             phase_name="phase2_microquestions",
444:             success=False,
445:             error=missing_p2_error,
446:             started_at=p2_block_started_at.isoformat(),
447:             finished_at=p2_block_finished_at.isoformat(),
448:             duration_ms=(p2_block_finished_at-p2_block_started_at).total_seconds() * 1000,
```

```
449:         )
450:         self.manifest_builder.record_phase(
451:             phase_name="phase2_microquestions",
452:             metadata=p2_metadata,
453:             input_validation=ContractValidationResult(passed=False, contract_type="input", phase_name="phase2_microquestions", errors=[missing_p2_error]),
454:             output_validation=ContractValidationResult(passed=False, contract_type="output", phase_name="phase2_microquestions", errors=[missing_p2_error]),
455:             invariants_checked=[],
456:             artifacts=[],
457:         )
458:         self.manifest_builder.phases["phase2_microquestions"]["question_count"] = 0
459:         self.manifest_builder.phases["phase2_microquestions"]["errors"] = [missing_p2_error]
460:
461:
462: # =====
463: # PIPELINE SUCCESS
464: # =====
465: # Success is now conditional on all canonical phases, including Phase 2
466: all_phases_ok = all(
467:     p.get("status") == "success"
468:     for p in self.manifest_builder.phases.values()
469: )
470:
471: if all_phases_ok:
472:     result.success = True
473:     logger.info("=" * 70)
474:     logger.info(f"PIPELINE COMPLETED SUCCESSFULLY")
475:     logger.info(f"Phases completed: {result.phases_completed}")
476:     logger.info(f"Total duration: {result.total_duration_ms:.0f}ms")
477:     logger.info("=" * 70)
478: else:
479:     # Ensure result.success is False if we got here with a failure
480:     result.success = False
481:     final_error = f"Pipeline failed. Check manifest for details. Completed: {result.phases_completed}, Failed: {result.phases_failed}"
482:     if not result.errors:
483:         result.errors.append(final_error)
484:     logger.error(final_error)
485:
486: except Exception as e:
487:     # Capture error
488:     error_msg = f"Pipeline failed: {e}"
489:     logger.error(error_msg, exc_info=True)
490:     result.errors.append(error_msg)
491:     result.success = False
492:     result.phases_failed += 1
493:
494: finally:
495:     # Always generate manifest
496:     result.manifest = self.manifest_builder.to_dict()
497:     phase2_entry = result.manifest.get("phases", {}).get("phase2_microquestions")
498:     if phase2_entry is not None:
499:         result.manifest["phases"]["phase2"] = phase2_entry
500:
501:     # Save manifest if artifacts_dir provided
502:     if artifacts_dir:
```

```
503:             manifest_path = artifacts_dir / "phase_manifest.json"
504:             self.manifest_builder.save(manifest_path)
505:             logger.info(f"Phase manifest saved to {manifest_path}")
506:
507:         return result
508:
509:
510:     __all__ = [
511:         "PhaseOrchestrator",
512:         "PipelineResult",
513:     ]
514:
515:
516:
517: =====
518: FILE: src/farfan_pipeline/core/phases/phase_protocol.py
519: =====
520:
521: """
522: Phase Contract Protocol - Constitutional Constraint System
523: =====
524:
525: This module implements the constitutional constraint framework where each phase:
526:
527: 1. Has an EXPLICIT input contract (typed, validated)
528: 2. Has an EXPLICIT output contract (typed, validated)
529: 3. Communicates ONLY through these contracts (no side channels)
530: 4. Is enforced by validators (runtime contract checking)
531: 5. Is tracked in the verification manifest (full traceability)
532:
533: Design Principles:
534: -----
535: - **Single Entry Point**: Each phase accepts exactly ONE input type
536: - **Single Exit Point**: Each phase produces exactly ONE output type
537: - **No Bypass**: The orchestrator enforces sequential execution
538: - **Verifiable**: All contracts are validated and logged
539: - **Deterministic**: Same input → same output (modulo controlled randomness)
540:
541: Phase Structure:
542: -----
543: phase0_input_validation:
544:     Input: Phase0Input (raw PDF path + run_id)
545:     Output: CanonicalInput (validated, hashed, ready)
546:
547: phasel_spc_ingestion:
548:     Input: CanonicalInput
549:     Output: CanonPolicyPackage (60 chunks, PAGEDIM structured)
550:
551: phasel_to_phase2_adapter:
552:     Input: CanonPolicyPackage
553:     Output: PreprocessedDocument (chunked mode)
554:
555: phase2_microquestions:
556:     Input: PreprocessedDocument
557:     Output: Phase2Result (305 questions answered)
558:
```

```
559: Author: F.A.R.F.A.N Architecture Team
560: Date: 2025-01-19
561: """
562:
563: from __future__ import annotations
564:
565: import hashlib
566: import json
567: from abc import ABC, abstractmethod
568: from dataclasses import asdict, dataclass, field
569: from datetime import datetime, timezone
570: from pathlib import Path
571: from typing import Any, Generic, TypeVar
572:
573: from pydantic import BaseModel, Field, ValidationError
574:
575: # Type variables for generic phase contracts
576: TInput = TypeVar("TInput")
577: TOutput = TypeVar("TOutput")
578:
579:
580: @dataclass
581: class PhaseInvariant:
582:     """An invariant that must hold for a phase."""
583:
584:     name: str
585:     description: str
586:     check: callable # Function that returns bool
587:     error_message: str
588:
589:
590: @dataclass
591: class PhaseMetadata:
592:     """Metadata for a phase execution."""
593:
594:     phase_name: str
595:     started_at: str
596:     finished_at: str | None = None
597:     duration_ms: float | None = None
598:     success: bool = False
599:     error: str | None = None
600:
601:
602: @dataclass
603: class ContractValidationResult:
604:     """Result of validating a contract."""
605:
606:     passed: bool
607:     contract_type: str # "input" or "output"
608:     phase_name: str
609:     errors: list[str] = field(default_factory=list)
610:     warnings: list[str] = field(default_factory=list)
611:     validation_timestamp: str = field(
612:         default_factory=lambda: datetime.now(timezone.utc).isoformat()
613:     )
614:
```

```
615:
616: class PhaseContract(ABC, Generic[TInput, TOutput]):
617:     """
618:         Abstract base class for phase contracts.
619:
620:         Each phase must implement:
621:             1. Input contract validation
622:             2. Output contract validation
623:             3. Invariant checking
624:             4. Phase execution logic
625:
626:         This enforces the constitutional constraint that phases communicate
627:         ONLY through validated contracts.
628:     """
629:
630:     def __init__(self, phase_name: str):
631:         """
632:             Initialize phase contract.
633:
634:             Args:
635:                 phase_name: Canonical name of the phase (e.g., "phase0_input_validation")
636:             """
637:         self.phase_name = phase_name
638:         self.invariants: list[PhaseInvariant] = []
639:         self.metadata: PhaseMetadata | None = None
640:
641:     @abstractmethod
642:     def validate_input(self, input_data: Any) -> ContractValidationResult:
643:         """
644:             Validate input contract.
645:
646:             Args:
647:                 input_data: Input to validate
648:
649:             Returns:
650:                 ContractValidationResult with validation status
651:             """
652:         pass
653:
654:     @abstractmethod
655:     def validate_output(self, output_data: Any) -> ContractValidationResult:
656:         """
657:             Validate output contract.
658:
659:             Args:
660:                 output_data: Output to validate
661:
662:             Returns:
663:                 ContractValidationResult with validation status
664:             """
665:         pass
666:
667:     @abstractmethod
668:     async def execute(self, input_data: TInput) -> TOutput:
669:         """
670:             Execute the phase logic.
```

```
671:
672:     Args:
673:         input_data: Validated input conforming to input contract
674:
675:     Returns:
676:         Output conforming to output contract
677:
678:     Raises:
679:         ValueError: If input contract validation fails
680:         RuntimeError: If phase execution fails
681:     """
682:     pass
683:
684: def addInvariant(
685:     self,
686:     name: str,
687:     description: str,
688:     check: callable,
689:     error_message: str,
690: ) -> None:
691:     """
692:     Add an invariant to this phase.
693:
694:     Args:
695:         name: Invariant name
696:         description: Human-readable description
697:         check: Function that returns bool (True = invariant holds)
698:         error_message: Error message if invariant fails
699:     """
700:     self.invariants.append(
701:         PhaseInvariant(
702:             name=name,
703:             description=description,
704:             check=check,
705:             error_message=error_message,
706:         )
707:     )
708:
709: def checkInvariants(self, data: Any) -> tuple[bool, list[str]]:
710:     """
711:     Check all invariants for this phase.
712:
713:     Args:
714:         data: Data to check invariants against
715:
716:     Returns:
717:         Tuple of (all_passed, failed_invariant_messages)
718:     """
719:     failed_messages = []
720:     for inv in self.invariants:
721:         try:
722:             if not inv.check(data):
723:                 failed_messages.append(f"{inv.name}: {inv.error_message}")
724:         except Exception as e:
725:             failed_messages.append(f"{inv.name}: Exception during check: {e}")
726:
```

```
727:         return len(failed_messages) == 0, failed_messages
728:
729:     async def run(self, input_data: TInput) -> tuple[TOutput, PhaseMetadata]:
730:         """
731:             Run the complete phase with validation and invariant checking.
732:
733:             This is the ONLY way to execute a phase - it enforces:
734:             1. Input validation
735:             2. Invariant checking (pre-execution if applicable)
736:             3. Phase execution
737:             4. Output validation
738:             5. Invariant checking (post-execution)
739:             6. Metadata recording
740:
741:         Args:
742:             input_data: Input to the phase
743:
744:         Returns:
745:             Tuple of (output_data, phase_metadata)
746:
747:         Raises:
748:             ValueError: If contract validation fails
749:             RuntimeError: If invariants fail or execution fails
750:
751:         """
752:         started_at = datetime.now(timezone.utc)
753:         metadata = PhaseMetadata(
754:             phase_name=self.phase_name,
755:             started_at=started_at.isoformat(),
756:         )
757:
758:         try:
759:             # 1. Validate input contract
760:             input_validation = self.validate_input(input_data)
761:             if not input_validation.passed:
762:                 error_msg = f"Input contract validation failed: {input_validation.errors}"
763:                 metadata.error = error_msg
764:                 metadata.success = False
765:                 raise ValueError(error_msg)
766:
767:             # 2. Execute phase
768:             output_data = await self.execute(input_data)
769:
770:             # 3. Validate output contract
771:             output_validation = self.validate_output(output_data)
772:             if not output_validation.passed:
773:                 error_msg = f"Output contract validation failed: {output_validation.errors}"
774:                 metadata.error = error_msg
775:                 metadata.success = False
776:                 raise ValueError(error_msg)
777:
778:             # 4. Check invariants
779:             invariants_passed, failed_invariants = self.check_invariants(output_data)
780:             if not invariants_passed:
781:                 error_msg = f"Phase invariants failed: {failed_invariants}"
782:                 metadata.error = error_msg
783:                 metadata.success = False
```

```
783:             raise RuntimeError(error_msg)
784:
785:         # Success
786:         metadata.success = True
787:         return output_data, metadata
788:
789:     except Exception as e:
790:         metadata.error = str(e)
791:         metadata.success = False
792:         raise
793:
794:     finally:
795:         finished_at = datetime.now(timezone.utc)
796:         metadata.finished_at = finished_at.isoformat()
797:         metadata.duration_ms = (
798:             finished_at - started_at
799:         ).total_seconds() * 1000
800:         self.metadata = metadata
801:
802:
803: @dataclass
804: class PhaseArtifact:
805:     """An artifact produced by a phase."""
806:
807:     artifact_name: str
808:     artifact_path: Path
809:     sha256: str
810:     size_bytes: int
811:     created_at: str
812:
813:
814: class PhaseManifestBuilder:
815:     """
816:         Builds the phase-explicit section of the verification manifest.
817:
818:         Each phase execution is recorded with:
819:             - Input/output contract hashes
820:             - Invariants checked
821:             - Artifacts produced
822:             - Timing information
823:     """
824:
825:     def __init__(self):
826:         """Initialize manifest builder."""
827:         self.phases: dict[str, dict[str, Any]] = {}
828:
829:     def record_phase(
830:         self,
831:         phase_name: str,
832:         metadata: PhaseMetadata,
833:         input_validation: ContractValidationResult,
834:         output_validation: ContractValidationResult,
835:         invariants_checked: list[str],
836:         artifacts: list[PhaseArtifact],
837:     ) -> None:
838:         """
```

```
839:     Record a phase execution in the manifest.
840:
841:     Args:
842:         phase_name: Name of the phase
843:         metadata: Phase execution metadata
844:         input_validation: Input contract validation result
845:         output_validation: Output contract validation result
846:         invariants_checked: List of invariant names that were checked
847:         artifacts: List of artifacts produced by this phase
848:
849:     """
850:     self.phases[phase_name] = {
851:         "status": "success" if metadata.success else "failed",
852:         "started_at": metadata.started_at,
853:         "finished_at": metadata.finished_at,
854:         "duration_ms": metadata.duration_ms,
855:         "input_contract": {
856:             "validation_passed": input_validation.passed,
857:             "errors": input_validation.errors,
858:             "warnings": input_validation.warnings,
859:         },
860:         "output_contract": {
861:             "validation_passed": output_validation.passed,
862:             "errors": output_validation.errors,
863:             "warnings": output_validation.warnings,
864:         },
865:         "invariants_checked": invariants_checked,
866:         "invariants_satisfied": metadata.success,
867:         "artifacts": [
868:             {
869:                 "name": a.artifact_name,
870:                 "path": str(a.artifact_path),
871:                 "sha256": a.sha256,
872:                 "size_bytes": a.size_bytes,
873:             }
874:             for a in artifacts
875:         ],
876:         "error": metadata.error,
877:     }
878:
879:     def to_dict(self) -> dict[str, Any]:
880:
881:         """
882:             Convert manifest to dictionary.
883:
884:             Returns:
885:                 Dictionary representation of the phase manifest
886:
887:         """
888:         return {
889:             "phases": self.phases,
890:             "total_phases": len(self.phases),
891:             "successful_phases": sum(
892:                 1 for p in self.phases.values() if p["status"] == "success"
893:             ),
894:             "failed_phases": sum(
895:                 1 for p in self.phases.values() if p["status"] == "failed"
896:             ),
897:         }
```

```
895:
896:     def save(self, output_path: Path) -> None:
897:         """
898:             Save manifest to JSON file.
899:
900:             Args:
901:                 output_path: Path to save manifest
902:             """
903:             with open(output_path, "w") as f:
904:                 json.dump(self.to_dict(), f, indent=2)
905:
906:
907: def compute_contract_hash(contract_data: Any) -> str:
908:     """
909:         Compute SHA256 hash of a contract's data.
910:
911:         Args:
912:             contract_data: Contract data (dict, dataclass, or Pydantic model)
913:
914:         Returns:
915:             Hex-encoded SHA256 hash
916:             """
917:             # Convert to dict if needed
918:             if hasattr(contract_data, "dict"):
919:                 # Pydantic model
920:                 data_dict = contract_data.dict()
921:             elif hasattr(contract_data, "__dataclass_fields__"):
922:                 # Dataclass
923:                 data_dict = astype(contract_data)
924:             elif isinstance(contract_data, dict):
925:                 data_dict = contract_data
926:             else:
927:                 raise TypeError(f"Cannot hash contract data of type {type(contract_data)}")
928:
929:             # Serialize to JSON with sorted keys for determinism
930:             json_str = json.dumps(data_dict, sort_keys=True, separators=(", ", ":"), separators=(", ", ":")))
931:             return hashlib.sha256(json_str.encode("utf-8")).hexdigest()
932:
933:
934: __all__ = [
935:     "PhaseContract",
936:     "PhaseInvariant",
937:     "PhaseMetadata",
938:     "ContractValidationResult",
939:     "PhaseArtifact",
940:     "PhaseManifestBuilder",
941:     "compute_contract_hash",
942: ]
943:
944:
945:
946: =====
947: FILE: src/farfan_pipeline/core/ports.py
948: =====
949:
950: """
```

```
951: Port interfaces for dependency injection.
952:
953: Ports define abstract interfaces for external interactions (I/O, time, environment).
954: These are implemented by adapters in the infrastructure layer.
955:
956: This follows the Ports and Adapters (Hexagonal) architecture pattern:
957: - Ports are in the core layer (no dependencies)
958: - Adapters are in the infrastructure layer (can import anything)
959: - Core modules depend on ports (abstractions), not adapters (implementations)
960:
961: Version: 1.0.0
962: """
963:
964: from datetime import datetime
965: from typing import Any, Protocol
966:
967: from farfan_pipeline.core.analysis_port import RecommendationEnginePort
968:
969:
970: class FilePort(Protocol):
971:     """Port for file system operations.
972:
973:     Implementations provide access to file reading and writing.
974:     Core modules receive a FilePort instance via dependency injection.
975:     """
976:
977:     def read_text(self, path: str, encoding: str = "utf-8") -> str:
978:         """Read text from a file.
979:
980:         Args:
981:             path: File path to read
982:             encoding: Text encoding (default: utf-8)
983:
984:         Returns:
985:             File contents as string
986:
987:         Raises:
988:             FileNotFoundError: If file does not exist
989:             PermissionError: If file cannot be read
990:         """
991:
992:         ...
993:
994:     def write_text(self, path: str, content: str, encoding: str = "utf-8") -> None:
995:         """Write text to a file.
996:
997:         Args:
998:             path: File path to write
999:             content: Text content to write
1000:            encoding: Text encoding (default: utf-8)
1001:
1002:         Raises:
1003:             PermissionError: If file cannot be written
1004:         """
1005:
1006:     def read_bytes(self, path: str) -> bytes:
```

```
1007:     """Read bytes from a file.
1008:
1009:     Args:
1010:         path: File path to read
1011:
1012:     Returns:
1013:         File contents as bytes
1014:
1015:     Raises:
1016:         FileNotFoundError: If file does not exist
1017:         PermissionError: If file cannot be read
1018:     """
1019:
1020:
1021: def write_bytes(self, path: str, content: bytes) -> None:
1022:     """Write bytes to a file.
1023:
1024:     Args:
1025:         path: File path to write
1026:         content: Bytes content to write
1027:
1028:     Raises:
1029:         PermissionError: If file cannot be written
1030:     """
1031:
1032:
1033: def exists(self, path: str) -> bool:
1034:     """Check if a file or directory exists.
1035:
1036:     Args:
1037:         path: Path to check
1038:
1039:     Returns:
1040:         True if path exists, False otherwise
1041:     """
1042:
1043:
1044: def mkdir(self, path: str, parents: bool = False, exist_ok: bool = False) -> None:
1045:     """Create a directory.
1046:
1047:     Args:
1048:         path: Directory path to create
1049:         parents: Create parent directories if needed
1050:         exist_ok: Don't raise error if directory exists
1051:
1052:     Raises:
1053:         FileExistsError: If directory exists and exist_ok is False
1054:     """
1055:
1056:
1057:
1058: class JsonPort(Protocol):
1059:     """Port for JSON serialization/deserialization.
1060:
1061:     Separates JSON operations from file I/O for better composability.
1062:     """
1063:
```

```
1063:  
1064:     def loads(self, text: str) -> Any:  
1065:         """Parse JSON from string.  
1066:  
1067:             Args:  
1068:                 text: JSON string  
1069:  
1070:             Returns:  
1071:                 Parsed Python object  
1072:  
1073:             Raises:  
1074:                 ValueError: If JSON is invalid  
1075:             """  
1076:             ...  
1077:  
1078:     def dumps(self, obj: Any, indent: int | None = None) -> str:  
1079:         """Serialize object to JSON string.  
1080:  
1081:             Args:  
1082:                 obj: Python object to serialize  
1083:                 indent: Indentation spaces (None for compact)  
1084:  
1085:             Returns:  
1086:                 JSON string  
1087:  
1088:             Raises:  
1089:                 TypeError: If object is not serializable  
1090:             """  
1091:             ...  
1092:  
1093:  
1094: class EnvPort(Protocol):  
1095:     """Port for environment variable access.  
1096:  
1097:     Allows core modules to access configuration without direct os.environ coupling.  
1098:     """  
1099:  
1100:     def get(self, key: str, default: str | None = None) -> str | None:  
1101:         """Get environment variable.  
1102:  
1103:             Args:  
1104:                 key: Environment variable name  
1105:                 default: Default value if not set  
1106:  
1107:             Returns:  
1108:                 Environment variable value or default  
1109:             """  
1110:             ...  
1111:  
1112:     def get_required(self, key: str) -> str:  
1113:         """Get required environment variable.  
1114:  
1115:             Args:  
1116:                 key: Environment variable name  
1117:  
1118:             Returns:
```

