```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 17
================================================================================
Generated: 2025-12-07T06:17:23.668686
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/core/phases/phase_orchestrator.py
================================================================================

"""
Phase Orchestrator - Constitutional Sequence Enforcement
=========================================================

This module implements the PhaseOrchestrator which GUARANTEES that:

1. Phases execute in STRICT sequence (0 â\206\222 1 â\206\222 Adapter â\206\222 2)
2. Each phase’s output becomes the NEXT phase’s input
3. NO phase can be bypassed
4. ALL contracts are validated at boundaries
5. ALL invariants are checked
6. FULL traceability in manifest

The orchestrator is the SINGLE point of entry for pipeline execution.
It is IMPOSSIBLE to run phases out of order or skip validation.

Design Principles:
------------------
- **Single Entry Point**: Only ‘run_pipeline()‘ executes the full sequence
- **No Bypass**: Phases cannot be called directly from outside
- **Contract Enforcement**: All inputs/outputs validated
- **Deterministic**: Same Phase0Input â\206\222 same outputs
- **Auditable**: Full manifest with all phase boundaries

Phase Sequence (IMMUTABLE):
---------------------------
Phase 0: input_validation
    Input: Phase0Input (pdf_path, run_id, questionnaire_path)
    Output: CanonicalInput
    â\206\223
Phase 1: spc_ingestion
    Input: CanonicalInput
    Output: CanonPolicyPackage
    â\206\223
Adapter: phase1_to_phase2
    Input: CanonPolicyPackage
    Output: PreprocessedDocument
    â\206\223
Phase 2: microquestions
    Input: PreprocessedDocument
    Output: Phase2Result

Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
```

```
"""

from __future__ import annotations

import logging
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

from farfan_pipeline.core.orchestrator.factory import build_processor
from farfan_pipeline.core.phases.phase_protocol import (
    ContractValidationResult,
    PhaseManifestBuilder,
    PhaseMetadata,
)
from farfan_pipeline.core.phases.phase0_input_validation import (
    CanonicalInput,
    Phase0Input,
    Phase0ValidationContract,
)
from farfan_pipeline.core.phases.phase1_spc_ingestion import (
    Phase1SPCIngestionContract,
)
from farfan_pipeline.core.phases.phase2_types import validate_phase2_result
from farfan_pipeline.core.phases.phase3_chunk_routing import (
    Phase3ChunkRoutingContract,
    Phase3Input,
)

logger = logging.getLogger(__name__)


@dataclass
class PipelineResult:
    """
    Complete result of pipeline execution.

    This is the ONLY output of PhaseOrchestrator.run_pipeline().
    """

    success: bool
    run_id: str

    # Phase outputs (populated if phase succeeded)
    canonical_input: CanonicalInput | None = None
    canon_policy_package: Any | None = None  # CanonPolicyPackage
    preprocessed_document: Any | None = None  # PreprocessedDocument
    phase2_result: Any | None = None  # Phase2Result
    phase3_result: Any | None = None  # Phase3Result

    # Execution metadata
    phases_completed: int = 0
    phases_failed: int = 0
    total_duration_ms: float = 0.0

    # Error tracking
```

~~~
    errors: list[str] = field(default_factory=list)

    # Manifest
    manifest: dict[str, Any] = field(default_factory=dict)


class PhaseOrchestrator:
    """
    Orchestrator that enforces the canonical phase sequence.

    This class is the CONSTITUTIONAL GUARANTEE that phases execute
    in order with full contract validation.

    Usage:
    ------
    ```python
    orchestrator = PhaseOrchestrator()
    result = await orchestrator.run_pipeline(
        pdf_path=Path("plan.pdf"),
        run_id="plan1",
        questionnaire_path=Path("questionnaire.json"),
        artifacts_dir=Path("artifacts/plan1"),
    )

    if result.success:
        print(f"Pipeline succeeded: {result.phases_completed} phases")
    else:
        print(f"Pipeline failed: {result.errors}")
    ```
    """

    def __init__(self):
        """Initialize orchestrator with phase contracts."""
        logger.info("Initializing PhaseOrchestrator with constitutional constraints")

        # Initialize phase contracts
        self.phase0 = Phase0ValidationContract()
        self.phase1 = Phase1SPCIngestionContract()

        # Import and initialize adapter contract
        from farfan_pipeline.core.phases.phase1_to_phase2_adapter import AdapterContract
        self.adapter = AdapterContract()

        # Initialize Phase 3 contract
        self.phase3 = Phase3ChunkRoutingContract()

        # self.phase2 = Phase2Contract()     # To be implemented

        # Initialize manifest builder
        self.manifest_builder = PhaseManifestBuilder()

        logger.info("PhaseOrchestrator initialized successfully")

    async def run_pipeline(
        self,
        pdf_path: Path,
~~~

```
        run_id: str,
        questionnaire_path: Path | None = None,
        artifacts_dir: Path | None = None,
    ) -> PipelineResult:
        """
        Execute the COMPLETE canonical pipeline in STRICT sequence.

        This is the ONLY way to run the pipeline. It enforces:
        1. Phase 0 â\206\222 Phase 1 â\206\222 Adapter â\206\222 Phase 2
        2. Contract validation at ALL boundaries
        3. Invariant checking for ALL phases
        4. Full manifest generation

        Args:
            pdf_path: Path to input PDF
            run_id: Unique run identifier
            questionnaire_path: Optional questionnaire path
            artifacts_dir: Optional directory for artifacts

        Returns:
            PipelineResult with success status and all phase outputs

        Raises:
            This method does NOT raise exceptions. All errors are captured
            in PipelineResult.errors and PipelineResult.success = False.