```
1119:             Environment variable value
1120:
1121:             Raises:
1122:                 ValueError: If environment variable is not set
1123:                 """
1124:                 ...
1125:
1126:             def get_bool(self, key: str, default: bool = False) -> bool:
1127:                 """Get environment variable as boolean.
1128:
1129:                 Args:
1130:                     key: Environment variable name
1131:                     default: Default value if not set
1132:
1133:                 Returns:
1134:                     Boolean value (true/false/yes/no/1/0)
1135:                     """
1136:                     ...
1137:
1138:
1139:             class ClockPort(Protocol):
1140:                 """Port for time operations.
1141:
1142:                     Allows core modules to get current time without direct datetime.now() calls.
1143:                     Enables time manipulation in tests.
1144:                     """
1145:
1146:                 def now(self) -> datetime:
1147:                     """Get current datetime.
1148:
1149:                     Returns:
1150:                         Current datetime
1151:                         """
1152:                         ...
1153:
1154:                 def utcnow(self) -> datetime:
1155:                     """Get current UTC datetime.
1156:
1157:                     Returns:
1158:                         Current UTC datetime
1159:                         """
1160:                         ...
1161:
1162:
1163:             class LogPort(Protocol):
1164:                 """Port for logging operations.
1165:
1166:                     Allows core modules to log without coupling to specific logging framework.
1167:                     """
1168:
1169:                 def debug(self, message: str, **kwargs: Any) -> None:
1170:                     """Log debug message."""
1171:                     ...
1172:
1173:                 def info(self, message: str, **kwargs: Any) -> None:
1174:                     """Log info message."""
```

```
1175:     ...
1176:
1177:     def warning(self, message: str, **kwargs: Any) -> None:
1178:         """Log warning message."""
1179:         ...
1180:
1181:     def error(self, message: str, **kwargs: Any) -> None:
1182:         """Log error message."""
1183:         ...
1184:
1185:
1186: class PortCPPIngest(Protocol):
1187:     """Port for CPP (Canon Policy Package) ingestion.
1188:
1189:     Ingests documents and produces Canon Policy Packages with complete provenance.
1190:     """
1191:
1192:     def ingest(self, input_uri: str) -> Any:
1193:         """Ingest document from URI and produce Canon Policy Package.
1194:
1195:         Args:
1196:             input_uri: URI to document (file://, http://, etc.)
1197:
1198:         Returns:
1199:             CanonPolicyPackage with complete chunk graph and metadata
1200:
1201:         Requires:
1202:             - Valid input URI
1203:             - Accessible document at URI
1204:
1205:         Ensures:
1206:             - chunk_graph is not None
1207:             - policy_manifest is not None
1208:             - provenance_completeness == 1.0
1209:         """
1210:         ...
1211:
1212:
1213: class PortCPPAdapter(Protocol):
1214:     """Port for CPP to PreprocessedDocument adaptation.
1215:
1216:     Converts Canon Policy Package to orchestrator's PreprocessedDocument format.
1217:
1218:     Note: CPP is the legacy name. Use PortSPCAAdapter for new code.
1219:     """
1220:
1221:     def to_preprocessed_document(self, cpp: Any, document_id: str) -> Any:
1222:         """Convert CPP to PreprocessedDocument.
1223:
1224:         Args:
1225:             cpp: Canon Policy Package from ingestion
1226:             document_id: Unique document identifier
1227:
1228:         Returns:
1229:             PreprocessedDocument for orchestrator
1230:
```

```
1231:     Requires:
1232:         - cpp with valid chunk_graph
1233:         - cpp.policy_manifest exists
1234:         - document_id is non-empty
1235:
1236:     Ensures:
1237:         - sentence_metadata is not empty
1238:         - resolution_index is consistent
1239:         - provenance_completeness == 1.0
1240:     """
1241: ...
1242:
1243:
1244: class PortSPCAdapter(Protocol):
1245:     """Port for SPC (Smart Policy Chunks) to PreprocessedDocument adaptation.
1246:
1247:     Converts Smart Policy Chunks to orchestrator's PreprocessedDocument format.
1248:     This is the preferred terminology for new code.
1249:     """
1250:
1251:     def to_preprocessed_document(self, spc: Any, document_id: str) -> Any:
1252:         """Convert SPC to PreprocessedDocument.
1253:
1254:         Args:
1255:             spc: Smart Policy Chunks package from ingestion
1256:             document_id: Unique document identifier
1257:
1258:         Returns:
1259:             PreprocessedDocument for orchestrator
1260:
1261:         Requires:
1262:             - spc with valid chunk_graph
1263:             - spc.policy_manifest exists
1264:             - document_id is non-empty
1265:
1266:         Ensures:
1267:             - sentence_metadata is not empty
1268:             - resolution_index is consistent
1269:             - provenance_completeness == 1.0
1270:     """
1271: ...
1272:
1273:
1274: class PortSignalsClient(Protocol):
1275:     """Port for fetching strategic signals.
1276:
1277:     Retrieves policy-aware signals from memory or HTTP sources.
1278:     Semantics: None return = 304 Not Modified or circuit breaker open.
1279:     """
1280:
1281:     def fetch(self, policy_area: str) -> Any | None:
1282:         """Fetch signals for policy area.
1283:
1284:         Args:
1285:             policy_area: Policy domain (fiscal, salud, ambiente, etc.)
1286:
```

```
1287:         Returns:
1288:             SignalPack if available, None if 304/breaker open
1289:
1290:         Requires:
1291:             - policy_area is valid PolicyArea literal
1292:
1293:         Ensures:
1294:             - If not None, returns valid SignalPack with version
1295:             - None is justified (304 or breaker state)
1296:             """
1297:             ...
1298:
1299:
1300: class PortSignalsRegistry(Protocol):
1301:     """Port for signal registry with TTL and LRU.
1302:
1303:     Manages in-memory cache of strategic signals with expiration.
1304:     """
1305:
1306:     def put(self, pack: Any) -> None:
1307:         """Store signal pack in registry.
1308:
1309:         Args:
1310:             pack: SignalPack to store
1311:
1312:         Requires:
1313:             - pack is valid SignalPack
1314:             - pack.version is present
1315:             """
1316:
1317:
1318:     def get(self, policy_area: str) -> dict[str, Any] | None:
1319:         """Retrieve signals for policy area.
1320:
1321:         Args:
1322:             policy_area: Policy domain
1323:
1324:         Returns:
1325:             Signal data if cached and not expired, None otherwise
1326:             """
1327:
1328:
1329:     def fingerprint(self) -> str:
1330:         """Compute registry fingerprint for drift detection.
1331:
1332:         Returns:
1333:             BLAKE3 hash of current registry state
1334:             """
1335:
1336:
1337:
1338: class PortArgRouter(Protocol):
1339:     """Port for argument routing and validation.
1340:
1341:     Routes method calls with strict parameter validation.
1342:     """
```

```
1343:  
1344:     def route(  
1345:         self, class_name: str, method_name: str, payload: dict[str, Any]  
1346:     ) -> tuple[tuple[Any, ...], dict[str, Any]]:  
1347:         """Route method call to (args, kwargs).  
1348:  
1349:             Args:  
1350:                 class_name: Target class name  
1351:                 method_name: Target method name  
1352:                 payload: Input parameters  
1353:  
1354:             Returns:  
1355:                 Tuple of (args, kwargs) for method call  
1356:  
1357:             Requires:  
1358:                 - class_name exists in registry  
1359:                 - method_name exists on class  
1360:                 - method signature is known or has **kwargs  
1361:  
1362:             Ensures:  
1363:                 - No silent parameter drops  
1364:                 - All required args present  
1365:                 - No unexpected kwargs (unless **kwargs in signature)  
1366:             """  
1367:             ...  
1368:  
1369:  
1370: class PortExecutor(Protocol):  
1371:     """Port for executing methods with configuration.  
1372:  
1373:     Executes methods with injected executor config and signals.  
1374:     """  
1375:  
1376:     def run(self, prompt: str, overrides: Any | None = None) -> Any:  
1377:         """Execute with prompt and optional config overrides.  
1378:  
1379:             Args:  
1380:                 prompt: Execution prompt/input  
1381:                 overrides: Optional ExecutorConfig overrides  
1382:  
1383:             Returns:  
1384:                 Result with metadata including used_signals  
1385:  
1386:             Requires:  
1387:                 - ExecutorConfig is injected  
1388:                 - SignalRegistry is available  
1389:  
1390:             Ensures:  
1391:                 - Result includes used_signals metadata  
1392:                 - Execution is deterministic if seed is set  
1393:             """  
1394:             ...  
1395:  
1396:  
1397: class PortAggregate(Protocol):  
1398:     """Port for aggregating enriched chunks.
```

```
1399:  
1400:    Aggregates processed chunks into PyArrow tables.  
1401:    """  
1402:  
1403:    def aggregate(self, enriched_chunks: list[dict[str, Any]]) -> Any:  
1404:        """Aggregate enriched chunks to PyArrow table.  
1405:  
1406:        Args:  
1407:            enriched_chunks: List of enriched chunk dictionaries  
1408:  
1409:        Returns:  
1410:            PyArrow Table with aggregated data  
1411:  
1412:        Requires:  
1413:            - enriched_chunks has required fields  
1414:            - All chunks have consistent schema  
1415:  
1416:        Ensures:  
1417:            - Returns valid pa.Table  
1418:            - All required columns present  
1419:        """  
1420:        ...  
1421:  
1422:  
1423: class PortScore(Protocol):  
1424:     """Port for scoring features.  
1425:  
1426:     Computes scores from feature tables with specified metrics.  
1427:     """  
1428:  
1429:     def score(self, features: Any, metrics: list[str]) -> Any:  
1430:         """Score features using specified metrics.  
1431:  
1432:         Args:  
1433:             features: PyArrow Table with features  
1434:             metrics: List of metric names to compute  
1435:  
1436:         Returns:  
1437:             Polars DataFrame with scores  
1438:  
1439:         Requires:  
1440:             - features is valid pa.Table  
1441:             - metrics are declared and implemented  
1442:             - Required columns present in features  
1443:  
1444:         Ensures:  
1445:             - Returns valid pl.DataFrame  
1446:             - All requested metrics computed  
1447:         """  
1448:         ...  
1449:  
1450:  
1451: class PortReport(Protocol):  
1452:     """Port for generating reports.  
1453:  
1454:     Generates output reports from scores and manifest.
```

```
1455: """
1456:
1457:     def report(self, scores: Any, manifest: Any) -> dict[str, str]:
1458:         """Generate reports from scores and manifest.
1459:
1460:         Args:
1461:             scores: Polars DataFrame with computed scores
1462:             manifest: Document manifest with metadata
1463:
1464:         Returns:
1465:             Dictionary mapping report name to output URI
1466:
1467:         Requires:
1468:             - scores is valid pl.DataFrame
1469:             - manifest has required metadata
1470:
1471:         Ensures:
1472:             - All declared reports generated
1473:             - URIs are accessible
1474: """
1475: ...
1476:
1477:
1478: class PortDocumentLoader(Protocol):
1479:     """Port for loading documents from various formats.
1480:
1481:     Loads policy documents from PDF, DOCX, and other formats.
1482:     """
1483:
1484:     def load_pdf(self, path: str) -> str:
1485:         """Load text from PDF file.
1486:
1487:         Args:
1488:             path: Path to PDF file
1489:
1490:         Returns:
1491:             Extracted text content
1492:
1493:         Raises:
1494:             FileNotFoundError: If file does not exist
1495: """
1496: ...
1497:
1498:     def load_docx(self, path: str) -> str:
1499:         """Load text from DOCX file.
1500:
1501:         Args:
1502:             path: Path to DOCX file
1503:
1504:         Returns:
1505:             Extracted text content
1506:
1507:         Raises:
1508:             FileNotFoundError: If file does not exist
1509: """
1510: ...
```

```
1511:  
1512:  
1513: class PortMunicipalOntology(Protocol):  
1514:     """Port for municipal policy ontology.  
1515:  
1516:     Provides domain knowledge for policy analysis.  
1517:     """  
1518:  
1519:     def __init__(self) -> None:  
1520:         """Initialize ontology."""  
1521:         ...  
1522:  
1523:  
1524: class PortSemanticAnalyzer(Protocol):  
1525:     """Port for semantic analysis of policy text.  
1526:  
1527:     Extracts semantic features from policy documents.  
1528:     """  
1529:  
1530:     def extract_semantic_cube(self, sentences: list[str]) -> dict[str, Any]:  
1531:         """Extract semantic features from sentences.  
1532:  
1533:         Args:  
1534:             sentences: List of policy text sentences  
1535:  
1536:         Returns:  
1537:             Dictionary containing semantic analysis results  
1538:  
1539:         Requires:  
1540:             - sentences is non-empty list  
1541:         """  
1542:         ...  
1543:  
1544:  
1545: class PortPerformanceAnalyzer(Protocol):  
1546:     """Port for performance analysis.  
1547:  
1548:     Analyzes performance metrics and indicators in policy text.  
1549:     """  
1550:  
1551:     def analyze_performance(self, text: str, semantic_data: dict[str, Any]) -> dict[str, Any]:  
1552:         """Analyze performance indicators and metrics.  
1553:  
1554:         Args:  
1555:             text: Policy text to analyze  
1556:             semantic_data: Semantic features from semantic analyzer  
1557:  
1558:         Returns:  
1559:             Dictionary containing performance analysis results  
1560:         """  
1561:         ...  
1562:  
1563:  
1564: class PortContradictionDetector(Protocol):  
1565:     """Port for detecting contradictions in policy text.  
1566:
```

```
1567:     Identifies logical contradictions and inconsistencies.
1568:     """
1569:
1570:     def detect(
1571:         self,
1572:         text: str,
1573:         plan_name: str,
1574:         dimension: Any,
1575:     ) -> dict[str, Any]:
1576:         """Detect contradictions in policy text.
1577:
1578:         Args:
1579:             text: Policy text to analyze
1580:             plan_name: Name of policy plan
1581:             dimension: Policy dimension to analyze
1582:
1583:         Returns:
1584:             Dictionary containing contradiction analysis results
1585:             """
1586:
1587: ...
1588: def _extract_policy_statements(self, text: str, dimension: Any) -> list[Any]:
1589:     """Extract policy statements for analysis.
1590:
1591:         Args:
1592:             text: Policy text
1593:             dimension: Policy dimension
1594:
1595:         Returns:
1596:             List of extracted policy statements
1597:             """
1598:
1599:
1600:
1601: class PortTemporalLogicVerifier(Protocol):
1602:     """Port for temporal logic verification.
1603:
1604:     Verifies temporal consistency in policy statements.
1605:     """
1606:
1607:     def verify_temporal_consistency(self, statements: list[Any]) -> tuple[bool, list[dict[str, Any]]]:
1608:         """Verify temporal consistency of statements.
1609:
1610:         Args:
1611:             statements: List of policy statements to verify
1612:
1613:         Returns:
1614:             Tuple of (is_consistent, list_of_conflicts)
1615:             """
1616:
1617:
1618:
1619: class PortBayesianConfidenceCalculator(Protocol):
1620:     """Port for Bayesian confidence calculation.
1621:
1622:     Calculates posterior confidence scores using Bayesian methods.
```

```
1623:     """
1624:
1625:     def calculate_posterior(
1626:         self,
1627:         evidence_strength: float,
1628:         observations: int,
1629:         domain_weight: float = 1.0,
1630:     ) -> float:
1631:         """Calculate posterior confidence score.
1632:
1633:         Args:
1634:             evidence_strength: Strength of evidence (0-1)
1635:             observations: Number of observations
1636:             domain_weight: Domain-specific weight
1637:
1638:         Returns:
1639:             Posterior probability (0-1)
1640:         """
1641:         ...
1642:
1643:
1644: class PortMunicipalAnalyzer(Protocol):
1645:     """Port for municipal policy analysis.
1646:
1647:     Performs comprehensive municipal policy analysis.
1648:     """
1649:
1650:     def _generate_summary(
1651:         self,
1652:         text: str,
1653:         evidence_by_dimension: dict[str, Any],
1654:         dimension_scores: dict[str, float],
1655:     ) -> dict[str, Any]:
1656:         """Generate analysis summary.
1657:
1658:         Args:
1659:             text: Policy text
1660:             evidence_by_dimension: Evidence organized by dimension
1661:             dimension_scores: Scores for each dimension
1662:
1663:         Returns:
1664:             Dictionary containing analysis summary
1665:         """
1666:         ...
1667:
1668:
1669:     __all__ = [
1670:         'FilePort',
1671:         'JsonPort',
1672:         'EnvPort',
1673:         'ClockPort',
1674:         'LogPort',
1675:         'PortCPPIngest',
1676:         'PortCPPAdapter',
1677:         'PortSPCAdapter',
1678:         'PortSignalsClient',
```

```
1679:     'PortSignalsRegistry',
1680:     'PortArgRouter',
1681:     'PortExecutor',
1682:     'PortAggregate',
1683:     'PortScore',
1684:     'PortReport',
1685:     'PortDocumentLoader',
1686:     'PortMunicipalOntology',
1687:     'PortSemanticAnalyzer',
1688:     'PortPerformanceAnalyzer',
1689:     'PortContradictionDetector',
1690:     'PortTemporalLogicVerifier',
1691:     'PortBayesianConfidenceCalculator',
1692:     'PortMunicipalAnalyzer',
1693:     'RecommendationEnginePort',
1694: ]
1695:
1696:
1697:
1698: =====
1699: FILE: src/farfan_pipeline/core/runtime_config.py
1700: =====
1701:
1702: """
1703: Global runtime configuration system for F.A.R.F.A.N.
1704:
1705: This module provides runtime mode enforcement (PROD/DEV/EXPLORATORY) with strict
1706: fallback policies, configuration validation, and environment variable parsing.
1707:
1708: FALLBACK CATEGORIZATION AND ASSESSMENT:
1709:
1710: CATEGORY A (CRITICAL - System Integrity):
1711:     Variables: ALLOW_CONTRADICTION_FALLBACK, ALLOW_VALIDATOR_DISABLE, ALLOW_EXECUTION_ESTIMATES
1712:     Assessment: These indicate missing CRITICAL components. In PROD, the system MUST fail fast
1713:     to prevent incorrect analysis results. No fallback is acceptable.
1714:
1715: CATEGORY B (QUALITY - Quality Degradation):
1716:     Variables: ALLOW_NETWORKX_FALLBACK, ALLOW_SPACY_FALLBACK
1717:     Assessment: These degrade output quality but don't invalidate core analysis. Allowed in
1718:     PROD with explicit flag and warnings logged. Results remain scientifically valid but less rich.
1719:
1720: CATEGORY C (DEVELOPMENT - Development Convenience):
1721:     Variables: ALLOW_DEV_INGESTION_FALLBACKS, ALLOW_AGGREGATION_DEFAULTS, ALLOW_MISSING_BASE_WEIGHTS
1722:     Assessment: STRICTLY FORBIDDEN in PROD. These exist only for development/testing to avoid
1723:     infrastructure dependencies. Using these in PROD invalidates results.
1724:
1725: CATEGORY D (OPERATIONAL - Operational Flexibility):
1726:     Variables: ALLOW_HASH_FALLBACK, ALLOW_PDFPLUMBER_FALLBACK
1727:     Assessment: Safe fallbacks maintaining correctness with different implementation strategies.
1728:     Generally allowed as they don't affect scientific validity.
1729:
1730: Environment Variables:
1731:     SAAAAAA_RUNTIME_MODE: Runtime mode (prod/dev/exploratory), default: prod
1732:
1733:     # Category A - Critical System Integrity
1734:     ALLOW_CONTRADICTION_FALLBACK: Allow contradiction detection fallback, default: false
```