        """
        logger.info(f"Starting pipeline execution: run_id={run_id}")

        result = PipelineResult(
            success=False,  # Will be set to True only if ALL phases succeed
            run_id=run_id,
        )

        # Create artifacts directory if provided
        if artifacts_dir:
            artifacts_dir.mkdir(parents=True, exist_ok=True)

        try:
            # ================================================================
            # PHASE 0: Input Validation
            # ================================================================
            logger.info("=" * 70)
            logger.info("PHASE 0: Input Validation")
            logger.info("=" * 70)

            phase0_input = Phase0Input(
                pdf_path=pdf_path,
                run_id=run_id,
                questionnaire_path=questionnaire_path,
            )

            canonical_input, phase0_metadata = await self.phase0.run(phase0_input)

            # Record Phase 0 in manifest
            self.manifest_builder.record_phase(
                phase_name="phase0_input_validation",
                metadata=phase0_metadata,
                input_validation=self.phase0.validate_input(phase0_input),
                output_validation=self.phase0.validate_output(canonical_input),
                invariants_checked=[inv.name for inv in self.phase0.invariants],
                artifacts=[],  # No artifacts for Phase 0
            )

            result.canonical_input = canonical_input
            result.phases_completed += 1
            result.total_duration_ms += phase0_metadata.duration_ms or 0.0

            logger.info(
                f"Phase 0 completed successfully in {phase0_metadata.duration_ms:.0f}ms"
            )

            # ================================================================
            # PHASE 1: SPC Ingestion
            # ================================================================
            logger.info("=" * 70)
            logger.info("PHASE 1: SPC Ingestion (15 subfases)")
            logger.info("=" * 70)

            # Phase 1 input is Phase 0 output (guaranteed by type system)
            cpp, phase1_metadata = await self.phase1.run(canonical_input)

            # Record Phase 1 in manifest
            self.manifest_builder.record_phase(
                phase_name="phase1_spc_ingestion",
                metadata=phase1_metadata,
                input_validation=self.phase1.validate_input(canonical_input),
                output_validation=self.phase1.validate_output(cpp),
                invariants_checked=[inv.name for inv in self.phase1.invariants],
                artifacts=[],  # Artifacts tracked separately
            )

            result.canon_policy_package = cpp
            result.phases_completed += 1
            result.total_duration_ms += phase1_metadata.duration_ms or 0.0

            logger.info(
                f"Phase 1 completed successfully in {phase1_metadata.duration_ms:.0f}ms"
            )
            logger.info(f"Generated {len(cpp.chunk_graph.chunks)} chunks")

            # ================================================================
            # ADAPTER: Phase 1 â\206\222 Phase 2
            # ================================================================
            logger.info("=" * 70)
            logger.info("ADAPTER: CanonPolicyPackage â\206\222 PreprocessedDocument")
            logger.info("=" * 70)

            # Run adapter with contract enforcement
            preprocessed, adapter_metadata = await self.adapter.run(cpp)

            # Record Adapter in manifest
            self.manifest_builder.record_phase(
                phase_name="phase1_to_phase2_adapter",
                metadata=adapter_metadata,
                input_validation=self.adapter.validate_input(cpp),
                output_validation=self.adapter.validate_output(preprocessed),
                invariants_checked=[inv.name for inv in self.adapter.invariants],
                artifacts=[],
            )

            result.preprocessed_document = preprocessed
            result.phases_completed += 1
            result.total_duration_ms += adapter_metadata.duration_ms or 0.0

            logger.info(
                f"Adapter completed successfully in {adapter_metadata.duration_ms:.0f}ms"
            )
            logger.info(
                f"PreprocessedDocument: {len(preprocessed.sentences)} sentences, "
                f"mode={preprocessed.processing_mode}"
            )

            # ==================================================================
            # CORE ORCHESTRATOR: Phases 0-10 (Includes Micro-Questions)
            # ==================================================================
            logger.info("=" * 70)
            logger.info("CORE ORCHESTRATOR: Executing Phases 0-10")
            logger.info("=" * 70)

            # --- Imports for Phase 2 Integration ---
            from datetime import datetime, timedelta, timezone

            # --- Execute Core Orchestrator ---
            processor = build_processor()
            p2_block_started_at = datetime.now(timezone.utc)
            core_results = await processor.orchestrator.process_development_plan_async(
                pdf_path=str(pdf_path),
                preprocessed_document=preprocessed,
            )
            p2_block_finished_at = datetime.now(timezone.utc)

            # --- Process and Record Phase 2 ---
            phase2_success = False
            phase2_errors: list[str] = []
            phase2_questions: list[dict[str, Any]] | None = None

            if len(core_results) >= 3:
                phase2_core = core_results[2]  # FASE 2 - Micro Preguntas
                result.phase2_result = phase2_core.data if phase2_core.success else None

                if phase2_core.success:
                    is_valid, validation_errors, normalized_questions = validate_phase2_result(
                        phase2_core.data
                    )
                    phase2_questions = normalized_questions
                    if not is_valid:
                        phase2_errors.extend(validation_errors)
                        phase2_errors.append(
                            "Phase 2 failed structural invariant: questions list is empty or missing."