```
1735: ALLOW_VALIDATOR_DISABLE: Allow wiring validator disabling, default: false
1736: ALLOW_EXECUTION_ESTIMATES: Allow execution metric estimation, default: false
1737:
1738: # Category B - Quality Degradation
1739: ALLOW_NETWORKX_FALLBACK: Allow NetworkX unavailability, default: false
1740: ALLOW_SPACY_FALLBACK: Allow spaCy model fallback, default: false
1741:
1742: # Category C - Development Convenience (FORBIDDEN in PROD)
1743: ALLOW_DEV_INGESTION_FALLBACKS: Allow dev ingestion fallbacks, default: false
1744: ALLOW_AGGREGATION_DEFAULTS: Allow aggregation defaults, default: false
1745: ALLOW_MISSING_BASE_WEIGHTS: Allow missing _base_weights in calibration, default: false
1746:
1747: # Category D - Operational Flexibility
1748: ALLOW_HASH_FALLBACK: Allow hash algorithm fallback, default: true
1749: ALLOW_PDFPLUMBER_FALLBACK: Allow pdfplumber unavailability, default: false
1750:
1751: # Calibration and Quality Controls
1752: STRICT_CALIBRATION: Require complete calibration files, default: true
1753:
1754: # Model and Processing Configuration
1755: PREFERRED_SPACY_MODEL: Preferred spaCy model, default: es_core_news_lg
1756: PREFERRED_EMBEDDING_MODEL: Preferred embedding model, default: sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2
1757:
1758: # Path Configuration
1759: SAAAAAA_PROJECT_ROOT: Project root override
1760: SAAAAAA_DATA_DIR: Data directory override
1761: SAAAAAA_OUTPUT_DIR: Output directory override
1762: SAAAAAA_CACHE_DIR: Cache directory override
1763: SAAAAAA_LOGS_DIR: Logs directory override
1764:
1765: # External Dependencies
1766: HF_ONLINE: Allow HuggingFace online access (0 or 1), default: 0
1767:
1768: # Processing Limits
1769: EXPECTED_QUESTION_COUNT: Expected question count, default: 305
1770: EXPECTED_METHOD_COUNT: Expected method count, default: 416
1771: PHASE_TIMEOUT_SECONDS: Phase timeout in seconds, default: 300
1772: MAX_WORKERS: Maximum worker threads, default: 4
1773: BATCH_SIZE: Batch size for processing, default: 100
1774:
1775: Example:
1776:     >>> config = RuntimeConfig.from_env()
1777:     >>> if config.mode == RuntimeMode.PROD:
1778:         ...     assert not config.allow_dev_ingestionFallbacks
1779: """
1780:
1781: import os
1782: from dataclasses import dataclass
1783: from enum import Enum
1784: from typing import ClassVar, Optional
1785:
1786:
1787: class RuntimeMode(Enum):
1788:     """Runtime execution mode with different strictness levels."""
1789:
1790:     PROD = "prod"
```

```
1791:     """Production mode: strict enforcement, no fallbacks unless explicitly allowed."""
1792:
1793:     DEV = "dev"
1794:     """Development mode: permissive with flags, allows controlled degradation."""
1795:
1796:     EXPLORATORY = "exploratory"
1797:     """Exploratory mode: maximum flexibility for research and experimentation."""
1798:
1799:
1800: class FallbackCategory(Enum):
1801:     """Categorization of fallback types by impact."""
1802:
1803:     CRITICAL = "critical"
1804:     """Category A: System integrity - failures indicate missing critical dependencies."""
1805:
1806:     QUALITY = "quality"
1807:     """Category B: Quality degradation - system continues with reduced quality."""
1808:
1809:     DEVELOPMENT = "development"
1810:     """Category C: Development convenience - only allowed in DEV/EXPLORATORY."""
1811:
1812:     OPERATIONAL = "operational"
1813:     """Category D: Operational flexibility - safe fallbacks for operational concerns."""
1814:
1815:
1816: class ConfigurationError(Exception):
1817:     """Raised when runtime configuration is invalid or contains illegal combinations."""
1818:
1819:     def __init__(self, message: str, illegal_combo: str | None = None) -> None:
1820:         self.illegal_combo = illegal_combo
1821:         super().__init__(message)
1822:
1823:
1824: @dataclass(frozen=True)
1825: class RuntimeConfig:
1826:     """
1827:         Immutable runtime configuration parsed from environment variables.
1828:
1829:         This configuration controls system behavior across all components, enforcing
1830:         strict policies in PROD mode and allowing controlled degradation in DEV/EXPLORATORY.
1831:
1832:         Attributes:
1833:             mode: Runtime execution mode
1834:
1835:             # Category A - Critical System Integrity
1836:             allow_contradictionFallback: Allow fallback when contradiction module unavailable
1837:             allow_validator_disable: Allow disabling wiring validator
1838:             allow_execution_estimates: Allow execution metric estimation
1839:
1840:             # Category B - Quality Degradation
1841:             allow_networkxFallback: Allow NetworkX unavailability
1842:             allow_spacyFallback: Allow spaCy model fallback
1843:
1844:             # Category C - Development Convenience
1845:             allow_dev_ingestionFallbacks: Allow development ingestion fallbacks
1846:             allow_aggregation_defaults: Allow aggregation default values
```

```
1847:     allow_missing_base_weights: Allow missing _base_weights in calibration
1848:
1849:     # Category D - Operational Flexibility
1850:     allow_hashFallback: Allow hash algorithm fallback
1851:     allow_pdfplumberFallback: Allow pdfplumber unavailability
1852:
1853:     # Calibration
1854:     strict_calibration: Require complete calibration files with _base_weights
1855:
1856:     # Model Configuration
1857:     preferred_spacy_model: Preferred spaCy model name
1858:     preferred_embedding_model: Preferred embedding model name
1859:
1860:     # Path Configuration
1861:     project_root_override: Project root path override
1862:     data_dir_override: Data directory override
1863:     output_dir_override: Output directory override
1864:     cache_dir_override: Cache directory override
1865:     logs_dir_override: Logs directory override
1866:
1867:     # External Dependencies
1868:     hf_online: Allow HuggingFace online access
1869:
1870:     # Processing Configuration
1871:     expected_question_count: Expected question count for validation
1872:     expected_method_count: Expected method count for validation
1873:     phase_timeout_seconds: Phase timeout in seconds
1874:     max_workers: Maximum worker threads
1875:     batch_size: Batch size for processing
1876: """
1877:
1878: mode: RuntimeMode
1879:
1880:     # Category A - Critical
1881:     allow_contradictionFallback: bool
1882:     allow_validatorDisable: bool
1883:     allow_executionEstimates: bool
1884:
1885:     # Category B - Quality
1886:     allow_networkxFallback: bool
1887:     allow_spacyFallback: bool
1888:
1889:     # Category C - Development
1890:     allow_devIngestionFallbacks: bool
1891:     allow_aggregationDefaults: bool
1892:     allow_missingBaseWeights: bool
1893:
1894:     # Category D - Operational
1895:     allow_hashFallback: bool
1896:     allow_pdfplumberFallback: bool
1897:
1898:     # Calibration
1899:     strict_calibration: bool
1900:
1901:     # Model Configuration
1902:     preferred_spacy_model: str
```

```
1903:     preferred_embedding_model: str
1904:
1905:     # Path Configuration
1906:     project_root_override: Optional[str]
1907:     data_dir_override: Optional[str]
1908:     output_dir_override: Optional[str]
1909:     cache_dir_override: Optional[str]
1910:     logs_dir_override: Optional[str]
1911:
1912:     # External Dependencies
1913:     hf_online: bool
1914:
1915:     # Processing Configuration
1916:     expected_question_count: int
1917:     expected_method_count: int
1918:     phase_timeout_seconds: int
1919:     max_workers: int
1920:     batch_size: int
1921:
1922:     # Illegal combinations in PROD mode
1923:     _PROD_ILLEGAL_COMBOs: ClassVar[dict[str, tuple[str, FallbackCategory]]] = {
1924:         "ALLOW_DEV_INGESTION_FALLBACKS": (
1925:             "Development ingestion fallbacks not allowed in PROD - they bypass quality gates",
1926:             FallbackCategory.DEVELOPMENT
1927:         ),
1928:         "ALLOW_EXECUTION_ESTIMATES": (
1929:             "Execution metric estimation not allowed in PROD - actual measurements required",
1930:             FallbackCategory.CRITICAL
1931:         ),
1932:         "ALLOW_AGGREGATION_DEFAULTS": (
1933:             "Aggregation defaults not allowed in PROD - explicit calibration required",
1934:             FallbackCategory.DEVELOPMENT
1935:         ),
1936:         "ALLOW_MISSING_BASE_WEIGHTS": (
1937:             "Missing base weights not allowed in PROD - complete calibration required",
1938:             FallbackCategory.DEVELOPMENT
1939:         ),
1940:     }
1941:
1942:     @classmethod
1943:     def from_env(cls) -> "RuntimeConfig":
1944:         """
1945:             Parse runtime configuration from environment variables.
1946:
1947:             Returns:
1948:                 RuntimeConfig: Validated configuration instance
1949:
1950:             Raises:
1951:                 ConfigurationError: If configuration is invalid or contains illegal combinations
1952:
1953:             Example:
1954:                 >>> os.environ['SAAAAAA_RUNTIME_MODE'] = 'prod'
1955:                 >>> config = RuntimeConfig.from_env()
1956:                 >>> assert config.mode == RuntimeMode.PROD
1957:                 """
1958:             # Parse runtime mode
```

```
1959:     mode_str = os.getenv("SAAAAAA_RUNTIME_MODE", "prod").lower()
1960:     try:
1961:         mode = RuntimeMode(mode_str)
1962:     except ValueError as e:
1963:         raise ConfigurationError(
1964:             f"Invalid SAAAAAA_RUNTIME_MODE: {mode_str}. "
1965:             f"Must be one of: {', '.join(m.value for m in RuntimeMode)}"
1966:         ) from e
1967:
1968:     # Parse Category A - Critical Fallbacks
1969:     allow_contradiction_fallback = _parse_bool_env("ALLOW_CONTRADICTION_FALLBACK", False)
1970:     allow_validator_disable = _parse_bool_env("ALLOW_VALIDATOR_DISABLE", False)
1971:     allow_execution_estimates = _parse_bool_env("ALLOW_EXECUTION_ESTIMATES", False)
1972:
1973:     # Parse Category B - Quality Fallbacks
1974:     allow_networkx_fallback = _parse_bool_env("ALLOW_NETWORKX_FALLBACK", False)
1975:     allow_spacy_fallback = _parse_bool_env("ALLOW_SPACY_FALLBACK", False)
1976:
1977:     # Parse Category C - Development Fallbacks
1978:     allow_dev_ingestionFallbacks = _parse_bool_env("ALLOW_DEV_INGESTION_FALLBACKS", False)
1979:     allow_aggregation_defaults = _parse_bool_env("ALLOW_AGGREGATION_DEFAULTS", False)
1980:     allow_missing_base_weights = _parse_bool_env("ALLOW_MISSING_BASE_WEIGHTS", False)
1981:
1982:     # Parse Category D - Operational Fallbacks
1983:     allow_hash_fallback = _parse_bool_env("ALLOW_HASH_FALLBACK", True)
1984:     allow_pdfplumber_fallback = _parse_bool_env("ALLOW_PDFPLUMBER_FALLBACK", False)
1985:
1986:     # Parse calibration config
1987:     strict_calibration = _parse_bool_env("STRICT_CALIBRATION", True)
1988:
1989:     # Parse model configuration
1990:     preferred_spacy_model = os.getenv("PREFERRED_SPACY_MODEL", "es_core_news_lg")
1991:     preferred_embedding_model = os.getenv(
1992:         "PREFERRED_EMBEDDING_MODEL",
1993:         "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
1994:     )
1995:
1996:     # Parse path configuration
1997:     project_root_override = os.getenv("SAAAAAA_PROJECT_ROOT")
1998:     data_dir_override = os.getenv("SAAAAAA_DATA_DIR")
1999:     output_dir_override = os.getenv("SAAAAAA_OUTPUT_DIR")
2000:     cache_dir_override = os.getenv("SAAAAAA_CACHE_DIR")
2001:     logs_dir_override = os.getenv("SAAAAAA_LOGS_DIR")
2002:
2003:     # Parse external dependencies
2004:     hf_online = os.getenv("HF_ONLINE", "0") == "1"
2005:
2006:     # Parse processing configuration
2007:     expected_question_count = _parse_int_env("EXPECTED_QUESTION_COUNT", 305)
2008:     expected_method_count = _parse_int_env("EXPECTED_METHOD_COUNT", 416)
2009:     phase_timeout_seconds = _parse_int_env("PHASE_TIMEOUT_SECONDS", 300)
2010:     max_workers = _parse_int_env("MAX_WORKERS", 4)
2011:     batch_size = _parse_int_env("BATCH_SIZE", 100)
2012:
2013:     # Create config instance
2014:     config = cls(
```

```
2015:     mode=mode,
2016:     allow_contradictionFallback=allow_contradictionFallback,
2017:     allow_validatorDisable=allow_validatorDisable,
2018:     allow_executionEstimates=allow_executionEstimates,
2019:     allow_networkxFallback=allow_networkxFallback,
2020:     allow_spacyFallback=allow_spacyFallback,
2021:     allow_devIngestionFallbacks=allow_devIngestionFallbacks,
2022:     allow_aggregationDefaults=allow_aggregationDefaults,
2023:     allow_missingBaseWeights=allow_missingBaseWeights,
2024:     allow_hashFallback=allow_hashFallback,
2025:     allow_pdfplumberFallback=allow_pdfplumberFallback,
2026:     strictCalibration=strictCalibration,
2027:     preferredSpacyModel=preferredSpacyModel,
2028:     preferredEmbeddingModel=preferredEmbeddingModel,
2029:     projectRootOverride=projectRootOverride,
2030:     dataDirOverride=dataDirOverride,
2031:     outputDirOverride=outputDirOverride,
2032:     cacheDirOverride=cacheDirOverride,
2033:     logsDirOverride=logsDirOverride,
2034:     hfOnline=hfOnline,
2035:     expectedQuestionCount=expectedQuestionCount,
2036:     expectedMethodCount=expectedMethodCount,
2037:     phaseTimeoutSeconds=phaseTimeoutSeconds,
2038:     maxWorkers=maxWorkers,
2039:     batchSize=batchSize,
2040: )
2041:
2042: # Validate configuration
2043: config._validate()
2044:
2045: return config
2046:
2047: def _validate(self) -> None:
2048: """
2049:     Validate configuration for illegal combinations.
2050:
2051:     In PROD mode, certain ALLOW_* flags are prohibited to ensure strict behavior.
2052:
2053:     Raises:
2054:         ConfigurationError: If illegal combination detected
2055: """
2056: if self.mode != RuntimeMode.PROD:
2057:     return # DEV/EXPLORATORY modes allow all combinations
2058:
2059: # Check for illegal PROD combinations
2060: violations = []
2061:
2062: if self.allowDevIngestionFallbacks:
2063:     msg, cat = self._PROD_ILLEGAL_COMBO["ALLOW_DEV_INGESTION_FALLBACKS"]
2064:     violations.append(
2065:         f"PROD + ALLOW_DEV_INGESTION_FALLBACKS=true: {msg} [Category: {cat.value}]"
2066:     )
2067:
2068: if self.allowExecutionEstimates:
2069:     msg, cat = self._PROD_ILLEGAL_COMBO["ALLOW_EXECUTION_ESTIMATES"]
2070:     violations.append(
```

```
2071:         f"PROD + ALLOW_EXECUTION_ESTIMATES=true: {msg} [Category: {cat.value}]"
2072:     )
2073:
2074:     if self.allow_aggregation_defaults:
2075:         msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_AGGREGATION_DEFAULTS"]
2076:         violations.append(
2077:             f"PROD + ALLOW_AGGREGATION_DEFAULTS=true: {msg} [Category: {cat.value}]"
2078:         )
2079:
2080:     if self.allow_missing_base_weights:
2081:         msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_MISSING_BASE_WEIGHTS"]
2082:         violations.append(
2083:             f"PROD + ALLOW_MISSING_BASE_WEIGHTS=true: {msg} [Category: {cat.value}]"
2084:         )
2085:
2086:     if violations:
2087:         raise ConfigurationError(
2088:             "Illegal configuration combinations detected:\n" + "\n".join(f" - {v}" for v in violations),
2089:             illegal_combo="; ".join(violations)
2090:         )
2091:
2092:     def is_strict_mode(self) -> bool:
2093:         """Check if running in strict mode (PROD with no fallbacks allowed)."""
2094:         return (
2095:             self.mode == RuntimeMode.PROD
2096:             and not self.allow_contradiction_fallback
2097:             and not self.allow_validator_disable
2098:         )
2099:
2100:     def get_fallback_summary(self) -> dict[str, dict[str, bool]]:
2101:         """
2102:             Get summary of all fallback configurations grouped by category.
2103:
2104:             Returns:
2105:                 Dictionary mapping category names to flag dictionaries
2106:         """
2107:         return {
2108:             "critical": {
2109:                 "contradiction_fallback": self.allow_contradiction_fallback,
2110:                 "validator_disable": self.allow_validator_disable,
2111:                 "execution_estimates": self.allow_execution_estimates,
2112:             },
2113:             "quality": {
2114:                 "networkx_fallback": self.allow_networkx_fallback,
2115:                 "spacy_fallback": self.allow_spacy_fallback,
2116:             },
2117:             "development": {
2118:                 "dev_ingestion_fallbacks": self.allow_dev_ingestion_fallbacks,
2119:                 "aggregation_defaults": self.allow_aggregation_defaults,
2120:                 "missing_base_weights": self.allow_missing_base_weights,
2121:             },
2122:             "operational": {
2123:                 "hash_fallback": self.allow_hash_fallback,
2124:                 "pdfplumber_fallback": self.allow_pdfplumber_fallback,
2125:             },
2126:         }
```

```
2127:  
2128:     def __repr__(self) -> str:  
2129:         """String representation showing mode and key flags."""  
2130:         flags = []  
2131:         if self.allow_contradiction_fallback:  
2132:             flags.append("contradiction_fallback")  
2133:         if self.allow_validator_disable:  
2134:             flags.append("validator_disable")  
2135:         if self.allow_execution_estimates:  
2136:             flags.append("execution_estimates")  
2137:         if self.allow_networkx_fallback:  
2138:             flags.append("networkx_fallback")  
2139:         if self.allow_spacy_fallback:  
2140:             flags.append("spacy_fallback")  
2141:         if self.allow_dev_ingestion_FALLBACKS:  
2142:             flags.append("dev_ingestion_FALLBACKS")  
2143:         if self.allow_aggregation_defaults:  
2144:             flags.append("aggregation_defaults")  
2145:         if self.allow_missing_base_weights:  
2146:             flags.append("missing_base_weights")  
2147:         if not self.strict_calibration:  
2148:             flags.append("relaxed_calibration")  
2149:  
2150:         flags_str = f", flags={flags}" if flags else ""  
2151:         return f"RuntimeConfig(mode={self.mode.value}{flags_str})"  
2152:  
2153:  
2154: def _parse_bool_env(var_name: str, default: bool) -> bool:  
2155:     """  
2156:         Parse boolean environment variable with case-insensitive handling.  
2157:  
2158:     Args:  
2159:         var_name: Environment variable name  
2160:         default: Default value if not set  
2161:  
2162:     Returns:  
2163:         Parsed boolean value  
2164:  
2165:     Raises:  
2166:         ConfigurationError: If value is not a valid boolean  
2167:     """  
2168:     value = os.getenv(var_name)  
2169:     if value is None:  
2170:         return default  
2171:  
2172:     value_lower = value.lower()  
2173:     if value_lower in ("true", "1", "yes", "on"):  
2174:         return True  
2175:     elif value_lower in ("false", "0", "no", "off"):  
2176:         return False  
2177:     else:  
2178:         raise ConfigurationError(  
2179:             f"Invalid boolean value for {var_name}: {value}. "  
2180:             f"Must be one of: true/false, 1/0, yes/no, on/off"  
2181:         )  
2182:
```

```
2183:  
2184: def _parse_int_env(var_name: str, default: int) -> int:  
2185:     """  
2186:         Parse integer environment variable with validation.  
2187:  
2188:     Args:  
2189:         var_name: Environment variable name  
2190:         default: Default value if not set  
2191:  
2192:     Returns:  
2193:         Parsed integer value  
2194:  
2195:     Raises:  
2196:         ConfigurationError: If value is not a valid integer  
2197:     """  
2198:     value = os.getenv(var_name)  
2199:     if value is None:  
2200:         return default  
2201:  
2202:     try:  
2203:         return int(value)  
2204:     except ValueError:  
2205:         raise ConfigurationError(  
2206:             f"Invalid integer value for {var_name}: {value}. "  
2207:             f"Must be a valid integer."  
2208:     )  
2209:  
2210:  
2211: # Global singleton instance (lazy-initialized)  
2212: _global_config: RuntimeConfig | None = None  
2213:  
2214:  
2215: def get_runtime_config() -> RuntimeConfig:  
2216:     """  
2217:         Get global runtime configuration instance (lazy-initialized).  
2218:  
2219:     Returns:  
2220:         RuntimeConfig: Global configuration instance  
2221:  
2222:     Note:  
2223:         This is initialized once on first call. For testing, use from_env() directly.  
2224:     """  
2225:     global _global_config  
2226:     if _global_config is None:  
2227:         _global_config = RuntimeConfig.from_env()  
2228:     return _global_config  
2229:  
2230:  
2231: def reset_runtime_config() -> None:  
2232:     """  
2233:         Reset global runtime configuration (for testing only).  
2234:  
2235:     Warning:  
2236:         This should only be used in tests. Production code should never reset config.  
2237:     """  
2238:     global _global_config
```