                        )
                    phase2_success = phase2_core.success and is_valid
                else:
                    phase2_errors.append(
                        f"Core phase 2 returned error: {phase2_core.error}"
                    )

                # --- Create Manifest Entry for Phase 2 ---
                p2_error_msg = "; ".join(phase2_errors) if phase2_errors else None

                # Approximate start/end times for the manifest metadata
                p2_duration = timedelta(milliseconds=phase2_core.duration_ms)
                p2_started_at_approx = p2_block_finished_at - p2_duration

                p2_metadata = PhaseMetadata(
                    phase_name="phase2_microquestions",
                    success=phase2_success,
                    error=p2_error_msg,
                    duration_ms=phase2_core.duration_ms,
                    started_at=p2_started_at_approx.isoformat(),
                    finished_at=p2_block_finished_at.isoformat(),
                )

                # Create validation results to satisfy the manifest builder
                dummy_input_validation = ContractValidationResult(
                    passed=True,
                    contract_type="input",
                    phase_name="phase2_microquestions",
                )
                dummy_output_validation = ContractValidationResult(
                    passed=phase2_success,
                    contract_type="output",
                    phase_name="phase2_microquestions",
                    errors=phase2_errors,
                )

                self.manifest_builder.record_phase(
                    phase_name="phase2_microquestions",
                    metadata=p2_metadata,
                    input_validation=dummy_input_validation,
                    output_validation=dummy_output_validation,
                    invariants_checked=["questions_are_present_and_non_empty"],
                    artifacts=[],
                )
                self.manifest_builder.phases["phase2_microquestions"]["question_count"] = len(phase2_questions or [])
                if phase2_errors:
                    self.manifest_builder.phases["phase2_microquestions"]["errors"] = list(phase2_errors)

                if not phase2_success:
                    error_msg = f"Core Orchestrator Phase 2 failed: {p2_error_msg}"
                    logger.error(error_msg)
                    result.errors.append(error_msg)
                    result.phases_failed += 1
                else:
                    # Only add core result count if Phase 2 was successful
                    result.phase2_result = {"questions": phase2_questions or []}
                    result.phases_completed += len(core_results)
                    logger.info(
                        f"Core Orchestrator completed {len(core_results)} phases successfully"
                    )

                    # ==================================================================
                    # PHASE 3: Chunk Routing
                    # ==================================================================
                    logger.info("=" * 70)
                    logger.info("PHASE 3: Chunk Routing")
                    logger.info("=" * 70)

                    # Phase 3 input is preprocessed document + Phase 2 questions
                    phase3_input = Phase3Input(
                        preprocessed_document=preprocessed,
                        questions=phase2_questions or []
                    )

                    phase3_result, phase3_metadata = await self.phase3.run(phase3_input)

                    # Record Phase 3 in manifest
                    self.manifest_builder.record_phase(
                        phase_name="phase3_chunk_routing",
                        metadata=phase3_metadata,
                        input_validation=self.phase3.validate_input(phase3_input),
                        output_validation=self.phase3.validate_output(phase3_result),
                        invariants_checked=[inv.name for inv in self.phase3.invariants],
                        artifacts=[]
                    )

                    result.phase3_result = phase3_result
                    result.phases_completed += 1
                    result.total_duration_ms += phase3_metadata.duration_ms or 0.0

                    logger.info(
                        f"Phase 3 completed successfully in {phase3_metadata.duration_ms:.0f}ms"
                    )
                    logger.info(
                        f"Routed {phase3_result.successful_routes}/{phase3_result.total_questions} questions"
                    )

            else:
                # Phase 2 was not even present in the results
                missing_p2_error = "Core Orchestrator did not produce a result for Phase 2."
                logger.error(missing_p2_error)
                result.errors.append(missing_p2_error)
                result.phases_failed += 1
                # Create a failure record in the manifest
                p2_metadata = PhaseMetadata(
                    phase_name="phase2_microquestions",
                    success=False,
                    error=missing_p2_error,
                    started_at=p2_block_started_at.isoformat(),
                    finished_at=p2_block_finished_at.isoformat(),
                    duration_ms=(p2_block_finished_at-p2_block_started_at).total_seconds() * 1000,
                )
                self.manifest_builder.record_phase(
                    phase_name="phase2_microquestions",
                    metadata=p2_metadata,
                    input_validation=ContractValidationResult(passed=False, contract_type="input", phase_name="phase2_microquestions", errors=[missing_p2_error]),
                    output_validation=ContractValidationResult(passed=False, contract_type="output", phase_name="phase2_microquestions", errors=[missing_p2_error]),
                    invariants_checked=[],
                    artifacts=[],
                )
                self.manifest_builder.phases["phase2_microquestions"]["question_count"] = 0
                self.manifest_builder.phases["phase2_microquestions"]["errors"] = [missing_p2_error]


            # ================================================================
            # PIPELINE SUCCESS
            # ================================================================
            # Success is now conditional on all canonical phases, including Phase 2
            all_phases_ok = all(
                p.get("status") == "success"
                for p in self.manifest_builder.phases.values()
            )

            if all_phases_ok:
                result.success = True
                logger.info("=" * 70)
                logger.info(f"PIPELINE COMPLETED SUCCESSFULLY")
                logger.info(f"Phases completed: {result.phases_completed}")
                logger.info(f"Total duration: {result.total_duration_ms:.0f}ms")
                logger.info("=" * 70)
            else:
                # Ensure result.success is False if we got here with a failure
                result.success = False