```
2239:     _global_config = None
2240:
2241:
2242:
2243: =====
2244: FILE: src/farfan_pipeline/core/types.py
2245: =====
2246:
2247: """
2248: Core type definitions shared across layers.
2249:
2250: This module contains types that need to be referenced by both core and analysis
2251: layers without creating circular dependencies.
2252: """
2253:
2254: from __future__ import annotations
2255:
2256: import re
2257: from dataclasses import dataclass, field
2258: from datetime import datetime
2259: from enum import Enum
2260: from pathlib import Path
2261: from typing import Any, Literal
2262:
2263: __all__ = [
2264:     "CategoriaCausal",
2265:     "ChunkData",
2266:     "PreprocessedDocument",
2267:     "Provenance",
2268: ]
2269:
2270:
2271: class CategoriaCausal(Enum):
2272:     """
2273:     JerarquÃ-a axiomÃtica de categorÃ-as causales en una teorÃ-a de cambio.
2274:     El orden numÃ©rico impone la secuencia lÃ³gica obligatoria.
2275:
2276:     Originally from farfan_core.analysis.teoria_cambio, moved here to break
2277:     architectural dependency (core should not import from analysis).
2278:     """
2279:
2280:     INSUMOS = 1
2281:     ACTIVIDADES = 2
2282:     PRODUCTOS = 3
2283:     RESULTADOS = 4
2284:     CAUSALIDAD = 5
2285:
2286:
2287: @dataclass(frozen=True)
2288: class Provenance:
2289:     """Provenance metadata for a chunk."""
2290:
2291:     page_number: int
2292:     section_header: str | None = None
2293:     bbox: tuple[float, float, float, float] | None = None
2294:     span_in_page: tuple[int, int] | None = None
```

```
2295:     source_file: str | None = None
2296:
2297:
2298: @dataclass(frozen=True)
2299: class ChunkData:
2300:     """Single semantic chunk from SPC (Smart Policy Chunks).
2301:
2302:     Preserves chunk structure and metadata from the ingestion pipeline,
2303:     enabling chunk-aware executor routing and scoped processing.
2304:     """
2305:
2306:     id: int
2307:     text: str
2308:     chunk_type: Literal[
2309:         "diagnostic", "activity", "indicator", "resource", "temporal", "entity"
2310:     ]
2311:     sentences: list[int]
2312:     tables: list[int]
2313:     start_pos: int
2314:     end_pos: int
2315:     confidence: float
2316:     chunk_id: str | None = None
2317:     edges_out: list[int] = field(default_factory=list)
2318:     edges_in: list[int] = field(default_factory=list)
2319:     policy_area_id: str | None = None
2320:     dimension_id: str | None = None
2321:     provenance: Provenance | None = None
2322:     expected_elements: list[dict[str, Any]] = field(default_factory=list)
2323:     document_position: tuple[int, int] | None = None
2324:
2325: _CHUNK_ID_PATTERN = re.compile(r"^\w{1,9}\w{10}-\w{1,6}$")
2326:
2327: def __post_init__(self) -> None:
2328:     """Validate chunk_id presence and format (PA{01-10}-DIM{01-06}) and new fields.
2329:
2330:     Enforces Phase 1 output contract:
2331:     - Non-empty text content
2332:     - Valid chunk_id format (PA01-PA10, DIM01-DIM06)
2333:     - Consistency between chunk_id and policy_area_id/dimension_id
2334:     - Valid expected_elements structure
2335:     - Valid document_position range
2336:     """
2337:     import logging
2338:
2339:     logger = logging.getLogger(__name__)
2340:
2341:     if not isinstance(self.text, str):
2342:         raise ValueError(
2343:             f"ChunkData text must be a string, got {type(self.text).__name__}"
2344:         )
2345:
2346:     if not self.text or not self.text.strip():
2347:         raise ValueError(
2348:             "ChunkData text cannot be empty or whitespace-only. "
2349:             "Phase 1 must populate all chunks with non-empty text content."
2350:         )
```

```
2351:  
2352:         self._validate_expected_elements()  
2353:         self._validate_document_position(logger)  
2354:  
2355:         chunk_id = self.chunk_id  
2356:         if chunk_id is None:  
2357:             if self.policy_area_id and self.dimension_id:  
2358:                 chunk_id = f"{self.policy_area_id}-{self.dimension_id}"  
2359:                 object.__setattr__(self, "chunk_id", chunk_id)  
2360:             else:  
2361:                 raise ValueError(  
2362:                     "chunk_id is required and must follow format PA{01-10}-DIM{01-06}. "  
2363:                     "Provide chunk_id explicitly or set both policy_area_id and dimension_id "  
2364:                     "to derive it. Phase 1 must populate these fields for all chunks."  
2365:                 )  
2366:  
2367:         if not self._CHUNK_ID_PATTERN.match(chunk_id):  
2368:             raise ValueError(  
2369:                 f"Invalid chunk_id '{chunk_id}'. Expected format PA{{01-10}}-DIM{{01-06}}. "  
2370:                 "Valid examples: 'PA01-DIM01', 'PA10-DIM06'. "  
2371:                 "Phase 1 must generate chunk_id values that match this pattern."  
2372:             )  
2373:  
2374:         match = self._CHUNK_ID_PATTERN.match(chunk_id)  
2375:         if match:  
2376:             pa_code = f"PA{match.group(1)}"  
2377:             dim_code = f"DIM{match.group(2)}"  
2378:             if self.policy_area_id and self.policy_area_id != pa_code:  
2379:                 raise ValueError(  
2380:                     f"chunk_id '{chunk_id}' mismatches policy_area_id '{self.policy_area_id}'. "  
2381:                     f"Expected policy_area_id to be '{pa_code}' based on chunk_id. "  
2382:                     "Phase 1 must ensure consistency between chunk_id and metadata fields."  
2383:                 )  
2384:             if self.dimension_id and self.dimension_id != dim_code:  
2385:                 raise ValueError(  
2386:                     f"chunk_id '{chunk_id}' mismatches dimension_id '{self.dimension_id}'. "  
2387:                     f"Expected dimension_id to be '{dim_code}' based on chunk_id. "  
2388:                     "Phase 1 must ensure consistency between chunk_id and metadata fields."  
2389:                 )  
2390:  
2391:     def _validate_expected_elements(self) -> None:  
2392:         """Validate expected_elements field structure."""  
2393:         if not isinstance(self.expected_elements, list):  
2394:             raise ValueError(  
2395:                 f"expected_elements must be a list, got {type(self.expected_elements).__name__}"  
2396:             )  
2397:  
2398:         for idx, element in enumerate(self.expected_elements):  
2399:             if not isinstance(element, dict):  
2400:                 raise ValueError(  
2401:                     f"expected_elements[{idx}] must be a dict, got {type(element).__name__}"  
2402:                 )  
2403:  
2404:             if "type" not in element:  
2405:                 raise ValueError(  
2406:                     f"expected_elements[{idx}] missing required 'type' key"
```

```
2407:         )
2408:
2409:     if not isinstance(element["type"], str):
2410:         raise ValueError(
2411:             f"expected_elements[{idx}]['type'] must be a string, "
2412:             f"got {type(element['type'])).__name__}"
2413:         )
2414:
2415:     if "required" in element:
2416:         if not isinstance(element["required"], bool):
2417:             raise ValueError(
2418:                 f"expected_elements[{idx}]['required'] must be a boolean, "
2419:                 f"got {type(element['required'])).__name__}"
2420:             )
2421:
2422:     if "minimum" in element:
2423:         if not isinstance(element["minimum"], int):
2424:             raise ValueError(
2425:                 f"expected_elements[{idx}]['minimum'] must be an integer, "
2426:                 f"got {type(element['minimum'])).__name__}"
2427:             )
2428:         if element["minimum"] < 0:
2429:             raise ValueError(
2430:                 f"expected_elements[{idx}]['minimum'] must be non-negative, "
2431:                 f"got {element['minimum']}"
2432:             )
2433:
2434:     def _validate_document_position(self, logger: Any) -> None:
2435:         """Validate document_position field structure."""
2436:         if self.document_position is None:
2437:             return
2438:
2439:         if not isinstance(self.document_position, tuple):
2440:             raise ValueError(
2441:                 f"document_position must be a tuple, got {type(self.document_position).__name__}"
2442:             )
2443:
2444:         if len(self.document_position) != 2:
2445:             raise ValueError(
2446:                 f"document_position must have exactly 2 elements, got {len(self.document_position)}"
2447:             )
2448:
2449:         start, end = self.document_position
2450:
2451:         if not isinstance(start, int):
2452:             raise ValueError(
2453:                 f"document_position[0] (start) must be an integer, got {type(start).__name__}"
2454:             )
2455:
2456:         if not isinstance(end, int):
2457:             raise ValueError(
2458:                 f"document_position[1] (end) must be an integer, got {type(end).__name__}"
2459:             )
2460:
2461:         if start < 0:
2462:             raise ValueError(
```

```
2463:             f"document_position start offset must be non-negative, got {start}"
2464:         )
2465:
2466:     if end < start:
2467:         raise ValueError(
2468:             f"document_position end offset ({end}) must be >= start offset ({start})"
2469:         )
2470:
2471:     if start == end:
2472:         logger.warning(
2473:             f"ChunkData {self.id} has zero-length document_position [{start}, {end}]"
2474:         )
2475:
2476:
2477: @dataclass
2478: class PreprocessedDocument:
2479:     """Orchestrator representation of a processed document.
2480:
2481:     This is the normalized document format used internally by the orchestrator.
2482:     It can be constructed from ingestion payloads or created directly.
2483:     """
2484:
2485:     document_id: str
2486:     raw_text: str
2487:     sentences: list[Any]
2488:     tables: list[Any]
2489:     metadata: dict[str, Any]
2490:     source_path: Path | None = None
2491:     sentence_metadata: list[Any] = field(default_factory=list)
2492:     indexes: dict[str, Any] | None = None
2493:     structured_text: dict[str, Any] | None = None
2494:     language: str | None = None
2495:     ingested_at: datetime | None = None
2496:     full_text: str | None = None
2497:
2498:     chunks: list[ChunkData] = field(default_factory=list)
2499:     chunk_index: dict[str, int] = field(default_factory=dict)
2500:     chunk_graph: dict[str, Any] = field(default_factory=dict)
2501:     processing_mode: Literal["flat", "chunked"] = "chunked"
2502:
2503:     def __post_init__(self) -> None:
2504:         """Validate document fields after initialization.
2505:
2506:         Raises:
2507:             ValueError: If raw_text is empty, chunks are missing, or processing mode is invalid
2508:         """
2509:         if (not self.raw_text or not self.raw_text.strip()) and self.full_text:
2510:             self.raw_text = self.full_text
2511:         if not self.raw_text or not self.raw_text.strip():
2512:             raise ValueError(
2513:                 "PreprocessedDocument cannot have empty raw_text. "
2514:                 "Use PreprocessedDocument.ensure() to create from SPC pipeline. "
2515:                 "Phase 1 must populate raw_text field."
2516:             )
2517:         if self.processing_mode != "chunked":
2518:             raise ValueError(
```

```
2519:             f"processing_mode must be 'chunked' for irrigation; got {self.processing_mode!r}. "
2520:             "Phase 1 must set processing_mode to 'chunked' for chunk-based routing."
2521:         )
2522:
2523:         if not isinstance(self.chunks, list):
2524:             raise ValueError(
2525:                 f"PreprocessedDocument.chunks must be a list, got {type(self.chunks).__name__}. "
2526:                 "Phase 1 must populate chunks as a list of ChunkData instances."
2527:             )
2528:
2529:         for idx, chunk in enumerate(self.chunks):
2530:             if not isinstance(chunk, ChunkData):
2531:                 raise ValueError(
2532:                     f"PreprocessedDocument.chunks[{idx}] must be ChunkData instance, "
2533:                     f"got {type(chunk).__name__}. "
2534:                     "Phase 1 must produce only ChunkData instances."
2535:                 )
2536:
2537:     @staticmethod
2538:     def _dataclass_to_dict(value: Any) -> Any:
2539:         """Convert a dataclass to a dictionary if applicable."""
2540:         from dataclasses import asdict, is_dataclass
2541:
2542:         if is_dataclass(value):
2543:             return asdict(value)
2544:         return value
2545:
2546:     @classmethod
2547:     def ensure(
2548:         cls,
2549:         document: Any,
2550:         *,
2551:         document_id: str | None = None,
2552:         use_spc_ingestion: bool = True,
2553:     ) -> PreprocessedDocument:
2554:         """Normalize arbitrary ingestion payloads into orchestrator documents.
2555:
2556:         Args:
2557:             document: Document to normalize (PreprocessedDocument or CanonPolicyPackage)
2558:             document_id: Optional document ID override
2559:             use_spc_ingestion: Must be True (SPC is now the only supported ingestion method)
2560:
2561:         Returns:
2562:             PreprocessedDocument instance
2563:
2564:         Raises:
2565:             ValueError: If use_spc_ingestion is False
2566:             TypeError: If document type is not supported
2567:         """
2568:         import logging
2569:
2570:         logger = logging.getLogger(__name__)
2571:
2572:         if not use_spc_ingestion:
2573:             raise ValueError(
2574:                 "SPC ingestion is now required. Set use_spc_ingestion=True or remove the parameter. "
```

```
2575:             "Legacy ingestion methods (document_ingestion module) are no longer supported."
2576:         )
2577:
2578:     if isinstance(document, type):
2579:         class_name = getattr(document, "__name__", str(document))
2580:         raise TypeError(
2581:             f"Expected document instance, got class type '{class_name}'. "
2582:             "Pass an instance of the document, not the class itself."
2583:         )
2584:
2585:     if isinstance(document, cls):
2586:         return document
2587:
2588:     if hasattr(document, "chunk_graph"):
2589:         chunk_graph = getattr(document, "chunk_graph", None)
2590:         if chunk_graph is None:
2591:             raise ValueError(
2592:                 "Document has chunk_graph attribute but it is None. "
2593:                 "Ensure SPC ingestion pipeline completed successfully."
2594:             )
2595:
2596:     if not hasattr(chunk_graph, "chunks") or not chunk_graph.chunks:
2597:         raise ValueError(
2598:             "Document chunk_graph is empty. "
2599:             "Ensure SPC ingestion pipeline completed successfully and extracted chunks."
2600:         )
2601:
2602:     try:
2603:         from farfan_pipeline.utils.spc_adapter import SPCAdapter
2604:
2605:         adapter = SPCAdapter()
2606:         preprocessed = adapter.to_preprocessed_document(
2607:             document, document_id=document_id
2608:         )
2609:
2610:         validation_results = []
2611:
2612:         if not preprocessed.raw_text or not preprocessed.raw_text.strip():
2613:             raise ValueError(
2614:                 "SPC ingestion produced empty document. "
2615:                 "Check that the source document contains extractable text."
2616:             )
2617:         text_length = len(preprocessed.raw_text)
2618:         validation_results.append(f"raw_text: {text_length} chars")
2619:
2620:         sentence_count = (
2621:             len(preprocessed.sentences) if preprocessed.sentences else 0
2622:         )
2623:         if sentence_count == 0:
2624:             logger.warning(
2625:                 "SPC ingestion produced zero sentences - document may be malformed"
2626:             )
2627:             validation_results.append(f"sentences: {sentence_count}")
2628:
2629:         chunk_count = preprocessed.metadata.get("chunk_count", 0)
2630:         validation_results.append(f"chunks: {chunk_count}")
```

```
2631:  
2632:         logger.info(  
2633:             f"SPC ingestion validation passed: {', '.join(validation_results)}"  
2634:         )  
2635:  
2636:     return preprocessed  
2637: except ImportError as e:  
2638:     raise ImportError(  
2639:         "SPC ingestion requires spc_adapter module. "  
2640:         "Ensure farfan_core.utils.spc_adapter is available."  
2641:     ) from e  
2642: except ValueError:  
2643:     raise  
2644: except Exception as e:  
2645:     raise TypeError(  
2646:         f"Failed to adapt SPC document: {e}. "  
2647:         "Ensure document is a valid CanonPolicyPackage instance from SPC pipeline."  
2648:     ) from e  
2649:  
2650:     raise TypeError(  
2651:         "Unsupported preprocessed document payload. "  
2652:         f"Expected PreprocessedDocument or CanonPolicyPackage with chunk_graph, got {type(document)}!r}. "  
2653:         "Documents must be processed through the SPC ingestion pipeline first."  
2654:     )  
2655:  
2656:  
2657:  
2658: =====  
2659: FILE: src/farfan_pipeline/core/wiring/__init__.py  
2660: =====  
2661:  
2662: """Wiring System - Fine-Grained Module Connection and Contract Validation.  
2663:  
2664: This package implements the complete wiring architecture for SAAAAAA,  
2665: providing deterministic initialization, contract validation, and observability  
2666: for all module connections.  
2667:  
2668: Architecture:  
2669: - Ports and adapters (hexagonal architecture)  
2670: - Dependency injection via constructors  
2671: - Feature flags for conditional wiring  
2672: - Contract validation between all links  
2673: - OpenTelemetry observability  
2674: - Deterministic initialization order  
2675:  
2676: Key Modules:  
2677: - errors: Typed error classes for wiring failures  
2678: - contracts: Pydantic models for link contracts  
2679: - feature_flags: Typed feature flags  
2680: - bootstrap: Deterministic initialization engine  
2681: - validation: Contract validation between links  
2682: - observability: Tracing and metrics  
2683: - analysis_factory: Factory for analysis module dependency injection  
2684: """  
2685:  
2686: from farfan_pipeline.core.wiring.analysis_factory import (
```

```
2687:     create_analysis_components,
2688:     create_bayesian_confidence_calculator,
2689:     create_contradiction_detector,
2690:     create_document_loader,
2691:     create_municipal_analyzer,
2692:     create_municipal_ontology,
2693:     create_performance_analyzer,
2694:     create_semantic_analyzer,
2695:     create_temporal_logic_verifier,
2696: )
2697:
2698: __all__ = [
2699:     'create_analysis_components',
2700:     'create_bayesian_confidence_calculator',
2701:     'create_contradiction_detector',
2702:     'create_document_loader',
2703:     'create_municipal_analyzer',
2704:     'create_municipal_ontology',
2705:     'create_performance_analyzer',
2706:     'create_semantic_analyzer',
2707:     'create_temporal_logic_verifier',
2708: ]
2709:
2710:
2711:
2712: =====
2713: FILE: src/farfan_pipeline/core/wiring/analysis_factory.py
2714: =====
2715:
2716:
2717:
2718:
2719:
2720:
2721: =====
2722: FILE: src/farfan_pipeline/core/wiring/bootstrap.py
2723: =====
2724:
2725: """Bootstrap module for deterministic wiring initialization.
2726:
2727: Implements the complete initialization sequence with:
2728: 1. Resource loading (QuestionnaireResourceProvider)
2729: 2. Signal system setup (memory:// by default, HTTP optional)
2730: 3. CoreModuleFactory with DI
2731: 4. ArgRouterExtended (â\211Y30 routes)
2732: 5. Orchestrator assembly
2733:
2734: All initialization is deterministic and observable.
2735: """
2736:
2737: from __future__ import annotations
2738:
2739: import json
2740: import time
2741: from collections import OrderedDict
2742: from dataclasses import dataclass, field
```