                final_error = f"Pipeline failed. Check manifest for details. Completed: {result.phases_completed}, Failed: {result.phases_failed}"
                if not result.errors:
                    result.errors.append(final_error)
                logger.error(final_error)

        except Exception as e:
            # Capture error
            error_msg = f"Pipeline failed: {e}"
            logger.error(error_msg, exc_info=True)
            result.errors.append(error_msg)
            result.success = False
            result.phases_failed += 1

        finally:
            # Always generate manifest
            result.manifest = self.manifest_builder.to_dict()
            phase2_entry = result.manifest.get("phases", {}).get("phase2_microquestions")
            if phase2_entry is not None:
                result.manifest["phases"]["phase2"] = phase2_entry

            # Save manifest if artifacts_dir provided
            if artifacts_dir:
                manifest_path = artifacts_dir / "phase_manifest.json"
                self.manifest_builder.save(manifest_path)
                logger.info(f"Phase manifest saved to {manifest_path}")

        return result


__all__ = [
    "PhaseOrchestrator",
    "PipelineResult",
]



================================================================================
FILE: src/farfan_pipeline/core/phases/phase_protocol.py
================================================================================

"""
Phase Contract Protocol - Constitutional Constraint System
============================================================

This module implements the constitutional constraint framework where each phase:

1. Has an EXPLICIT input contract (typed, validated)
2. Has an EXPLICIT output contract (typed, validated)
3. Communicates ONLY through these contracts (no side channels)
4. Is enforced by validators (runtime contract checking)
5. Is tracked in the verification manifest (full traceability)

Design Principles:
------------------
- **Single Entry Point**: Each phase accepts exactly ONE input type
- **Single Exit Point**: Each phase produces exactly ONE output type
- **No Bypass**: The orchestrator enforces sequential execution
- **Verifiable**: All contracts are validated and logged
- **Deterministic**: Same input â\206\222 same output (modulo controlled randomness)

Phase Structure:
----------------
phase0_input_validation:
    Input: Phase0Input (raw PDF path + run_id)
    Output: CanonicalInput (validated, hashed, ready)

phase1_spc_ingestion:
    Input: CanonicalInput
    Output: CanonPolicyPackage (60 chunks, PAÃ\227DIM structured)

phase1_to_phase2_adapter:
    Input: CanonPolicyPackage
    Output: PreprocessedDocument (chunked mode)

phase2_microquestions:
    Input: PreprocessedDocument
    Output: Phase2Result (305 questions answered)

```

```
Author: F.A.R.F.A.N Architecture Team
Date: 2025-01-19
"""

from __future__ import annotations

import hashlib
import json
from abc import ABC, abstractmethod
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any, Generic, TypeVar

from pydantic import BaseModel, Field, ValidationError

# Type variables for generic phase contracts
TInput = TypeVar("TInput")
TOutput = TypeVar("TOutput")


@dataclass
class PhaseInvariant:
    """An invariant that must hold for a phase."""

    name: str
    description: str
    check: callable  # Function that returns bool
    error_message: str


@dataclass
class PhaseMetadata:
    """Metadata for a phase execution."""

    phase_name: str
    started_at: str
    finished_at: str | None = None
    duration_ms: float | None = None
    success: bool = False
    error: str | None = None


@dataclass
class ContractValidationResult:
    """Result of validating a contract."""

    passed: bool
    contract_type: str  # "input" or "output"
    phase_name: str
    errors: list[str] = field(default_factory=list)
    warnings: list[str] = field(default_factory=list)
    validation_timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )

```

```

class PhaseContract(ABC, Generic[TInput, TOutput]):
    """
    Abstract base class for phase contracts.

    Each phase must implement:
    1. Input contract validation
    2. Output contract validation
    3. Invariant checking
    4. Phase execution logic

    This enforces the constitutional constraint that phases communicate
    ONLY through validated contracts.
    """

    def __init__(self, phase_name: str):
        """
        Initialize phase contract.

        Args:
            phase_name: Canonical name of the phase (e.g., "phase0_input_validation")
        """
        self.phase_name = phase_name
        self.invariants: list[PhaseInvariant] = []
        self.metadata: PhaseMetadata | None = None

    @abstractmethod
    def validate_input(self, input_data: Any) -> ContractValidationResult:
        """
        Validate input contract.

        Args:
            input_data: Input to validate

        Returns:
            ContractValidationResult with validation status
        """
        pass

    @abstractmethod
    def validate_output(self, output_data: Any) -> ContractValidationResult:
        """
        Validate output contract.

        Args:
            output_data: Output to validate

        Returns:
            ContractValidationResult with validation status
        """
        pass

    @abstractmethod
    async def execute(self, input_data: TInput) -> TOutput:
        """
        Execute the phase logic.

        Args:
            input_data: Validated input conforming to input contract

        Returns:
            Output conforming to output contract

        Raises:
            ValueError: If input contract validation fails
            RuntimeError: If phase execution fails
        """
        pass

    def add_invariant(
        self,
        name: str,
        description: str,
        check: callable,
        error_message: str,
    ) -> None:
        """
        Add an invariant to this phase.

        Args:
            name: Invariant name
            description: Human-readable description
            check: Function that returns bool (True = invariant holds)
            error_message: Error message if invariant fails
        """
        self.invariants.append(
            PhaseInvariant(
                name=name,
                description=description,
                check=check,
                error_message=error_message,
            )
        )

    def check_invariants(self, data: Any) -> tuple[bool, list[str]]:
        """
        Check all invariants for this phase.

        Args:
            data: Data to check invariants against

        Returns:
            Tuple of (all_passed, failed_invariant_messages)
        """
        failed_messages = []
        for inv in self.invariants:
            try:
                if not inv.check(data):
                    failed_messages.append(f"{inv.name}: {inv.error_message}")
            except Exception as e:
                failed_messages.append(f"{inv.name}: Exception during check: {e}")

        return len(failed_messages) == 0, failed_messages

    async def run(self, input_data: TInput) -> tuple[TOutput, PhaseMetadata]:
        """
        Run the complete phase with validation and invariant checking.