```
2743: from pathlib import Path
2744: from typing import Any
2745:
2746: import structlog
2747:
2748: from farfan_pipeline.config.paths import CONFIG_DIR, DATA_DIR
2749: from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
2750: from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
2751: from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
2752: from farfan_pipeline.core.orchestrator.factory import CoreModuleFactory
2753: from farfan_pipeline.core.orchestrator.signals import (
2754:     InMemorySignalSource,
2755:     SignalClient,
2756:     SignalPack,
2757:     SignalRegistry,
2758: )
2759:
2760: @dataclass
2761: class QuestionnaireResourceProvider:
2762:     """Provider for questionnaire resources."""
2763:     questionnaire_path: Path | None = None
2764:     data_dir: Path = field(default_factory=lambda: DATA_DIR)
2765:
2766:     try: # Optional dependency: calibration orchestrator
2767:         from farfan_pipeline.core.calibration.orchestrator import CalibrationOrchestrator as _CalibrationOrchestrator
2768:         from farfan_pipeline.core.calibration.config import DEFAULT_CALIBRATION_CONFIG as _DEFAULT_CALIBRATION_CONFIG
2769:         _HAS_CALIBRATION = True
2770:     except Exception: # pragma: no cover - only during stripped installs
2771:         _CalibrationOrchestrator = None # type: ignore[assignment]
2772:         _DEFAULT_CALIBRATION_CONFIG = None # type: ignore[assignment]
2773:         _HAS_CALIBRATION = False
2774:
2775:     from farfan_pipeline.core.wiring.errors import MissingDependencyError, WiringInitializationError
2776:     from farfan_pipeline.core.wiring.feature_flags import WiringFeatureFlags
2777:     from farfan_pipeline.core.wiring.phase_0_validator import Phase0Validator
2778:     from farfan_pipeline.core.wiring.validation import WiringValidator
2779:
2780:     logger = structlog.get_logger(__name__)
2781:
2782:
2783: @dataclass
2784: class WiringComponents:
2785:     """Container for all wired components.
2786:
2787:     Attributes:
2788:         provider: QuestionnaireResourceProvider
2789:         signal_client: SignalClient (memory:// or HTTP)
2790:         signal_registry: SignalRegistry with TTL and LRU
2791:         executor_config: ExecutorConfig with defaults
2792:         factory: CoreModuleFactory with DI
2793:         arg_router: ExtendedArgRouter with special routes
2794:         class_registry: Class registry for routing
2795:         validator: WiringValidator for contract checking
2796:         flags: Feature flags used during initialization
2797:         init_hashes: Hashes computed during initialization
2798:     """
2799:
```

```
2799:  
2800:     provider: QuestionnaireResourceProvider  
2801:     signal_client: SignalClient  
2802:     signal_registry: SignalRegistry  
2803:     executor_config: ExecutorConfig  
2804:     factory: CoreModuleFactory  
2805:     arg_router: ExtendedArgRouter  
2806:     class_registry: dict[str, type]  
2807:     validator: WiringValidator  
2808:     flags: WiringFeatureFlags  
2809:     calibration_orchestrator: "_CalibrationOrchestrator | None" = None  
2810:     init_hashes: dict[str, str] = field(default_factory=dict)  
2811:  
2812:  
2813: CANONICAL_POLICY_AREA_DEFINITIONS: "OrderedDict[str, dict[str, list[str] | str]]" = OrderedDict(  
2814:     [  
2815:         (  
2816:             "PA01",  
2817:             {  
2818:                 "name": "Derechos de las mujeres e igualdad de gÃ©nero",  
2819:                 "slug": "genero_mujeres",  
2820:                 "aliases": ["fiscal"],  
2821:             },  
2822:         ),  
2823:         (  
2824:             "PA02",  
2825:             {  
2826:                 "name": "PrevenciÃ³n de la violencia y protecciÃ³n",  
2827:                 "slug": "seguridad_violencia",  
2828:                 "aliases": ["salud"],  
2829:             },  
2830:         ),  
2831:         (  
2832:             "PA03",  
2833:             {  
2834:                 "name": "Ambiente sano y cambio climÃ¡tico",  
2835:                 "slug": "ambiente",  
2836:                 "aliases": ["ambiente"],  
2837:             },  
2838:         ),  
2839:         (  
2840:             "PA04",  
2841:             {  
2842:                 "name": "Derechos econÃ³micos, sociales y culturales",  
2843:                 "slug": "derechos_sociales",  
2844:                 "aliases": ["energÃ¡a"],  
2845:             },  
2846:         ),  
2847:         (  
2848:             "PA05",  
2849:             {  
2850:                 "name": "Derechos de las vÃctimas y construcciÃ³n de paz",  
2851:                 "slug": "paz_victimas",  
2852:                 "aliases": ["transporte"],  
2853:             },  
2854:         ),  
2855:
```

```
2855:     (
2856:         "PA06",
2857:         {
2858:             "name": "Derecho al futuro de la niñez y juventud",
2859:             "slug": "ninez_juventud",
2860:             "aliases": [],
2861:         },
2862:     ),
2863:     (
2864:         "PA07",
2865:         {
2866:             "name": "Tierras y territorios",
2867:             "slug": "tierras_territorios",
2868:             "aliases": [],
2869:         },
2870:     ),
2871:     (
2872:         "PA08",
2873:         {
2874:             "name": "LÃ-deres, lideresas y defensores de DD. HH.",
2875:             "slug": "liderazgos_ddhh",
2876:             "aliases": [],
2877:         },
2878:     ),
2879:     (
2880:         "PA09",
2881:         {
2882:             "name": "Derechos de personas privadas de libertad",
2883:             "slug": "privados_libertad",
2884:             "aliases": [],
2885:         },
2886:     ),
2887:     (
2888:         "PA10",
2889:         {
2890:             "name": "MigraciÃ³n transfronteriza",
2891:             "slug": "migracion",
2892:             "aliases": [],
2893:         },
2894:     ),
2895:   ]
2896: )
2897:
2898: SIGNAL_PACK_VERSION = "1.0.0"
2899: MAX_PATTERNS_PER_POLICY_AREA = 32
2900:
2901: class WiringBootstrap:
2902:     """Bootstrap engine for deterministic wiring initialization.
2903:
2904:     Follows strict initialization order:
2905:     1. Load resources (questionnaire)
2906:     2. Build signal system (memory:// or HTTP)
2907:     3. Create factory with DI
2908:     4. Initialize arg router
2909:     5. Validate all contracts
2910:     """

```

```
2911:  
2912:     def __init__(  
2913:         self,  
2914:         questionnaire_path: str | Path,  
2915:         questionnaire_hash: str,  
2916:         executor_config_path: str | Path,  
2917:         calibration_profile: str,  
2918:         abort_on_insufficient: bool,  
2919:         resource_limits: dict[str, int],  
2920:         flags: WiringFeatureFlags | None = None,  
2921:     ) -> None:  
2922:         """Initialize bootstrap engine.  
2923:  
2924:     Args:  
2925:         questionnaire_path: Path to questionnaire monolith JSON.  
2926:         questionnaire_hash: Expected SHA-256 hash of the monolith.  
2927:         executor_config_path: Path to the executor configuration.  
2928:         calibration_profile: The calibration profile to use.  
2929:         abort_on_insufficient: Flag to abort on insufficient data.  
2930:         resource_limits: Resource limit settings.  
2931:         flags: Feature flags (defaults to environment).  
2932:     """  
2933:         self.questionnaire_path = questionnaire_path  
2934:         self.questionnaire_hash = questionnaire_hash  
2935:         self.executor_config_path = executor_config_path  
2936:         self.calibration_profile = calibration_profile  
2937:         self.abort_on_insufficient = abort_on_insufficient  
2938:         self.resource_limits = resource_limits  
2939:         self.flags = flags or WiringFeatureFlags.from_env()  
2940:         self._start_time = time.time()  
2941:  
2942:         # Validate flags  
2943:         warnings = self.flags.validate()  
2944:         for warning in warnings:  
2945:             logger.warning("feature_flag_warning", message=warning)  
2946:  
2947:         logger.info(  
2948:             "wiring_bootstrap_initialized",  
2949:             questionnaire_path=str(questionnaire_path) if questionnaire_path else None,  
2950:             flags=self.flags.to_dict(),  
2951:         )  
2952:  
2953:     def bootstrap(self) -> WiringComponents:  
2954:         """Execute complete bootstrap sequence.  
2955:  
2956:         Returns:  
2957:             WiringComponents with all initialized modules  
2958:  
2959:         Raises:  
2960:             WiringInitializationError: If any phase fails  
2961:         """  
2962:         logger.info("wiring_bootstrap_start")  
2963:  
2964:         try:  
2965:             # Phase 0: Validate configuration contract  
2966:             logger.info("wiring_init_phase", phase="phase_0_validation")
```

```
2967:         phase_0_validator = Phase0Validator()
2968:         raw_config = {
2969:             "monolith_path": self.questionnaire_path,
2970:             "questionnaire_hash": self.questionnaire_hash,
2971:             "executor_config_path": self.executor_config_path,
2972:             "calibration_profile": self.calibration_profile,
2973:             "abort_on_insufficient": self.abort_on_insufficient,
2974:             "resource_limits": self.resource_limits,
2975:         }
2976:         phase_0_validator.validate(raw_config)
2977:         logger.info("phase_0_validation_passed")
2978:
2979:     # Phase 1: Load resources
2980:     provider = self._load_resources()
2981:
2982:     # Phase 2: Build signal system
2983:     signal_client, signal_registry = self._build_signal_system(provider)
2984:
2985:     # Phase 3: Create executor config
2986:     executor_config = self._create_executor_config()
2987:
2988:     # Phase 4: Create factory with DI
2989:     factory = self._create_factory(provider, signal_registry, executor_config)
2990:
2991:     # Phase 5: Build class registry
2992:     class_registry = self._build_class_registry()
2993:
2994:     # Phase 6: Initialize arg router
2995:     arg_router = self._create_arg_router(class_registry)
2996:
2997:     # Phase 7: Create validator
2998:     validator = WiringValidator()
2999:
3000:     # Phase 8: Create calibration orchestrator (optional enhancement)
3001:     calibration_orchestrator = self._create_calibration_orchestrator()
3002:
3003:     # Phase 9: Seed signals (if memory mode)
3004:     if signal_client._transport == "memory":
3005:         metrics = self._seed_canonical_policy_area_signals(
3006:             signal_client._memory_source,
3007:             signal_registry,
3008:             provider,
3009:         )
3010:         logger.info(
3011:             "signals_seeded",
3012:             areas=metrics["canonical_areas"],
3013:             aliases=metrics["legacy_aliases"],
3014:             hit_rate=metrics["hit_rate"],
3015:         )
3016:
3017:     # Compute initialization hashes
3018:     init_hashes = self._compute_init_hashes(
3019:         provider, signal_registry, factory, arg_router
3020:     )
3021:
3022:     components = WiringComponents(
```

```
3023:         provider=provider,
3024:         signal_client=signal_client,
3025:         signal_registry=signal_registry,
3026:         executor_config=executor_config,
3027:         factory=factory,
3028:         arg_router=arg_router,
3029:         class_registry=class_registry,
3030:         validator=validator,
3031:         calibration_orchestrator=calibration_orchestrator,
3032:         flags=self.flags,
3033:         init_hashes=init_hashes,
3034:     )
3035:
3036:     elapsed = time.time() - self._start_time
3037:
3038:     logger.info(
3039:         "wiring_bootstrap_complete",
3040:         elapsed_s=elapsed,
3041:         factory_instances=19, # Expected count
3042:         argrouter_routes=arg_router.get_special_route_coverage(),
3043:         signals_mode=signal_client._transport,
3044:         init_hashes={k: v[:16] for k, v in init_hashes.items()},
3045:     )
3046:
3047:     return components
3048:
3049: except Exception as e:
3050:     elapsed = time.time() - self._start_time
3051:     logger.error(
3052:         "wiring_bootstrap_failed",
3053:         elapsed_s=elapsed,
3054:         error=str(e),
3055:         error_type=type(e).__name__,
3056:     )
3057:     raise
3058:
3059: def _load_resources(self) -> QuestionnaireResourceProvider:
3060:     """Load questionnaire resources.
3061:
3062:     Returns:
3063:         QuestionnaireResourceProvider instance
3064:
3065:     Raises:
3066:         WiringInitializationError: If loading fails
3067:     """
3068:     logger.info("wiring_init_phase", phase="load_resources")
3069:
3070:     try:
3071:         if self.questionnaire_path:
3072:             path = Path(self.questionnaire_path)
3073:             if not path.exists():
3074:                 raise MissingDependencyError(
3075:                     dependency=str(path),
3076:                     required_by="WiringBootstrap",
3077:                     fix=f"Ensure questionnaire file exists at {path}",
3078:                 )
3079:
```

```
3079:
3080:             with open(path, encoding="utf-8") as f:
3081:                 data = json.load(f)
3082:
3083:                 provider = QuestionnaireResourceProvider(data)
3084:             else:
3085:                 # Use default/empty provider
3086:                 provider = QuestionnaireResourceProvider({})
3087:
3088:             logger.info(
3089:                 "questionnaire_loaded",
3090:                 path=str(self.questionnaire_path) if self.questionnaire_path else "default",
3091:             )
3092:
3093:             return provider
3094:
3095:         except Exception as e:
3096:             raise WiringInitializationError(
3097:                 phase="load_resources",
3098:                 component="QuestionnaireResourceProvider",
3099:                 reason=str(e),
3100:             ) from e
3101:
3102:     def _build_signal_system(
3103:         self,
3104:         provider: QuestionnaireResourceProvider,
3105:     ) -> tuple[SignalClient, SignalRegistry]:
3106:         """Build signal system (memory:// or HTTP).
3107:
3108:         Args:
3109:             provider: QuestionnaireResourceProvider for signal data
3110:
3111:         Returns:
3112:             Tuple of (SignalClient, SignalRegistry)
3113:
3114:         Raises:
3115:             WiringInitializationError: If setup fails
3116:
3117:         logger.info("wiring_init_phase", phase="build_signal_system")
3118:
3119:     try:
3120:         # Create registry first
3121:         registry = SignalRegistry(
3122:             max_size=100,
3123:             default_ttl_s=3600,
3124:         )
3125:
3126:         # Create signal source
3127:         if self.flags.enable_http_signals:
3128:             # HTTP mode (requires explicit configuration)
3129:             base_url = "http://127.0.0.1:8000" # Default, should be configurable
3130:             logger.info("signal_client_http_mode", base_url=base_url)
3131:
3132:             client = SignalClient(
3133:                 base_url=base_url,
3134:                 enable_http_signals=True,
```

```
3135:         )
3136:     else:
3137:         # Memory mode (default)
3138:         memory_source = InMemorySignalSource()
3139:
3140:         client = SignalClient(
3141:             base_url="memory://",
3142:             enable_http_signals=False,
3143:             memory_source=memory_source,
3144:         )
3145:
3146:         logger.info("signal_client_memory_mode")
3147:
3148:     return client, registry
3149:
3150: except Exception as e:
3151:     raise WiringInitializationError(
3152:         phase="build_signal_system",
3153:         component="SignalClient/SignalRegistry",
3154:         reason=str(e),
3155:     ) from e
3156:
3157: def _create_executor_config(self) -> ExecutorConfig:
3158:     """Create executor configuration.
3159:
3160:     Returns:
3161:         ExecutorConfig with defaults
3162:     """
3163:     logger.info("wiring_init_phase", phase="create_executor_config")
3164:
3165:     config = ExecutorConfig(
3166:         max_tokens=2048,
3167:         temperature=0.0, # Deterministic
3168:         timeout_s=30.0,
3169:         retry=2,
3170:         seed=0 if self.flags.deterministic_mode else None,
3171:     )
3172:
3173:     logger.info(
3174:         "executor_config_created",
3175:         deterministic=self.flags.deterministic_mode,
3176:         seed=config.seed,
3177:     )
3178:
3179:     return config
3180:
3181: def _create_factory(
3182:     self,
3183:     provider: QuestionnaireResourceProvider,
3184:     registry: SignalRegistry,
3185:     config: ExecutorConfig,
3186: ) -> CoreModuleFactory:
3187:     """Create CoreModuleFactory with DI.
3188:
3189:     Args:
3190:         provider: QuestionnaireResourceProvider
```

```
3191:             registry: SignalRegistry for injection
3192:             config: ExecutorConfig for injection
3193:
3194:         Returns:
3195:             CoreModuleFactory instance
3196:
3197:         Raises:
3198:             WiringInitializationError: If creation fails
3199:             """
3200:             logger.info("wiring_init_phase", phase="create_factory")
3201:
3202:         try:
3203:             factory = CoreModuleFactory(
3204:                 data_dir=provider.data_dir,
3205:             )
3206:
3207:             logger.info(
3208:                 "factory_created",
3209:                 data_dir=str(provider.data_dir),
3210:             )
3211:
3212:             return factory
3213:
3214:         except Exception as e:
3215:             raise WiringInitializationError(
3216:                 phase="create_factory",
3217:                 component="CoreModuleFactory",
3218:                 reason=str(e),
3219:             ) from e
3220:
3221:     def _build_class_registry(self) -> dict[str, type]:
3222:         """Build class registry for arg router.
3223:
3224:         Returns:
3225:             Class registry mapping names to types
3226:
3227:         Raises:
3228:             WiringInitializationError: If build fails
3229:             """
3230:             logger.info("wiring_init_phase", phase="build_class_registry")
3231:
3232:         try:
3233:             registry = build_class_registry()
3234:
3235:             logger.info(
3236:                 "class_registry_built",
3237:                 class_count=len(registry),
3238:             )
3239:
3240:             return registry
3241:
3242:         except Exception as e:
3243:             raise WiringInitializationError(
3244:                 phase="build_class_registry",
3245:                 component="ClassRegistry",
3246:                 reason=str(e),
```

```
3247:             ) from e
3248:
3249:     def _create_arg_router(
3250:         self,
3251:         class_registry: dict[str, type],
3252:     ) -> ExtendedArgRouter:
3253:         """Create ExtendedArgRouter with special routes.
3254:
3255:         Args:
3256:             class_registry: Class registry for routing
3257:
3258:         Returns:
3259:             ExtendedArgRouter instance
3260:
3261:         Raises:
3262:             WiringInitializationError: If creation fails
3263:         """
3264:         logger.info("wiring_init_phase", phase="create_arg_router")
3265:
3266:         try:
3267:             router = ExtendedArgRouter(class_registry)
3268:
3269:             route_count = router.get_special_route_coverage()
3270:
3271:             if route_count < 30:
3272:                 logger.warning(
3273:                     "argrouter_coverage_low",
3274:                     count=route_count,
3275:                     expected=30,
3276:                 )
3277:
3278:             logger.info(
3279:                 "arg_router_created",
3280:                 special_routes=route_count,
3281:             )
3282:
3283:         return router
3284:
3285:     except Exception as e:
3286:         raise WiringInitializationError(
3287:             phase="create_arg_router",
3288:             component="ExtendedArgRouter",
3289:             reason=str(e),
3290:         ) from e
3291:
3292:     def _create_calibration_orchestrator(self) -> "_CalibrationOrchestrator | None":
3293:         """
3294:             Create CalibrationOrchestrator when calibration stack is available.
3295:
3296:             Returns:
3297:                 CalibrationOrchestrator instance or None if unavailable.
3298:         """
3299:         if not _HAS_CALIBRATION or _CalibrationOrchestrator is None or _DEFAULT_CALIBRATION_CONFIG is None:
3300:             logger.info("calibration_system_unavailable")
3301:             return None
3302:
```

```
3303:     data_dir = DATA_DIR
3304:     config_dir = CONFIG_DIR
3305:
3306:     kwargs: dict[str, Any] = {"config": _DEFAULT_CALIBRATION_CONFIG}
3307:
3308:     intrinsic_path = config_dir / "intrinsic_calibration.json"
3309:     if intrinsic_path.exists():
3310:         kwargs["intrinsic_calibration_path"] = intrinsic_path
3311:
3312:     compatibility_path = data_dir / "method_compatibility.json"
3313:     if compatibility_path.exists():
3314:         kwargs["compatibility_path"] = compatibility_path
3315:
3316:     registry_path = data_dir / "method_registry.json"
3317:     if registry_path.exists():
3318:         kwargs["method_registry_path"] = registry_path
3319:
3320:     signatures_path = data_dir / "method_signatures.json"
3321:     if signatures_path.exists():
3322:         kwargs["method_signatures_path"] = signatures_path
3323:
3324:     try:
3325:         orchestrator = _CalibrationOrchestrator(**kwargs)
3326:         logger.info(
3327:             "calibration_orchestrator_ready",
3328:             intrinsic=str(intrinsic_path),
3329:             compatibility=str(compatibility_path),
3330:         )
3331:         return orchestrator
3332:     except Exception as exc: # pragma: no cover - defensive guardrail
3333:         logger.warning(
3334:             "calibration_orchestrator_initialization_failed",
3335:             error=str(exc),
3336:         )
3337:     return None
3338:
3339: def _build_signal_pack(
3340:     self,
3341:     provider: QuestionnaireResourceProvider,
3342:     canonical_id: str,
3343:     meta: dict[str, Any],
3344:     *,
3345:     alias: str | None = None,
3346: ) -> SignalPack:
3347:     """Build a SignalPack for a canonical policy area (and optional alias)."""
3348:     pattern_source = getattr(provider, "get_patterns_for_area", None)
3349:     patterns = pattern_source(canonical_id, MAX_PATTERNS_PER_POLICY_AREA) if callable(pattern_source) else []
3350:
3351:     pack = SignalPack(
3352:         version=SIGNAL_PACK_VERSION,
3353:         policy_area=alias or canonical_id, # type: ignore[arg-type]
3354:         patterns=patterns,
3355:         metadata={
3356:             "canonical_id": canonical_id,
3357:             "display_name": meta["name"],
3358:             "slug": meta["slug"],
```