        This is the ONLY way to execute a phase - it enforces:
        1. Input validation
        2. Invariant checking (pre-execution if applicable)
        3. Phase execution
        4. Output validation
        5. Invariant checking (post-execution)
        6. Metadata recording

        Args:
            input_data: Input to the phase

        Returns:
            Tuple of (output_data, phase_metadata)

        Raises:
            ValueError: If contract validation fails
            RuntimeError: If invariants fail or execution fails
        """
        started_at = datetime.now(timezone.utc)
        metadata = PhaseMetadata(
            phase_name=self.phase_name,
            started_at=started_at.isoformat(),
        )

        try:
            # 1. Validate input contract
            input_validation = self.validate_input(input_data)
            if not input_validation.passed:
                error_msg = f"Input contract validation failed: {input_validation.errors}"
                metadata.error = error_msg
                metadata.success = False
                raise ValueError(error_msg)

            # 2. Execute phase
            output_data = await self.execute(input_data)

            # 3. Validate output contract
            output_validation = self.validate_output(output_data)
            if not output_validation.passed:
                error_msg = f"Output contract validation failed: {output_validation.errors}"
                metadata.error = error_msg
                metadata.success = False
                raise ValueError(error_msg)

            # 4. Check invariants
            invariants_passed, failed_invariants = self.check_invariants(output_data)
            if not invariants_passed:
                error_msg = f"Phase invariants failed: {failed_invariants}"
                metadata.error = error_msg
                metadata.success = False
                raise RuntimeError(error_msg)

            # Success
            metadata.success = True
            return output_data, metadata

        except Exception as e:
            metadata.error = str(e)
            metadata.success = False
            raise

        finally:
            finished_at = datetime.now(timezone.utc)
            metadata.finished_at = finished_at.isoformat()
            metadata.duration_ms = (
                finished_at - started_at
            ).total_seconds() * 1000
            self.metadata = metadata


@dataclass
class PhaseArtifact:
    """An artifact produced by a phase."""

    artifact_name: str
    artifact_path: Path
    sha256: str
    size_bytes: int
    created_at: str


class PhaseManifestBuilder:
    """
    Builds the phase-explicit section of the verification manifest.

    Each phase execution is recorded with:
    - Input/output contract hashes
    - Invariants checked
    - Artifacts produced
    - Timing information
    """

    def __init__(self):
        """Initialize manifest builder."""
        self.phases: dict[str, dict[str, Any]] = {}

    def record_phase(
        self,
        phase_name: str,
        metadata: PhaseMetadata,
        input_validation: ContractValidationResult,
        output_validation: ContractValidationResult,
        invariants_checked: list[str],
        artifacts: list[PhaseArtifact],
    ) -> None:
        """
        Record a phase execution in the manifest.

        Args:
            phase_name: Name of the phase
            metadata: Phase execution metadata
            input_validation: Input contract validation result
            output_validation: Output contract validation result
            invariants_checked: List of invariant names that were checked
            artifacts: List of artifacts produced by this phase
        """
        self.phases[phase_name] = {
            "status": "success" if metadata.success else "failed",
            "started_at": metadata.started_at,
            "finished_at": metadata.finished_at,
            "duration_ms": metadata.duration_ms,
            "input_contract": {
                "validation_passed": input_validation.passed,
                "errors": input_validation.errors,
                "warnings": input_validation.warnings,
            },
            "output_contract": {
                "validation_passed": output_validation.passed,
                "errors": output_validation.errors,
                "warnings": output_validation.warnings,
            },
            "invariants_checked": invariants_checked,
            "invariants_satisfied": metadata.success,
            "artifacts": [
                {
                    "name": a.artifact_name,
                    "path": str(a.artifact_path),
                    "sha256": a.sha256,
                    "size_bytes": a.size_bytes,
                }
                for a in artifacts
            ],
            "error": metadata.error,
        }

    def to_dict(self) -> dict[str, Any]:
        """
        Convert manifest to dictionary.

        Returns:
            Dictionary representation of the phase manifest
        """
        return {
            "phases": self.phases,
            "total_phases": len(self.phases),
            "successful_phases": sum(
                1 for p in self.phases.values() if p["status"] == "success"
            ),
            "failed_phases": sum(
                1 for p in self.phases.values() if p["status"] == "failed"
            ),
        }

    def save(self, output_path: Path) -> None:
        """
        Save manifest to JSON file.

        Args:
            output_path: Path to save manifest
        """
        with open(output_path, "w") as f:
            json.dump(self.to_dict(), f, indent=2)


def compute_contract_hash(contract_data: Any) -> str:
    """
    Compute SHA256 hash of a contract's data.

    Args:
        contract_data: Contract data (dict, dataclass, or Pydantic model)

    Returns:
        Hex-encoded SHA256 hash
    """
    # Convert to dict if needed
    if hasattr(contract_data, "dict"):
        # Pydantic model
        data_dict = contract_data.dict()
    elif hasattr(contract_data, "__dataclass_fields__"):
        # Dataclass
        data_dict = asdict(contract_data)
    elif isinstance(contract_data, dict):
        data_dict = contract_data
    else:
        raise TypeError(f"Cannot hash contract data of type {type(contract_data)}")

    # Serialize to JSON with sorted keys for determinism
    json_str = json.dumps(data_dict, sort_keys=True, separators=(",", ":"))
    return hashlib.sha256(json_str.encode("utf-8")).hexdigest()


__all__ = [
    "PhaseContract",
    "PhaseInvariant",
    "PhaseMetadata",
    "ContractValidationResult",
    "PhaseArtifact",
    "PhaseManifestBuilder",
    "compute_contract_hash",
]



================================================================================
FILE: src/farfan_pipeline/core/ports.py
================================================================================

"""
```

```
Port interfaces for dependency injection.

Ports define abstract interfaces for external interactions (I/O, time, environment).
These are implemented by adapters in the infrastructure layer.

This follows the Ports and Adapters (Hexagonal) architecture pattern:
- Ports are in the core layer (no dependencies)
- Adapters are in the infrastructure layer (can import anything)
- Core modules depend on ports (abstractions), not adapters (implementations)

Version: 1.0.0
"""

from datetime import datetime
from typing import Any, Protocol

from farfan_pipeline.core.analysis_port import RecommendationEnginePort


class FilePort(Protocol):
    """Port for file system operations.