```
3359:         "alias": alias,
3360:     },
3361: )
3362:     fingerprint = pack.compute_hash()
3363:     return pack.model_copy(update={"source_fingerprint": fingerprint})
3364:
3365:     @staticmethod
3366:     def _register_signal_pack(
3367:         memory_source: InMemorySignalSource,
3368:         registry: SignalRegistry,
3369:         pack: SignalPack,
3370:     ) -> None:
3371:         """Register pack in both memory source and registry."""
3372:         memory_source.register(pack.policy_area, pack)
3373:         registry.put(pack.policy_area, pack)
3374:         logger.debug(
3375:             "signal_seeded",
3376:             policy_area=pack.policy_area,
3377:             canonical_id=pack.metadata.get("canonical_id"),
3378:             patterns=len(pack.patterns),
3379:         )
3380:
3381:     def _seed_canonical_policy_area_signals(
3382:         self,
3383:         memory_source: InMemorySignalSource,
3384:         registry: SignalRegistry,
3385:         provider: QuestionnaireResourceProvider,
3386:     ) -> dict[str, Any]:
3387:         """
3388:             Seed signal registry with canonical (PA01-PA10) policy areas.
3389:
3390:             Returns:
3391:                 Metrics dict with coverage and legacy alias info.
3392:
3393:         canonical_count = 0
3394:         alias_count = 0
3395:
3396:         for area_id, meta in CANONICAL_POLICY_AREA_DEFINITIONS.items():
3397:             pack = self._build_signal_pack(provider, area_id, meta)
3398:             self._register_signal_pack(memory_source, registry, pack)
3399:             canonical_count += 1
3400:
3401:             for alias in meta["aliases"]:
3402:                 # type: ignore[index]
3403:                 alias_pack = self._build_signal_pack(
3404:                     provider,
3405:                     area_id,
3406:                     meta,
3407:                     alias=alias,
3408:                 )
3409:                 self._register_signal_pack(memory_source, registry, alias_pack)
3410:                 alias_count += 1
3411:
3412:             hits = sum(
3413:                 1
3414:                 for area_id in CANONICAL_POLICY_AREA_DEFINITIONS
3415:                 if registry.get(area_id) is not None
3416:             )
3417:             if hits > 0:
3418:                 logger.info(f"Canonical policy area {area_id} registered successfully.")
3419:             else:
3420:                 logger.warning(f"Canonical policy area {area_id} registration failed.")
3421:
3422:         logger.info(f"Canonical policy areas registered: {canonical_count} canonical, {alias_count} aliases.")
```

```
3415:         )
3416:         total_required = len(CANONICAL_POLICY_AREA_DEFINITIONS)
3417:         hit_rate = hits / total_required if total_required else 0.0
3418:
3419:         return {
3420:             "canonical_areas": canonical_count,
3421:             "legacy_aliases": alias_count,
3422:             "hit_rate": hit_rate,
3423:             "required_hit_rate": 0.95,
3424:         }
3425:
3426:     def seed_signals_public(
3427:         self,
3428:         client: SignalClient,
3429:         registry: SignalRegistry,
3430:         provider: QuestionnaireResourceProvider,
3431:     ) -> dict[str, Any]:
3432:         """Seed initial signals in memory mode (PUBLIC API).
3433:
3434:         This replaces the private _seed_signals method with a public API that:
3435:         1. Validates the SignalClient is using memory transport
3436:         2. Returns deterministic metrics for validation
3437:         3. Enforces the 95% hit rate requirement
3438:
3439:         Args:
3440:             client: SignalClient to seed (must be in memory mode)
3441:             registry: SignalRegistry to populate
3442:             provider: QuestionnaireResourceProvider for patterns
3443:
3444:         Returns:
3445:             Dict with seeding metrics (areas_seeded, total_signals, hit_rate)
3446:
3447:         Raises:
3448:             ValueError: If client is not in memory mode
3449:             WiringInitializationError: If hit rate requirement is not met
3450:         """
3451:         logger.info("wiring_init_phase", phase="seed_signals_public")
3452:
3453:         if getattr(client, "_transport", None) != "memory":
3454:             raise ValueError(
3455:                 "Signal seeding requires memory mode. "
3456:                 "Set enable_http_signals=False in WiringFeatureFlags."
3457:             )
3458:
3459:         memory_source = getattr(client, "_memory_source", None)
3460:         if memory_source is None:
3461:             raise ValueError("Signal client memory source not initialized.")
3462:
3463:         metrics = self._seed_canonical_policy_area_signals(
3464:             memory_source,
3465:             registry,
3466:             provider,
3467:         )
3468:
3469:         if metrics["hit_rate"] < metrics["required_hit_rate"]:
3470:             raise WiringInitializationError(
```

```
3471:             phase="seed_signals",
3472:             component="SignalRegistry",
3473:             reason=(
3474:                 f"Signal hit rate {metrics['hit_rate']:.2%} below "
3475:                 f"required threshold {metrics['required_hit_rate']:.2%}"
3476:             ),
3477:         )
3478:
3479:     return metrics
3480:
3481:
3482:
3483:     def _compute_init_hashes(
3484:         self,
3485:         provider: QuestionnaireResourceProvider,
3486:         registry: SignalRegistry,
3487:         factory: CoreModuleFactory,
3488:         router: ExtendedArgRouter,
3489:     ) -> dict[str, str]:
3490:         """Compute hashes for initialized components.
3491:
3492:         Args:
3493:             provider: QuestionnaireResourceProvider
3494:             registry: SignalRegistry
3495:             factory: CoreModuleFactory
3496:             router: ExtendedArgRouter
3497:
3498:         Returns:
3499:             Dict of component names to their hashes
3500:         """
3501:         import blake3
3502:
3503:         hashes = {}
3504:
3505:         # Provider hash (based on data keys)
3506:         provider_keys = sorted(provider._data.keys()) if hasattr(provider, '_data') else []
3507:         hashes["provider"] = blake3.blake3(
3508:             json.dumps(provider_keys, sort_keys=True).encode('utf-8')
3509:         ).hexdigest()
3510:
3511:         # Registry hash (based on metrics)
3512:         registry_metrics = registry.get_metrics()
3513:         hashes["registry"] = blake3.blake3(
3514:             json.dumps(registry_metrics, sort_keys=True).encode('utf-8')
3515:         ).hexdigest()
3516:
3517:         # Router hash (based on special routes count)
3518:         router_data = {"route_count": router.get_special_route_coverage()}
3519:         hashes["router"] = blake3.blake3(
3520:             json.dumps(router_data, sort_keys=True).encode('utf-8')
3521:         ).hexdigest()
3522:
3523:     return hashes
3524:
3525:
3526: __all__ = [
```

```
3527:     'WiringComponents',
3528:     'WiringBootstrap',
3529: ]
3530:
3531:
3532:
3533: =====
3534: FILE: src/farfan_pipeline/core/wiring/contracts.py
3535: =====
3536:
3537: """Contract models for wiring validation.
3538:
3539: Defines Pydantic models for each link's deliverable and expectation.
3540: Validation ensures type safety and completeness at every boundary.
3541: """
3542:
3543: from __future__ import annotations
3544:
3545: from typing import Any
3546:
3547: from pydantic import BaseModel, Field, field_validator
3548:
3549:
3550: class CPPDeliverable(BaseModel):
3551:     """Contract for CPP ingestion output (Deliverable).
3552:
3553:     DEPRECATED: Use SPCDeliverable instead. This model is kept for backward compatibility.
3554:
3555:     Note: CPP (Canon Policy Package) is the legacy name for SPC (Smart Policy Chunks).
3556: """
3557:
3558:     chunk_graph: dict[str, Any] = Field(
3559:         description="Chunk graph with all chunks"
3560:     )
3561:     policy_manifest: dict[str, Any] = Field(
3562:         description="Policy metadata manifest"
3563:     )
3564:     provenance_completeness: float = Field(
3565:         ge=0.0,
3566:         le=1.0,
3567:         description="Provenance completeness score (must be 1.0)"
3568:     )
3569:     schema_version: str = Field(
3570:         description="CPP schema version"
3571:     )
3572:
3573:     model_config = {
3574:         "frozen": True,
3575:         "extra": "forbid",
3576:     }
3577:
3578:     def __init__(self, **data: Any) -> None:
3579:         import warnings
3580:         warnings.warn(
3581:             "CPPDeliverable is deprecated. Use SPCDeliverable instead.",
3582:             DeprecationWarning,
```

```
3583:         stacklevel=2
3584:     )
3585:     super().__init__(**data)
3586:
3587:     @field_validator("provenance_completeness")
3588:     @classmethod
3589:     def validate_completeness(cls, v: float) -> float:
3590:         """Ensure provenance is 100% complete."""
3591:         if v != 1.0:
3592:             raise ValueError(
3593:                 f"provenance_completeness must be 1.0, got {v}. "
3594:                 "Ensure ingestion pipeline completed successfully."
3595:             )
3596:         return v
3597:
3598:
3599: class SPCDeliverable(BaseModel):
3600:     """Contract for SPC (Smart Policy Chunks) ingestion output (Deliverable).
3601:
3602:     This is the preferred terminology for new code. SPC is the successor to CPP.
3603:     """
3604:
3605:     chunk_graph: dict[str, Any] = Field(
3606:         description="Chunk graph with all chunks"
3607:     )
3608:     policy_manifest: dict[str, Any] = Field(
3609:         description="Policy metadata manifest"
3610:     )
3611:     provenance_completeness: float = Field(
3612:         ge=0.0,
3613:         le=1.0,
3614:         description="Provenance completeness score (must be 1.0)"
3615:     )
3616:     schema_version: str = Field(
3617:         description="SPC schema version"
3618:     )
3619:
3620:     model_config = {
3621:         "frozen": True,
3622:         "extra": "forbid",
3623:     }
3624:
3625:     @field_validator("provenance_completeness")
3626:     @classmethod
3627:     def validate_completeness(cls, v: float) -> float:
3628:         """Ensure provenance is 100% complete."""
3629:         if v != 1.0:
3630:             raise ValueError(
3631:                 f"provenance_completeness must be 1.0, got {v}. "
3632:                 "Ensure SPC ingestion pipeline completed successfully."
3633:             )
3634:         return v
3635:
3636:
3637: class AdapterExpectation(BaseModel):
3638:     """Contract for CPPAdapter input (Expectation)."""
```

```
3639:  
3640:     chunk_graph: dict[str, Any] = Field(  
3641:         description="Must have chunk_graph with chunks"  
3642:     )  
3643:     policy_manifest: dict[str, Any] = Field(  
3644:         description="Must have policy_manifest"  
3645:     )  
3646:     provenance_completeness: float = Field(  
3647:         ge=1.0,  
3648:         le=1.0,  
3649:         description="Must be exactly 1.0"  
3650:     )  
3651:  
3652:     model_config = {  
3653:         "frozen": True,  
3654:         "extra": "allow", # Allow additional fields  
3655:     }  
3656:  
3657:  
3658: class PreprocessedDocumentDeliverable(BaseModel):  
3659:     """Contract for CPPAdapter output (Deliverable)."""  
3660:  
3661:     sentence_metadata: list[dict[str, Any]] = Field(  
3662:         min_length=1,  
3663:         description="Must have at least one sentence"  
3664:     )  
3665:     resolution_index: dict[str, Any] = Field(  
3666:         description="Resolution index must be consistent"  
3667:     )  
3668:     provenance_completeness: float = Field(  
3669:         ge=1.0,  
3670:         le=1.0,  
3671:         description="Must maintain 1.0 completeness"  
3672:     )  
3673:     document_id: str = Field(  
3674:         min_length=1,  
3675:         description="Document ID must be non-empty"  
3676:     )  
3677:  
3678:     model_config = {  
3679:         "frozen": True,  
3680:         "extra": "forbid",  
3681:     }  
3682:  
3683:  
3684: class OrchestratorExpectation(BaseModel):  
3685:     """Contract for Orchestrator input (Expectation)."""  
3686:  
3687:     sentence_metadata: list[dict[str, Any]] = Field(  
3688:         min_length=1,  
3689:         description="Requires sentence_metadata"  
3690:     )  
3691:     document_id: str = Field(  
3692:         min_length=1,  
3693:         description="Requires document_id"  
3694:     )
```

```
3695:  
3696:     model_config = {  
3697:         "frozen": True,  
3698:         "extra": "allow",  
3699:     }  
3700:  
3701:  
3702: class ArgRouterPayloadDeliverable(BaseModel):  
3703:     """Contract for Orchestrator to ArgRouter (Deliverable)."""  
3704:  
3705:     class_name: str = Field(  
3706:         min_length=1,  
3707:         description="Target class name"  
3708:     )  
3709:     method_name: str = Field(  
3710:         min_length=1,  
3711:         description="Target method name"  
3712:     )  
3713:     payload: dict[str, Any] = Field(  
3714:         description="Method arguments payload"  
3715:     )  
3716:  
3717:     model_config = {  
3718:         "frozen": True,  
3719:         "extra": "forbid",  
3720:     }  
3721:  
3722:  
3723: class ArgRouterExpectation(BaseModel):  
3724:     """Contract for ArgRouter input (Expectation)."""  
3725:  
3726:     class_name: str = Field(  
3727:         min_length=1,  
3728:         description="Class must exist in registry"  
3729:     )  
3730:     method_name: str = Field(  
3731:         min_length=1,  
3732:         description="Method must exist on class"  
3733:     )  
3734:     payload: dict[str, Any] = Field(  
3735:         description="Payload with required arguments"  
3736:     )  
3737:  
3738:     model_config = {  
3739:         "frozen": True,  
3740:         "extra": "allow",  
3741:     }  
3742:  
3743:  
3744: class ExecutorInputDeliverable(BaseModel):  
3745:     """Contract for ArgRouter to Executor (Deliverable)."""  
3746:  
3747:     args: tuple[Any, ...] = Field(  
3748:         description="Positional arguments"  
3749:     )  
3750:     kwargs: dict[str, Any] = Field(  
3751:
```

```
3751:         description="Keyword arguments"
3752:     )
3753:     method_signature: str = Field(
3754:         description="Target method signature for validation"
3755:     )
3756:
3757:     model_config = {
3758:         "frozen": True,
3759:         "extra": "forbid",
3760:     }
3761:
3762:
3763: class SignalPackDeliverable(BaseModel):
3764:     """Contract for SignalsClient output (Deliverable)."""
3765:
3766:     version: str = Field(
3767:         description="Signal pack version (must be present)"
3768:     )
3769:     policy_area: str = Field(
3770:         description="Policy area for signals"
3771:     )
3772:     patterns: list[str] = Field(
3773:         default_factory=list,
3774:         description="Text patterns"
3775:     )
3776:     indicators: list[str] = Field(
3777:         default_factory=list,
3778:         description="KPI indicators"
3779:     )
3780:
3781:     model_config = {
3782:         "frozen": True,
3783:         "extra": "allow", # Allow additional signal fields
3784:     }
3785:
3786:     @field_validator("version")
3787:     @classmethod
3788:     def validate_version(cls, v: str) -> str:
3789:         """Validate version format."""
3790:         if not v or v.strip() == "":
3791:             raise ValueError("version must be non-empty")
3792:         return v
3793:
3794:
3795: class SignalRegistryExpectation(BaseModel):
3796:     """Contract for SignalRegistry input (Expectation)."""
3797:
3798:     version: str = Field(
3799:         min_length=1,
3800:         description="Requires version"
3801:     )
3802:     policy_area: str = Field(
3803:         min_length=1,
3804:         description="Requires policy_area"
3805:     )
3806:
```

```
3807:     model_config = {
3808:         "frozen": True,
3809:         "extra": "allow",
3810:     }
3811:
3812:
3813: class EnrichedChunkDeliverable(BaseModel):
3814:     """Contract for Executor output (Deliverable)."""
3815:
3816:     chunk_id: str = Field(
3817:         min_length=1,
3818:         description="Chunk identifier"
3819:     )
3820:     used_signals: list[str] = Field(
3821:         default_factory=list,
3822:         description="Signals used during execution"
3823:     )
3824:     enrichment: dict[str, Any] = Field(
3825:         description="Enrichment data"
3826:     )
3827:
3828:     model_config = {
3829:         "frozen": True,
3830:         "extra": "allow",
3831:     }
3832:
3833:
3834: class AggregateExpectation(BaseModel):
3835:     """Contract for Aggregate input (Expectation)."""
3836:
3837:     enriched_chunks: list[dict[str, Any]] = Field(
3838:         min_length=1,
3839:         description="Must have at least one enriched chunk"
3840:     )
3841:
3842:     model_config = {
3843:         "frozen": True,
3844:         "extra": "allow",
3845:     }
3846:
3847:
3848: class FeatureTableDeliverable(BaseModel):
3849:     """Contract for Aggregate output (Deliverable)."""
3850:
3851:     table_type: str = Field(
3852:         description="Must be 'pyarrow.Table'"
3853:     )
3854:     num_rows: int = Field(
3855:         ge=1,
3856:         description="Must have at least one row"
3857:     )
3858:     column_names: list[str] = Field(
3859:         min_length=1,
3860:         description="Must have required columns"
3861:     )
3862:
```

```
3863:     model_config = {
3864:         "frozen": True,
3865:         "extra": "forbid",
3866:     }
3867:
3868:
3869: class ScoreExpectation(BaseModel):
3870:     """Contract for Score input (Expectation)."""
3871:
3872:     table_type: str = Field(
3873:         description="Must be pa.Table"
3874:     )
3875:     required_columns: list[str] = Field(
3876:         min_length=1,
3877:         description="Required columns for scoring"
3878:     )
3879:
3880:     model_config = {
3881:         "frozen": True,
3882:         "extra": "allow",
3883:     }
3884:
3885:
3886: class ScoresDeliverable(BaseModel):
3887:     """Contract for Score output (Deliverable)."""
3888:
3889:     dataframe_type: str = Field(
3890:         description="Must be 'polars.DataFrame'"
3891:     )
3892:     num_rows: int = Field(
3893:         ge=1,
3894:         description="Must have at least one row"
3895:     )
3896:     metrics_computed: list[str] = Field(
3897:         min_length=1,
3898:         description="Metrics that were computed"
3899:     )
3900:
3901:     model_config = {
3902:         "frozen": True,
3903:         "extra": "forbid",
3904:     }
3905:
3906:
3907: class ReportExpectation(BaseModel):
3908:     """Contract for Report input (Expectation)."""
3909:
3910:     dataframe_type: str = Field(
3911:         description="Must be pl.DataFrame"
3912:     )
3913:     metrics_present: list[str] = Field(
3914:         min_length=1,
3915:         description="Metrics must be present"
3916:     )
3917:     manifest_present: bool = Field(
3918:         description="Manifest must be provided"
```

```
3919:     )
3920:
3921:     model_config = {
3922:         "frozen": True,
3923:         "extra": "allow",
3924:     }
3925:
3926:
3927: class ReportDeliverable(BaseModel):
3928:     """Contract for Report output (Deliverable)."""
3929:
3930:     report_uris: dict[str, str] = Field(
3931:         min_length=1,
3932:         description="Mapping of report name to URI"
3933:     )
3934:     all_reports_generated: bool = Field(
3935:         description="All declared reports generated"
3936:     )
3937:
3938:     model_config = {
3939:         "frozen": True,
3940:         "extra": "forbid",
3941:     }
3942:
3943:
3944: __all__ = [
3945:     'CPPDeliverable',
3946:     'SPCDeliverable',
3947:     'AdapterExpectation',
3948:     'PreprocessedDocumentDeliverable',
3949:     'OrchestratorExpectation',
3950:     'ArgRouterPayloadDeliverable',
3951:     'ArgRouterExpectation',
3952:     'ExecutorInputDeliverable',
3953:     'SignalPackDeliverable',
3954:     'SignalRegistryExpectation',
3955:     'EnrichedChunkDeliverable',
3956:     'AggregateExpectation',
3957:     'FeatureTableDeliverable',
3958:     'ScoreExpectation',
3959:     'ScoresDeliverable',
3960:     'ReportExpectation',
3961:     'ReportDeliverable',
3962: ]
3963:
3964:
3965:
3966: =====
3967: FILE: src/farfan_pipeline/core/wiring/errors.py
3968: =====
3969:
3970: """Typed error classes for wiring system.
3971:
3972: All wiring errors include prescriptive fix information to guide remediation.
3973: Errors are loud and explicit - no silent degradation is permitted.
3974: """
```

```
3975:  
3976: from __future__ import annotations  
3977:  
3978: from typing import Any  
3979:  
3980:  
3981: class WiringError(Exception):  
3982:     """Base class for all wiring errors."""  
3983:  
3984:     def __init__(self, message: str, details: dict[str, Any] | None = None) -> None:  
3985:         super().__init__(message)  
3986:         self.details = details or {}  
3987:  
3988:  
3989: class WiringContractError(WiringError):  
3990:     """Raised when a contract between two links is violated.  
3991:  
3992:     Attributes:  
3993:         link: Name of the violated link (e.g., "cpp->adapter")  
3994:         expected_schema: Expected schema/type  
3995:         received_schema: Actual schema/type received  
3996:         field: Specific field that failed (if applicable)  
3997:         fix: Prescriptive fix instructions  
3998:     """  
3999:  
4000:     def __init__(  
4001:             self,  
4002:             link: str,  
4003:             expected_schema: str,  
4004:             received_schema: str,  
4005:             field: str | None = None,  
4006:             fix: str | None = None,  
4007:         ) -> None:  
4008:         field_info = f" {field: {field}})" if field else ""  
4009:         fix_info = f"\n\nFix: {fix}" if fix else ""  
4010:  
4011:         message = (  
4012:             f"Contract violation in link '{link}'{field_info}\n"  
4013:             f"Expected: {expected_schema}\n"  
4014:             f"Received: {received_schema}"  
4015:             f"{fix_info}"  
4016:         )  
4017:  
4018:         super().__init__(  
4019:             message,  
4020:             details={  
4021:                 "link": link,  
4022:                 "expected_schema": expected_schema,  
4023:                 "received_schema": received_schema,  
4024:                 "field": field,  
4025:                 "fix": fix,  
4026:             }  
4027:         )  
4028:  
4029:  
4030: class MissingDependencyError(WiringError):
```

```
4031:     """Raised when a required dependency is not available.
4032:
4033:     Attributes:
4034:         dependency: Name of missing dependency
4035:         required_by: Module/component that requires it
4036:         fix: How to resolve the missing dependency
4037:     """
4038:
4039:     def __init__(
4040:         self,
4041:         dependency: str,
4042:         required_by: str,
4043:         fix: str | None = None,
4044:     ) -> None:
4045:         fix_info = f"\n\nFix: {fix}" if fix else ""
4046:
4047:         message = (
4048:             f"Missing dependency '{dependency}' required by '{required_by}'"
4049:             f"\n{fix_info}"
4050:
4051:     )
4052:
4053:     super().__init__(
4054:         message,
4055:         details={
4056:             "dependency": dependency,
4057:             "required_by": required_by,
4058:             "fix": fix,
4059:         }
4060:
4061:
4062: class WiringArgumentValidationError(WiringError):
4063:     """Raised when argument routing validation fails.
4064:
4065:     Attributes:
4066:         class_name: Class being routed to
4067:         method_name: Method being called
4068:         issue: Description of validation issue
4069:         provided_args: Arguments that were provided
4070:         expected_args: Arguments that were expected
4071:         fix: How to fix the argument mismatch
4072:     """
4073:
4074:     def __init__(
4075:         self,
4076:         class_name: str,
4077:         method_name: str,
4078:         issue: str,
4079:         provided_args: list[str] | None = None,
4080:         expected_args: list[str] | None = None,
4081:         fix: str | None = None,
4082:     ) -> None:
4083:         fix_info = f"\n\nFix: {fix}" if fix else ""
4084:
4085:         message = (
4086:             f"Argument validation failed for {class_name}.{method_name}\n"
```

```
4087:         f"Issue: {issue}"
4088:     )
4089:
4090:     if provided_args is not None:
4091:         message += f"\nProvided: {' , '.join(provided_args)}"
4092:     if expected_args is not None:
4093:         message += f"\nExpected: {' , '.join(expected_args)}"
4094:
4095:     message += fix_info
4096:
4097:     super().__init__(
4098:         message,
4099:         details={
4100:             "class_name": class_name,
4101:             "method_name": method_name,
4102:             "issue": issue,
4103:             "provided_args": provided_args,
4104:             "expected_args": expected_args,
4105:             "fix": fix,
4106:         }
4107:     )
4108:
4109:
4110: class WiringSignalUnavailableError(WiringError):
4111:     """Raised when required signals are unavailable.
4112:
4113:     Attributes:
4114:         policy_area: Policy area for which signals were requested
4115:         reason: Why signals are unavailable
4116:         breaker_state: Circuit breaker state if applicable
4117: """
4118:
4119:     def __init__(
4120:         self,
4121:         policy_area: str,
4122:         reason: str,
4123:         breaker_state: str | None = None,
4124:     ) -> None:
4125:         breaker_info = f" (breaker: {breaker_state})" if breaker_state else ""
4126:
4127:         message = (
4128:             f"Signals unavailable for policy area '{policy_area}'{breaker_info}\n"
4129:             f"Reason: {reason}"
4130:         )
4131:
4132:         super().__init__(
4133:             message,
4134:             details={
4135:                 "policy_area": policy_area,
4136:                 "reason": reason,
4137:                 "breaker_state": breaker_state,
4138:             }
4139:         )
4140:
4141:
4142: class WiringSignalSchemaError(WiringError):
```

```
4143:     """Raised when signal pack schema is invalid.
4144:
4145:     Attributes:
4146:         pack_version: Signal pack version
4147:         schema_issue: Description of schema problem
4148:         field: Field with schema issue
4149:     """
4150:
4151:     def __init__(
4152:         self,
4153:         pack_version: str,
4154:         schema_issue: str,
4155:         field: str | None = None,
4156:     ) -> None:
4157:         field_info = f" (field: {field})" if field else ""
4158:
4159:         message = (
4160:             f"Invalid signal pack schema{field_info}\n"
4161:             f"Version: {pack_version}\n"
4162:             f"Issue: {schema_issue}"
4163:         )
4164:
4165:         super().__init__(
4166:             message,
4167:             details={
4168:                 "pack_version": pack_version,
4169:                 "schema_issue": schema_issue,
4170:                 "field": field,
4171:             }
4172:         )
4173:
4174:
4175: class WiringInitializationError(WiringError):
4176:     """Raised when wiring initialization fails.
4177:
4178:     Attributes:
4179:         phase: Initialization phase that failed
4180:         component: Component being initialized
4181:         reason: Why initialization failed
4182:     """
4183:
4184:     def __init__(
4185:         self,
4186:         phase: str,
4187:         component: str,
4188:         reason: str,
4189:     ) -> None:
4190:         message = (
4191:             f"Wiring initialization failed in phase '{phase}'\n"
4192:             f"Component: {component}\n"
4193:             f"Reason: {reason}"
4194:         )
4195:
4196:         super().__init__(
4197:             message,
4198:             details={
```

```
4199:         "phase": phase,
4200:         "component": component,
4201:         "reason": reason,
4202:     }
4203: )
4204:
4205:
4206: __all__ = [
4207:     'WiringError',
4208:     'WiringContractError',
4209:     'MissingDependencyError',
4210:     'WiringArgumentValidationException',
4211:     'WiringSignalUnavailableError',
4212:     'WiringSignalSchemaError',
4213:     'WiringInitializationError',
4214: ]
4215:
4216:
4217:
4218: =====
4219: FILE: src/farfan_pipeline/core/wiring/feature_flags.py
4220: =====
4221:
4222: """Feature flags for wiring system configuration.
4223:
4224: All flags are typed and have explicit defaults. Flags control conditional
4225: wiring paths and validation strictness.
4226: """
4227:
4228: from __future__ import annotations
4229:
4230: import os
4231: from dataclasses import dataclass
4232:
4233:
4234: @dataclass(frozen=True)
4235: class WiringFeatureFlags:
4236:     """Feature flags for wiring configuration.
4237:
4238:     Attributes:
4239:         use_spc_ingestion: Use SPC (Smart Policy Chunks) ingestion pipeline - canonical phase-one (default: True)
4240:         enable_http_signals: Enable HTTP signal fetching (default: False)
4241:         allow_threshold_override: Allow runtime threshold overrides (default: False)
4242:         wiring_strict_mode: Enforce strict contract validation (default: True)
4243:         enable_observability: Enable OpenTelemetry tracing (default: True)
4244:         enable_metrics: Enable metrics collection (default: True)
4245:         deterministic_mode: Force deterministic execution (default: True)
4246:     """
4247:
4248:     use_spc_ingestion: bool = True
4249:     # Legacy alias for backwards compatibility
4250:     use_cpp_ingestion: bool = True
4251:     enable_http_signals: bool = False
4252:     allow_threshold_override: bool = False
4253:     wiring_strict_mode: bool = True
4254:     enable_observability: bool = True
```

```
4255:     enable_metrics: bool = True
4256:     deterministic_mode: bool = True
4257:
4258:     @classmethod
4259:     def from_env(cls) -> WiringFeatureFlags:
4260:         """Load feature flags from environment variables.
4261:
4262:             Environment variables:
4263:             - SAAAAAA_USE_SPC_INGESTION: "true" or "false" (canonical phase-one)
4264:             - SAAAAAA_USE_CPP_INGESTION: "true" or "false" (legacy alias)
4265:             - SAAAAAA_ENABLE_HTTP_SIGNALS: "true" or "false"
4266:             - SAAAAAA_ALLOW_THRESHOLD_OVERRIDE: "true" or "false"
4267:             - SAAAAAA_WIRING_STRICT_MODE: "true" or "false"
4268:             - SAAAAAA_ENABLE_OBSERVABILITY: "true" or "false"
4269:             - SAAAAAA_ENABLE_METRICS: "true" or "false"
4270:             - SAAAAAA_DETERMINISTIC_MODE: "true" or "false"
4271:
4272:             Returns:
4273:                 WiringFeatureFlags with values from environment
4274:             """
4275:     def get_bool(key: str, default: bool) -> bool:
4276:         value = os.getenv(key, str(default)).lower()
4277:         return value in ("true", "1", "yes", "on")
4278:
4279:     # Prefer new SPC name, fallback to legacy CPP name
4280:     spc_flag = get_bool("SAAAAAA_USE_SPC_INGESTION",
4281:                         get_bool("SAAAAAA_USE_CPP_INGESTION", True))
4282:
4283:     return cls(
4284:         use_spc_ingestion=spc_flag,
4285:         use_cpp_ingestion=spc_flag, # Keep in sync for backwards compatibility
4286:         enable_http_signals=get_bool("SAAAAAA_ENABLE_HTTP_SIGNALS", False),
4287:         allow_threshold_override=get_bool("SAAAAAA_ALLOW_THRESHOLD_OVERRIDE", False),
4288:         wiring_strict_mode=get_bool("SAAAAAA_WIRING_STRICT_MODE", True),
4289:         enable_observability=get_bool("SAAAAAA_ENABLE_OBSERVABILITY", True),
4290:         enable_metrics=get_bool("SAAAAAA_ENABLE_METRICS", True),
4291:         deterministic_mode=get_bool("SAAAAAA_DETERMINISTIC_MODE", True),
4292:     )
4293:
4294:     def to_dict(self) -> dict[str, bool]:
4295:         """Convert flags to dictionary.
4296:
4297:             Returns:
4298:                 Dictionary of flag names to values
4299:             """
4300:         return {
4301:             "use_spc_ingestion": self.use_spc_ingestion,
4302:             "use_cpp_ingestion": self.use_cpp_ingestion, # Legacy compatibility
4303:             "enable_http_signals": self.enable_http_signals,
4304:             "allow_threshold_override": self.allow_threshold_override,
4305:             "wiring_strict_mode": self.wiring_strict_mode,
4306:             "enable_observability": self.enable_observability,
4307:             "enable_metrics": self.enable_metrics,
4308:             "deterministic_mode": self.deterministic_mode,
4309:         }
4310:
```

```
4311:     def validate(self) -> list[str]:
4312:         """Validate flag combinations for conflicts.
4313:
4314:         Returns:
4315:             List of validation warnings (empty if valid)
4316:
4317:         """
4318:         warnings = []
4319:
4320:         if self.enable_http_signals and self.deterministic_mode:
4321:             warnings.append(
4322:                 "enable_http_signals=True with deterministic_mode=True may cause "
4323:                 "non-determinism due to HTTP variability. Consider using memory:// only."
4324:             )
4325:
4326:         if not self.wiring_strict_mode:
4327:             warnings.append(
4328:                 "wiring_strict_mode=False disables contract validation. "
4329:                 "This is NOT recommended for production."
4330:             )
4331:
4332:         if not self.enable_observability and not self.enable_metrics:
4333:             warnings.append(
4334:                 "Both observability and metrics are disabled. "
4335:                 "Debugging will be difficult without instrumentation."
4336:             )
4337:
4338:         return warnings
4339:
4340: # Default flags instance for convenience
4341: DEFAULT_FLAGS = WiringFeatureFlags()
4342:
4343:
4344: __all__ = [
4345:     'WiringFeatureFlags',
4346:     'DEFAULT_FLAGS',
4347: ]
4348:
4349:
4350:
4351: =====
4352: FILE: src/farfan_pipeline/core/wiring/observability.py
4353: =====
4354:
4355: """Observability instrumentation for wiring system.
4356:
4357: Provides OpenTelemetry tracing and structured logging for all wiring operations.
4358: """
4359:
4360: from __future__ import annotations
4361:
4362: import time
4363: from contextlib import contextmanager
4364: from typing import TYPE_CHECKING, Any
4365:
4366: import structlog
```

```
4367:
4368: if TYPE_CHECKING:
4369:     from collections.abc import Iterator
4370:
4371: try:
4372:     from opentelemetry import trace
4373:     from opentelemetry.trace import Status, StatusCode
4374:
4375:     HAS_OTEL = True
4376:     tracer = trace.get_tracer("farfan_core.wiring")
4377: except ImportError:
4378:     HAS_OTEL = False
4379:     tracer = None
4380:
4381:
4382: logger = structlog.get_logger(__name__)
4383:
4384:
4385: @contextmanager
4386: def trace_wiring_link(
4387:     link_name: str,
4388:     **attributes: Any,
4389: ) -> Iterator[dict[str, Any]]:
4390:     """Trace a wiring link operation.
4391:
4392:     Creates an OpenTelemetry span (if available) and logs structured messages.
4393:
4394:     Args:
4395:         link_name: Name of the wiring link (e.g., "cpp->adapter")
4396:         **attributes: Additional attributes to include in span/log
4397:
4398:     Yields:
4399:         Dict for adding dynamic attributes during operation
4400:
4401:     Example:
4402:         with trace_wiring_link("cpp->adapter", document_id="doc123") as attrs:
4403:             result = adapter.convert(cpp)
4404:             attrs["chunk_count"] = len(result.chunks)
4405: """
4406:     start_time = time.time()
4407:     dynamicAttrs: dict[str, Any] = {}
4408:
4409:     # Start span if OpenTelemetry is available
4410:     span = None
4411:     if HAS_OTEL and tracer:
4412:         span = tracer.start_span(f"wiring.link.{link_name}")
4413:         span.set_attribute("link", link_name)
4414:         for key, value in attributes.items():
4415:             if isinstance(value, (str, int, float, bool)):
4416:                 span.set_attribute(key, value)
4417:
4418:     # Log start
4419:     logger.info(
4420:         "wiring_link_start",
4421:         link=link_name,
4422:         **attributes,
```

```
4423:     )
4424:
4425:     try:
4426:         yield dynamic_attrs
4427:
4428:         # Success
4429:         latency_ms = (time.time() - start_time) * 1000
4430:
4431:         if span:
4432:             span.set_attribute("latency_ms", latency_ms)
4433:             span.set_attribute("ok", True)
4434:             for key, value in dynamic_attrs.items():
4435:                 if isinstance(value, (str, int, float, bool)):
4436:                     span.set_attribute(key, value)
4437:             span.set_status(Status.StatusCode.OK))
4438:
4439:             logger.info(
4440:                 "wiring_link_complete",
4441:                 link=link_name,
4442:                 latency_ms=latency_ms,
4443:                 ok=True,
4444:                 **attributes,
4445:                 **dynamic_attrs,
4446:             )
4447:
4448:     except Exception as e:
4449:         # Failure
4450:         latency_ms = (time.time() - start_time) * 1000
4451:
4452:         if span:
4453:             span.set_attribute("latency_ms", latency_ms)
4454:             span.set_attribute("ok", False)
4455:             span.set_attribute("error_type", type(e).__name__)
4456:             span.set_attribute("error_message", str(e))
4457:             span.set_status(Status.StatusCode.ERROR, str(e)))
4458:
4459:             logger.error(
4460:                 "wiring_link_failed",
4461:                 link=link_name,
4462:                 latency_ms=latency_ms,
4463:                 ok=False,
4464:                 error_type=type(e).__name__,
4465:                 error_message=str(e),
4466:                 **attributes,
4467:             )
4468:
4469:         raise
4470:
4471:     finally:
4472:         if span:
4473:             span.end()
4474:
4475:
4476: @contextmanager
4477: def trace_wiring_init(
4478:     phase: str,
```

```
4479:     **attributes: Any,
4480: ) -> Iterator[dict[str, Any]]:
4481:     """Trace a wiring initialization phase.
4482:
4483:     Args:
4484:         phase: Name of initialization phase
4485:         **attributes: Additional attributes
4486:
4487:     Yields:
4488:         Dict for adding dynamic attributes
4489:     """
4490:     start_time = time.time()
4491:     dynamic_attrs: dict[str, Any] = {}
4492:
4493:     span = None
4494:     if HAS_OTEL and tracer:
4495:         span = tracer.start_span(f"wiring.init.{phase}")
4496:         span.set_attribute("phase", phase)
4497:         for key, value in attributes.items():
4498:             if isinstance(value, (str, int, float, bool)):
4499:                 span.set_attribute(key, value)
4500:
4501:     logger.info(
4502:         "wiring_init_start",
4503:         phase=phase,
4504:         **attributes,
4505:     )
4506:
4507:     try:
4508:         yield dynamic_attrs
4509:
4510:         latency_ms = (time.time() - start_time) * 1000
4511:
4512:         if span:
4513:             span.set_attribute("latency_ms", latency_ms)
4514:             span.set_attribute("ok", True)
4515:             for key, value in dynamic_attrs.items():
4516:                 if isinstance(value, (str, int, float, bool)):
4517:                     span.set_attribute(key, value)
4518:             span.set_status(Status(StatusCode.OK))
4519:
4520:         logger.info(
4521:             "wiring_init_complete",
4522:             phase=phase,
4523:             latency_ms=latency_ms,
4524:             ok=True,
4525:             **attributes,
4526:             **dynamic_attrs,
4527:         )
4528:
4529:     except Exception as e:
4530:         latency_ms = (time.time() - start_time) * 1000
4531:
4532:         if span:
4533:             span.set_attribute("latency_ms", latency_ms)
4534:             span.set_attribute("ok", False)
```

```
4535:         span.set_attribute("error_type", type(e).__name__)
4536:         span.set_attribute("error_message", str(e))
4537:         span.set_status(Status.StatusCode.ERROR, str(e)))
4538:
4539:     logger.error(
4540:         "wiring_init_failed",
4541:         phase=phase,
4542:         latency_ms=latency_ms,
4543:         ok=False,
4544:         error_type=type(e).__name__,
4545:         error_message=str(e),
4546:         **attributes,
4547:     )
4548:
4549:     raise
4550:
4551: finally:
4552:     if span:
4553:         span.end()
4554:
4555:
4556: def log_wiring_metric(
4557:     metric_name: str,
4558:     value: float | int,
4559:     **labels: Any,
4560: ) -> None:
4561:     """Log a wiring metric.
4562:
4563:     Args:
4564:         metric_name: Name of the metric
4565:         value: Metric value
4566:         **labels: Metric labels
4567:     """
4568:     logger.info(
4569:         "wiring_metric",
4570:         metric=metric_name,
4571:         value=value,
4572:         **labels,
4573:     )
4574:
4575:
4576: __all__ = [
4577:     'trace_wiring_link',
4578:     'trace_wiring_init',
4579:     'log_wiring_metric',
4580:     'HAS_OTEL',
4581: ]
4582:
4583:
4584:
4585: =====
4586: FILE: src/farfan_pipeline/core/wiring/phase_0_validator.py
4587: =====
4588:
4589: """
4590: Phase 0 Configuration Validator
```