    Implementations provide access to file reading and writing.
    Core modules receive a FilePort instance via dependency injection.
    """

    def read_text(self, path: str, encoding: str = "utf-8") -> str:
        """Read text from a file.

        Args:
            path: File path to read
            encoding: Text encoding (default: utf-8)

        Returns:
            File contents as string

        Raises:
            FileNotFoundError: If file does not exist
            PermissionError: If file cannot be read
        """
        ...

    def write_text(self, path: str, content: str, encoding: str = "utf-8") -> None:
        """Write text to a file.

        Args:
            path: File path to write
            content: Text content to write
            encoding: Text encoding (default: utf-8)

        Raises:
            PermissionError: If file cannot be written
        """
        ...

    def read_bytes(self, path: str) -> bytes:
        """Read bytes from a file.

        Args:
            path: File path to read

        Returns:
            File contents as bytes

        Raises:
            FileNotFoundError: If file does not exist
            PermissionError: If file cannot be read
        """
        ...

    def write_bytes(self, path: str, content: bytes) -> None:
        """Write bytes to a file.

        Args:
            path: File path to write
            content: Bytes content to write

        Raises:
            PermissionError: If file cannot be written
        """
        ...

    def exists(self, path: str) -> bool:
        """Check if a file or directory exists.

        Args:
            path: Path to check

        Returns:
            True if path exists, False otherwise
        """
        ...

    def mkdir(self, path: str, parents: bool = False, exist_ok: bool = False) -> None:
        """Create a directory.

        Args:
            path: Directory path to create
            parents: Create parent directories if needed
            exist_ok: Don’t raise error if directory exists

        Raises:
            FileExistsError: If directory exists and exist_ok is False
        """
        ...


class JsonPort(Protocol):
    """Port for JSON serialization/deserialization.

    Separates JSON operations from file I/O for better composability.
    """

    def loads(self, text: str) -> Any:
        """Parse JSON from string.

        Args:
            text: JSON string

        Returns:
            Parsed Python object

        Raises:
            ValueError: If JSON is invalid
        """
        ...

    def dumps(self, obj: Any, indent: int | None = None) -> str:
        """Serialize object to JSON string.

        Args:
            obj: Python object to serialize
            indent: Indentation spaces (None for compact)

        Returns:
            JSON string

        Raises:
            TypeError: If object is not serializable
        """
        ...


class EnvPort(Protocol):
    """Port for environment variable access.

    Allows core modules to access configuration without direct os.environ coupling.
    """

    def get(self, key: str, default: str | None = None) -> str | None:
        """Get environment variable.

        Args:
            key: Environment variable name
            default: Default value if not set

        Returns:
            Environment variable value or default
        """
        ...

    def get_required(self, key: str) -> str:
        """Get required environment variable.

        Args:
            key: Environment variable name

        Returns:
            Environment variable value

        Raises:
            ValueError: If environment variable is not set
        """
        ...

    def get_bool(self, key: str, default: bool = False) -> bool:
        """Get environment variable as boolean.

        Args:
            key: Environment variable name
            default: Default value if not set

        Returns:
            Boolean value (true/false/yes/no/1/0)
        """
        ...


class ClockPort(Protocol):
    """Port for time operations.

    Allows core modules to get current time without direct datetime.now() calls.
    Enables time manipulation in tests.
    """

    def now(self) -> datetime:
        """Get current datetime.

        Returns:
            Current datetime
        """
        ...

    def utcnow(self) -> datetime:
        """Get current UTC datetime.

        Returns:
            Current UTC datetime
        """
        ...


class LogPort(Protocol):
    """Port for logging operations.

    Allows core modules to log without coupling to specific logging framework.
    """

    def debug(self, message: str, **kwargs: Any) -> None:
        """Log debug message."""
        ...

    def info(self, message: str, **kwargs: Any) -> None:
        """Log info message."""
        ...

    def warning(self, message: str, **kwargs: Any) -> None:
        """Log warning message."""
        ...

    def error(self, message: str, **kwargs: Any) -> None:
        """Log error message."""
        ...


class PortCPPIngest(Protocol):
    """Port for CPP (Canon Policy Package) ingestion.

    Ingests documents and produces Canon Policy Packages with complete provenance.
    """

    def ingest(self, input_uri: str) -> Any:
        """Ingest document from URI and produce Canon Policy Package.

        Args:
            input_uri: URI to document (file://, http://, etc.)

        Returns:
            CanonPolicyPackage with complete chunk graph and metadata

        Requires:
            - Valid input URI
            - Accessible document at URI

        Ensures:
            - chunk_graph is not None
            - policy_manifest is not None
            - provenance_completeness == 1.0
        """
        ...


class PortCPPAdapter(Protocol):
    """Port for CPP to PreprocessedDocument adaptation.

    Converts Canon Policy Package to orchestrator’s PreprocessedDocument format.

    Note: CPP is the legacy name. Use PortSPCAdapter for new code.
    """

    def to_preprocessed_document(self, cpp: Any, document_id: str) -> Any:
        """Convert CPP to PreprocessedDocument.

        Args:
            cpp: Canon Policy Package from ingestion
            document_id: Unique document identifier

        Returns:
            PreprocessedDocument for orchestrator

        Requires:
            - cpp with valid chunk_graph
            - cpp.policy_manifest exists
            - document_id is non-empty

        Ensures:
            - sentence_metadata is not empty
            - resolution_index is consistent
            - provenance_completeness == 1.0
        """
        ...


class PortSPCAdapter(Protocol):
    """Port for SPC (Smart Policy Chunks) to PreprocessedDocument adaptation.