```
4591:
4592: This module provides a dedicated validator to enforce the C0-CONFIG-V1.0 contract,
4593: as specified in docs/contracts/C0-CONFIG-V1.0.md. It is executed at the very
4594: beginning of the wiring bootstrap process to ensure the system starts in a
4595: known, valid state.
4596: """
4597:
4598: import os
4599: from pathlib import Path
4600: from typing import Any, Dict, List
4601:
4602: class Phase0ValidationError(ValueError):
4603:     """Custom exception for Phase 0 validation errors."""
4604:     def __init__(self, message: str, missing_keys: List[str] | None = None, invalid_paths: Dict[str, str] | None = None):
4605:         super().__init__(message)
4606:         self.missing_keys = missing_keys or []
4607:         self.invalid_paths = invalid_paths or {}
4608:
4609: class Phase0Validator:
4610:     """
4611:     Enforces the Phase 0 configuration contract.
4612:     """
4613:     REQUIRED_KEYS = {
4614:         "monolith_path",
4615:         "questionnaire_hash",
4616:         "executor_config_path",
4617:         "calibration_profile",
4618:         "abort_on_insufficient",
4619:         "resource_limits",
4620:     }
4621:
4622:     def validate(self, config: Dict[str, Any]) -> None:
4623:         """
4624:             Validates the raw configuration dictionary against the Phase 0 contract.
4625:
4626:             Args:
4627:                 config: The raw configuration dictionary.
4628:
4629:             Raises:
4630:                 Phase0ValidationError: If the configuration is invalid.
4631:             """
4632:             self._check_mandatory_keys(config)
4633:             self._check_paths_and_permissions(config)
4634:
4635:     def _check_mandatory_keys(self, config: Dict[str, Any]) -> None:
4636:         """Ensures all required configuration keys are present."""
4637:         missing_keys = self.REQUIRED_KEYS - set(config.keys())
4638:         if missing_keys:
4639:             raise Phase0ValidationError(
4640:                 "Missing mandatory configuration keys.",
4641:                 missing_keys=sorted(list(missing_keys))
4642:             )
4643:
4644:     def _check_paths_and_permissions(self, config: Dict[str, Any]) -> None:
4645:         """Validates that file paths exist and have the correct permissions."""
4646:         monolith_path = Path(config["monolith_path"])
```

```
4647:     executor_path = Path(config["executor_config_path"])
4648:     invalid_paths = {}
4649:
4650:     # Check for existence
4651:     if not monolith_path.exists():
4652:         invalid_paths["monolith_path"] = f"File not found at {monolith_path}"
4653:     if not executor_path.exists():
4654:         invalid_paths["executor_config_path"] = f"File not found at {executor_path}"
4655:
4656:     if invalid_paths:
4657:         raise Phase0ValidationError("Invalid file paths in configuration.", invalid_paths=invalid_paths)
4658:
4659:     # Check monolith permissions (must be read-only)
4660:     if not os.access(monolith_path, os.R_OK):
4661:         invalid_paths["monolith_path"] = f"File at {monolith_path} is not readable."
4662:         raise Phase0ValidationError(
4663:             "Invalid file permissions in configuration.",
4664:             invalid_paths=invalid_paths
4665:         )
4666:     elif os.access(monolith_path, os.W_OK):
4667:         invalid_paths["monolith_path"] = f"File at {monolith_path} must be read-only."
4668:         raise Phase0ValidationError(
4669:             "Invalid file permissions in configuration.",
4670:             invalid_paths=invalid_paths
4671:         )
4672:
4673:
4674:
4675: =====
4676: FILE: src/farfan_pipeline/core/wiring/validation.py
4677: =====
4678:
4679: """Contract validation between wiring links.
4680:
4681: Validates that deliverables from one module match expectations of the next.
4682: All validations use Pydantic models for type safety and prescriptive errors.
4683: """
4684:
4685: from __future__ import annotations
4686:
4687: from typing import Any
4688:
4689: import blake3
4690: import structlog
4691: from pydantic import BaseModel, ValidationError
4692:
4693: from farfan_pipeline.core.wiring.contracts import (
4694:     AdapterExpectation,
4695:     AggregateExpectation,
4696:     ArgRouterExpectation,
4697:     ArgRouterPayloadDeliverable,
4698:     CPPDeliverable,
4699:     SPCDeliverable,
4700:     EnrichedChunkDeliverable,
4701:     ExecutorInputDeliverable,
4702:     FeatureTableDeliverable,
```

```
4703:     OrchestratorExpectation,
4704:     PreprocessedDocumentDeliverable,
4705:     ReportExpectation,
4706:     ScoreExpectation,
4707:     ScoresDeliverable,
4708:     SignalPackDeliverable,
4709:     SignalRegistryExpectation,
4710: )
4711: from farfan_pipeline.core.wiring.errors import WiringContractError
4712:
4713: logger = structlog.get_logger(__name__)
4714:
4715:
4716: class LinkValidator:
4717:     """Validator for individual wiring links.
4718:
4719:     Validates deliverable\206\222expectation contracts and computes hashes for determinism.
4720:     """
4721:
4722:     def __init__(self, link_name: str) -> None:
4723:         """Initialize validator for a specific link.
4724:
4725:         Args:
4726:             link_name: Name of the link (e.g., "cpp->adapter")
4727:         """
4728:         self.link_name = link_name
4729:         self._validation_count = 0
4730:         self._failure_count = 0
4731:
4732:     def validate(
4733:         self,
4734:         deliverable_data: dict[str, Any],
4735:         deliverable_model: type[BaseModel],
4736:         expectation_model: type[BaseModel],
4737:     ) -> None:
4738:         """Validate deliverable matches expectation.
4739:
4740:         Args:
4741:             deliverable_data: Actual data being delivered
4742:             deliverable_model: Pydantic model for deliverable
4743:             expectation_model: Pydantic model for expectation
4744:
4745:         Raises:
4746:             WiringContractError: If validation fails
4747:         """
4748:         self._validation_count += 1
4749:
4750:         # Validate deliverable schema
4751:         try:
4752:             deliverable = deliverable_model.model_validate(deliverable_data)
4753:         except ValidationError as e:
4754:             self._failure_count += 1
4755:
4756:             errors = e.errors()
4757:             first_error = errors[0] if errors else {}
4758:             field = ".".join(str(loc) for loc in first_error.get("loc", []))
```

```
4759:             raise WiringContractError(
4760:                 link=self.link_name,
4761:                 expected_schema=deliverable_model.__name__,
4762:                 received_schema=type(deliverable_data).__name__,
4763:                 field=field or None,
4764:                 fix=f"Ensure {self.link_name} produces valid {deliverable_model.__name__}. "
4765:                     f"Error: {first_error.get('msg', 'Unknown')}",
4766:             ) from e
4767:
4768:
4769:     # Validate expectation schema
4770:     # (This ensures the downstream consumer can handle the deliverable)
4771:     try:
4772:         expectation_model.model_validate(deliverable.model_dump())
4773:     except ValidationError as e:
4774:         self._failure_count += 1
4775:
4776:         errors = e.errors()
4777:         first_error = errors[0] if errors else {}
4778:         field = ".".join(str(loc) for loc in first_error.get("loc", []))
4779:
4780:         raise WiringContractError(
4781:             link=self.link_name,
4782:             expected_schema=expectation_model.__name__,
4783:             received_schema=deliverable_model.__name__,
4784:             field=field or None,
4785:             fix=f"Deliverable from {self.link_name} does not meet expectations. "
4786:                 f"Error: {first_error.get('msg', 'Unknown')}",
4787:         ) from e
4788:
4789:     logger.debug(
4790:         "contract_validated",
4791:         link=self.link_name,
4792:         deliverable=deliverable_model.__name__,
4793:         expectation=expectation_model.__name__,
4794:     )
4795:
4796: def compute_hash(self, data: dict[str, Any]) -> str:
4797:     """Compute deterministic hash of data for this link.
4798:
4799:     Args:
4800:         data: Data to hash
4801:
4802:     Returns:
4803:         BLAKE3 hash hex string
4804:     """
4805:     import json
4806:
4807:     # Sort keys for deterministic hashing
4808:     json_str = json.dumps(data, sort_keys=True, separators=(',', ':'))
4809:     hash_value = blake3.blake3(json_str.encode('utf-8')).hexdigest()
4810:
4811:     logger.debug(
4812:         "link_hash_computed",
4813:         link=self.link_name,
4814:         hash=hash_value[:16],
```

```
4815:         )
4816:
4817:     return hash_value
4818:
4819: def get_metrics(self) -> dict[str, Any]:
4820:     """Get validation metrics.
4821:
4822:     Returns:
4823:         Dict with validation_count and failure_count
4824:     """
4825:     return {
4826:         "validation_count": self._validation_count,
4827:         "failure_count": self._failure_count,
4828:         "success_rate": (
4829:             (self._validation_count - self._failure_count) / self._validation_count
4830:             if self._validation_count > 0
4831:             else 1.0
4832:         ),
4833:     }
4834:
4835:
4836: class WiringValidator:
4837:     """Central validator for all wiring links.
4838:
4839:     Provides validation methods for each i\206\222i+1 link in the system.
4840:     """
4841:
4842:     def __init__(self) -> None:
4843:         """Initialize wiring validator."""
4844:         self._validators = {
4845:             "cpp->adapter": LinkValidator("cpp->adapter"),
4846:             "spc->adapter": LinkValidator("spc->adapter"),
4847:             "adapter->orchestrator": LinkValidator("adapter->orchestrator"),
4848:             "orchestrator->argrouter": LinkValidator("orchestrator->argrouter"),
4849:             "argrouter->executors": LinkValidator("argrouter->executors"),
4850:             "signals->registry": LinkValidator("signals->registry"),
4851:             "executors->aggregate": LinkValidator("executors->aggregate"),
4852:             "aggregate->score": LinkValidator("aggregate->score"),
4853:             "score->report": LinkValidator("score->report"),
4854:         }
4855:
4856:         logger.info("wiring_validator_initialized", links=len(self._validators))
4857:
4858:     def validate_spc_to_adapter(self, spc_data: dict[str, Any]) -> None:
4859:         """Validate SPC \206\222 Adapter link.
4860:
4861:         Args:
4862:             spc_data: SPC deliverable data
4863:
4864:         Raises:
4865:             WiringContractError: If validation fails
4866:         """
4867:         from farfan_pipeline.core.wiring.contracts import SPCDeliverable
4868:
4869:         validator = self._validators["spc->adapter"]
4870:         validator.validate()
```

```
4871:         deliverable_data=spc_data,
4872:         deliverable_model=SPCDeliverable,
4873:         expectation_model=AdapterExpectation,
4874:     )
4875:
4876:     def validate_cpp_to_adapter(self, cpp_data: dict[str, Any]) -> None:
4877:         """Validate CPP à\206\222 Adapter link.
4878:
4879:             DEPRECATED: Use validate_spc_to_adapter instead.
4880:
4881:             Args:
4882:                 cpp_data: CPP deliverable data
4883:
4884:             Raises:
4885:                 WiringContractError: If validation fails
4886:             """
4887:             # Forward to new validator if possible, but keep legacy link name for now
4888:             # to avoid breaking existing hashes if they depend on link name.
4889:             # However, we should warn.
4890:             import warnings
4891:             warnings.warn(
4892:                 "validate_cpp_to_adapter is deprecated. Use validate_spc_to_adapter instead.",
4893:                 DeprecationWarning,
4894:                 stacklevel=2
4895:             )
4896:
4897:             validator = self._validators["cpp->adapter"]
4898:             validator.validate(
4899:                 deliverable_data=cpp_data,
4900:                 deliverable_model=CPPDeliverable,
4901:                 expectation_model=AdapterExpectation,
4902:             )
4903:
4904:     def validate_adapter_to_orchestrator(
4905:         self,
4906:         preprocessed_doc_data: dict[str, Any],
4907:     ) -> None:
4908:         """Validate Adapter à\206\222 Orchestrator link.
4909:
4910:             Args:
4911:                 preprocessed_doc_data: PreprocessedDocument deliverable data
4912:
4913:             Raises:
4914:                 WiringContractError: If validation fails
4915:             """
4916:             validator = self._validators["adapter->orchestrator"]
4917:             validator.validate(
4918:                 deliverable_data=preprocessed_doc_data,
4919:                 deliverable_model=PreprocessedDocumentDeliverable,
4920:                 expectation_model=OrchestratorExpectation,
4921:             )
4922:
4923:     def validate_orchestrator_to_argrouter(
4924:         self,
4925:         payload_data: dict[str, Any],
4926:     ) -> None:
```

```
4927:     """Validate Orchestrator \206\222 ArgRouter link.
4928:
4929:     Args:
4930:         payload_data: ArgRouter payload deliverable data
4931:
4932:     Raises:
4933:         WiringContractError: If validation fails
4934:     """
4935:     validator = self._validators["orchestrator->argrouter"]
4936:     validator.validate(
4937:         deliverable_data=payload_data,
4938:         deliverable_model=ArgRouterPayloadDeliverable,
4939:         expectation_model=ArgRouterExpectation,
4940:     )
4941:
4942:     def validate_argrouter_to_executors(
4943:         self,
4944:         executor_input_data: dict[str, Any],
4945:     ) -> None:
4946:         """Validate ArgRouter \206\222 Executors link.
4947:
4948:         Args:
4949:             executor_input_data: Executor input deliverable data
4950:
4951:         Raises:
4952:             WiringContractError: If validation fails
4953:         """
4954:         self._validators["argrouter->executors"]
4955:         # Note: ExecutorInput doesn't have a matching expectation model yet
4956:         # For now, just validate the deliverable
4957:         from pydantic import ValidationError
4958:
4959:         try:
4960:             ExecutorInputDeliverable.model_validate(executor_input_data)
4961:         except ValidationError as e:
4962:             raise WiringContractError(
4963:                 link="argrouter->executors",
4964:                 expected_schema=ExecutorInputDeliverable.__name__,
4965:                 received_schema=type(executor_input_data).__name__,
4966:                 field=str(e.errors()[0].get("loc", [])) if e.errors() else None,
4967:                 fix="Ensure ArgRouter produces valid ExecutorInputDeliverable",
4968:             ) from e
4969:
4970:     def validate_signals_to_registry(
4971:         self,
4972:         signal_pack_data: dict[str, Any],
4973:     ) -> None:
4974:         """Validate Signals \206\222 Registry link.
4975:
4976:         Args:
4977:             signal_pack_data: SignalPack deliverable data
4978:
4979:         Raises:
4980:             WiringContractError: If validation fails
4981:         """
4982:         validator = self._validators["signals->registry"]
```

```
4983:         validator.validate(
4984:             deliverable_data=signal_pack_data,
4985:             deliverable_model=SignalPackDeliverable,
4986:             expectation_model=SignalRegistryExpectation,
4987:         )
4988:
4989:     def validate_executors_to_aggregate(
4990:         self,
4991:         enriched_chunks_data: list[dict[str, Any]],
4992:     ) -> None:
4993:         """Validate Executors à\206\222 Aggregate link.
4994:
4995:         Args:
4996:             enriched_chunks_data: List of enriched chunk deliverables
4997:
4998:         Raises:
4999:             WiringContractError: If validation fails
5000:         """
5001:         validator = self._validators["executors->aggregate"]
5002:
5003:         # Validate each chunk
5004:         for i, chunk_data in enumerate(enriched_chunks_data):
5005:             try:
5006:                 EnrichedChunkDeliverable.model_validate(chunk_data)
5007:             except ValidationError as e:
5008:                 raise WiringContractError(
5009:                     link="executors->aggregate",
5010:                     expected_schema=EnrichedChunkDeliverable.__name__,
5011:                     received_schema=type(chunk_data).__name__,
5012:                     field=f"chunk[{i}]",
5013:                     fix=f"Ensure all enriched chunks are valid. Chunk {i} failed validation.",
5014:                 ) from e
5015:
5016:         # Validate aggregate expectation
5017:         validator.validate(
5018:             deliverable_data={"enriched_chunks": enriched_chunks_data},
5019:             deliverable_model=AggregateExpectation,
5020:             expectation_model=AggregateExpectation, # Same for now
5021:         )
5022:
5023:     def validate_aggregate_to_score(
5024:         self,
5025:         feature_table_data: dict[str, Any],
5026:     ) -> None:
5027:         """Validate Aggregate à\206\222 Score link.
5028:
5029:         Args:
5030:             feature_table_data: Feature table deliverable data
5031:
5032:         Raises:
5033:             WiringContractError: If validation fails
5034:         """
5035:         validator = self._validators["aggregate->score"]
5036:         validator.validate(
5037:             deliverable_data=feature_table_data,
5038:             deliverable_model=FeatureTableDeliverable,
```

```
5039:             expectation_model=ScoreExpectation,
5040:         )
5041:
5042:     def validate_score_to_report(
5043:         self,
5044:         scores_data: dict[str, Any],
5045:     ) -> None:
5046:         """Validate Score â\206\222 Report link.
5047:
5048:         Args:
5049:             scores_data: Scores deliverable data
5050:
5051:         Raises:
5052:             WiringContractError: If validation fails
5053:         """
5054:         validator = self._validators["score->report"]
5055:         validator.validate(
5056:             deliverable_data=scores_data,
5057:             deliverable_model=ScoresDeliverable,
5058:             expectation_model=ReportExpectation,
5059:         )
5060:
5061:     def compute_link_hash(self, link_name: str, data: dict[str, Any]) -> str:
5062:         """Compute hash for a specific link.
5063:
5064:         Args:
5065:             link_name: Name of the link
5066:             data: Data to hash
5067:
5068:         Returns:
5069:             BLAKE3 hash hex string
5070:
5071:         Raises:
5072:             KeyError: If link_name is not recognized
5073:         """
5074:         validator = self._validators[link_name]
5075:         return validator.compute_hash(data)
5076:
5077:     def get_all_metrics(self) -> dict[str, dict[str, Any]]:
5078:         """Get metrics for all links.
5079:
5080:         Returns:
5081:             Dict mapping link names to their metrics
5082:         """
5083:         return {
5084:             link_name: validator.get_metrics()
5085:             for link_name, validator in self._validators.items()
5086:         }
5087:
5088:     def get_summary(self) -> dict[str, Any]:
5089:         """Get summary of all validation activity.
5090:
5091:         Returns:
5092:             Summary dict with total counts and success rate
5093:         """
5094:         all_metrics = self.get_all_metrics()
```

```
5095:
5096:     total_validations = sum(m["validation_count"] for m in all_metrics.values())
5097:     total_failures = sum(m["failure_count"] for m in all_metrics.values())
5098:
5099:     return {
5100:         "total_validations": total_validations,
5101:         "total_failures": total_failures,
5102:         "overall_success_rate": (
5103:             (total_validations - total_failures) / total_validations
5104:             if total_validations > 0
5105:             else 1.0
5106:         ),
5107:         "links": all_metrics,
5108:     }
5109:
5110:
5111: __all__ = [
5112:     'LinkValidator',
5113:     'WiringValidator',
5114: ]
5115:
5116:
5117:
5118: =====
5119: FILE: src/farfan_pipeline/dashboard_atroz/__init__.py
5120: =====
5121:
5122: """Dashboard Atroz package
5123:
5124: This package groups and labels the modules orchestrating the current dashboard
5125: without moving original sources. It provides stable import paths while
5126: preserving existing module locations.
5127: """
5128:
5129: # Re-export key orchestrator components for convenience
5130: from ..api.api_server import app as flask_app # type: ignore
5131:
5132:
5133:
5134: =====
5135: FILE: src/farfan_pipeline/dashboard_atroz/api_server.py
5136: =====
5137:
5138: """Shim to run the current dashboard Flask API server.
5139:
5140: Keeps import paths stable by referencing the existing server implementation
5141: under 'farfan_pipeline.api.api_server'.
5142: """
5143:
5144: from ..api.api_server import app
5145:
5146: if __name__ == "__main__":
5147:     # Delegate to the existing Flask app run configuration
5148:     app.run(host="0.0.0.0", port=5000, debug=True)
5149:
5150:
```

```
5151:  
5152: =====  
5153: FILE: src/farfan_pipeline/dashboard_atroz/data_service.py  
5154: =====  
5155:  
5156: """Shim for the dashboard transformer service.  
5157:  
5158: References the implementation in 'farfan_pipeline.api.dashboard_data_service'.  
5159: """  
5160:  
5161: from ..api.dashboard_data_service import DashboardDataService  
5162:  
5163: __all__ = ["DashboardDataService"]  
5164:  
5165:
```