    Converts Smart Policy Chunks to orchestrator’s PreprocessedDocument format.
    This is the preferred terminology for new code.
    """

    def to_preprocessed_document(self, spc: Any, document_id: str) -> Any:
        """Convert SPC to PreprocessedDocument.

        Args:
            spc: Smart Policy Chunks package from ingestion
            document_id: Unique document identifier

        Returns:
            PreprocessedDocument for orchestrator

        Requires:
            - spc with valid chunk_graph
            - spc.policy_manifest exists
            - document_id is non-empty

        Ensures:
            - sentence_metadata is not empty
            - resolution_index is consistent
            - provenance_completeness == 1.0
        """
        ...


class PortSignalsClient(Protocol):
    """Port for fetching strategic signals.

    Retrieves policy-aware signals from memory or HTTP sources.
    Sematics: None return = 304 Not Modified or circuit breaker open.
    """

    def fetch(self, policy_area: str) -> Any | None:
        """Fetch signals for policy area.

        Args:
            policy_area: Policy domain (fiscal, salud, ambiente, etc.)

        Returns:
            SignalPack if available, None if 304/breaker open

        Requires:
            - policy_area is valid PolicyArea literal

        Ensures:
            - If not None, returns valid SignalPack with version
            - None is justified (304 or breaker state)
        """
        ...


class PortSignalsRegistry(Protocol):
    """Port for signal registry with TTL and LRU.

    Manages in-memory cache of strategic signals with expiration.
    """

    def put(self, pack: Any) -> None:
        """Store signal pack in registry.

        Args:
            pack: SignalPack to store

        Requires:
            - pack is valid SignalPack
            - pack.version is present
        """
        ...

    def get(self, policy_area: str) -> dict[str, Any] | None:
        """Retrieve signals for policy area.

        Args:
            policy_area: Policy domain

        Returns:
            Signal data if cached and not expired, None otherwise
        """
        ...

    def fingerprint(self) -> str:
        """Compute registry fingerprint for drift detection.

        Returns:
            BLAKE3 hash of current registry state
        """
        ...


class PortArgRouter(Protocol):
    """Port for argument routing and validation.

    Routes method calls with strict parameter validation.
    """

    def route(
        self, class_name: str, method_name: str, payload: dict[str, Any]
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """Route method call to (args, kwargs).

        Args:
            class_name: Target class name
            method_name: Target method name
            payload: Input parameters

        Returns:
            Tuple of (args, kwargs) for method call

        Requires:
            - class_name exists in registry
            - method_name exists on class
            - method signature is known or has **kwargs

        Ensures:
            - No silent parameter drops
            - All required args present
            - No unexpected kwargs (unless **kwargs in signature)
        """
        ...


class PortExecutor(Protocol):
    """Port for executing methods with configuration.

    Executes methods with injected executor config and signals.
    """

    def run(self, prompt: str, overrides: Any | None = None) -> Any:
        """Execute with prompt and optional config overrides.

        Args:
            prompt: Execution prompt/input
            overrides: Optional ExecutorConfig overrides

        Returns:
            Result with metadata including used_signals

        Requires:
            - ExecutorConfig is injected
            - SignalRegistry is available

        Ensures:
            - Result includes used_signals metadata
            - Execution is deterministic if seed is set
        """
        ...


class PortAggregate(Protocol):
    """Port for aggregating enriched chunks.

    Aggregates processed chunks into PyArrow tables.
    """

    def aggregate(self, enriched_chunks: list[dict[str, Any]]) -> Any:
        """Aggregate enriched chunks to PyArrow table.

        Args:
            enriched_chunks: List of enriched chunk dictionaries

        Returns:
            PyArrow Table with aggregated data

        Requires:
            - enriched_chunks has required fields
            - All chunks have consistent schema

        Ensures:
            - Returns valid pa.Table
            - All required columns present
        """
        ...


class PortScore(Protocol):
    """Port for scoring features.

    Computes scores from feature tables with specified metrics.
    """

    def score(self, features: Any, metrics: list[str]) -> Any:
        """Score features using specified metrics.

        Args:
            features: PyArrow Table with features
            metrics: List of metric names to compute

        Returns:
            Polars DataFrame with scores

        Requires:
            - features is valid pa.Table
            - metrics are declared and implemented
            - Required columns present in features

        Ensures:
            - Returns valid pl.DataFrame
            - All requested metrics computed
        """
        ...


class PortReport(Protocol):
    """Port for generating reports.

    Generates output reports from scores and manifest.
    """

    def report(self, scores: Any, manifest: Any) -> dict[str, str]:
        """Generate reports from scores and manifest.

        Args:
            scores: Polars DataFrame with computed scores
            manifest: Document manifest with metadata

        Returns:
            Dictionary mapping report name to output URI

        Requires:
            - scores is valid pl.DataFrame
            - manifest has required metadata

        Ensures:
            - All declared reports generated
            - URIs are accessible
        """
        ...


class PortDocumentLoader(Protocol):
    """Port for loading documents from various formats.

    Loads policy documents from PDF, DOCX, and other formats.
    """

    def load_pdf(self, path: str) -> str:
        """Load text from PDF file.

        Args:
            path: Path to PDF file

        Returns:
            Extracted text content

        Raises:
            FileNotFoundError: If file does not exist
        """
        ...

    def load_docx(self, path: str) -> str:
        """Load text from DOCX file.

        Args:
            path: Path to DOCX file

        Returns:
            Extracted text content

        Raises:
            FileNotFoundError: If file does not exist
        """
        ...
```

```python


class PortMunicipalOntology(Protocol):
    """Port for municipal policy ontology.

    Provides domain knowledge for policy analysis.
    """

    def __init__(self) -> None:
        """Initialize ontology."""
        ...


class PortSemanticAnalyzer(Protocol):
    """Port for semantic analysis of policy text.

    Extracts semantic features from policy documents.
    """

    def extract_semantic_cube(self, sentences: list[str]) -> dict[str, Any]:
        """Extract semantic features from sentences.

        Args:
            sentences: List of policy text sentences

        Returns:
            Dictionary containing semantic analysis results

        Requires:
            - sentences is non-empty list
        """
        ...


class PortPerformanceAnalyzer(Protocol):
    """Port for performance analysis.

    Analyzes performance metrics and indicators in policy text.
    """

    def analyze_performance(self, text: str, semantic_data: dict[str, Any]) -> dict[str, Any]:
        """Analyze performance indicators and metrics.

        Args:
            text: Policy text to analyze
            semantic_data: Semantic features from semantic analyzer

        Returns:
            Dictionary containing performance analysis results
        """
        ...


class PortContradictionDetector(Protocol):
    """Port for detecting contradictions in policy text.

    Identifies logical contradictions and inconsistencies.
    """

    def detect(
        self,
        text: str,
        plan_name: str,
        dimension: Any,
    ) -> dict[str, Any]:
        """Detect contradictions in policy text.

        Args:
            text: Policy text to analyze
            plan_name: Name of policy plan
            dimension: Policy dimension to analyze

        Returns:
            Dictionary containing contradiction analysis results
        """
        ...

    def _extract_policy_statements(self, text: str, dimension: Any) -> list[Any]:
        """Extract policy statements for analysis.

        Args:
            text: Policy text
            dimension: Policy dimension

        Returns:
            List of extracted policy statements
        """
        ...


class PortTemporalLogicVerifier(Protocol):
    """Port for temporal logic verification.

    Verifies temporal consistency in policy statements.
    """

    def verify_temporal_consistency(self, statements: list[Any]) -> tuple[bool, list[dict[str, Any]]]:
        """Verify temporal consistency of statements.

        Args:
            statements: List of policy statements to verify

        Returns:
            Tuple of (is_consistent, list_of_conflicts)
        """
        ...


class PortBayesianConfidenceCalculator(Protocol):
    """Port for Bayesian confidence calculation.

    Calculates posterior confidence scores using Bayesian methods.
    """

    def calculate_posterior(
        self,
        evidence_strength: float,
        observations: int,
        domain_weight: float = 1.0,
    ) -> float:
        """Calculate posterior confidence score.

        Args:
            evidence_strength: Strength of evidence (0-1)
            observations: Number of observations
            domain_weight: Domain-specific weight

        Returns:
            Posterior probability (0-1)
        """
        ...


class PortMunicipalAnalyzer(Protocol):
    """Port for municipal policy analysis.

    Performs comprehensive municipal policy analysis.
    """

    def _generate_summary(
        self,
        text: str,
        evidence_by_dimension: dict[str, Any],
        dimension_scores: dict[str, float],
    ) -> dict[str, Any]:
        """Generate analysis summary.

        Args:
            text: Policy text
            evidence_by_dimension: Evidence organized by dimension
            dimension_scores: Scores for each dimension

        Returns:
            Dictionary containing analysis summary
        """
        ...


__all__ = [
    'FilePort',
    'JsonPort',
    'EnvPort',
    'ClockPort',
    'LogPort',
    'PortCPPIngest',
    'PortCPPAdapter',
    'PortSPCAdapter',
    'PortSignalsClient',
    'PortSignalsRegistry',
    'PortArgRouter',
    'PortExecutor',
    'PortAggregate',
    'PortScore',
    'PortReport',
    'PortDocumentLoader',
    'PortMunicipalOntology',
    'PortSemanticAnalyzer',
    'PortPerformanceAnalyzer',
    'PortContradictionDetector',
    'PortTemporalLogicVerifier',
    'PortBayesianConfidenceCalculator',
    'PortMunicipalAnalyzer',
    'RecommendationEnginePort',
]



================================================================================
FILE: src/farfan_pipeline/core/runtime_config.py
================================================================================

"""
Global runtime configuration system for F.A.R.F.A.N.

This module provides runtime mode enforcement (PROD/DEV/EXPLORATORY) with strict
fallback policies, configuration validation, and environment variable parsing.

FALLBACK CATEGORIZATION AND ASSESSMENT:

CATEGORY A (CRITICAL - System Integrity):
    Variables: ALLOW_CONTRADICTION_FALLBACK, ALLOW_VALIDATOR_DISABLE, ALLOW_EXECUTION_ESTIMATES
    Assessment: These indicate missing CRITICAL components. In PROD, the system MUST fail fast
    to prevent incorrect analysis results. No fallback is acceptable.

CATEGORY B (QUALITY - Quality Degradation):
    Variables: ALLOW_NETWORKX_FALLBACK, ALLOW_SPACY_FALLBACK
    Assessment: These degrade output quality but don't invalidate core analysis. Allowed in
    PROD with explicit flag and warnings logged. Results remain scientifically valid but less rich.

CATEGORY C (DEVELOPMENT - Development Convenience):
    Variables: ALLOW_DEV_INGESTION_FALLBACKS, ALLOW_AGGREGATION_DEFAULTS, ALLOW_MISSING_BASE_WEIGHTS
    Assessment: STRICTLY FORBIDDEN in PROD. These exist only for development/testing to avoid
    infrastructure dependencies. Using these in PROD invalidates results.

CATEGORY D (OPERATIONAL - Operational Flexibility):
    Variables: ALLOW_HASH_FALLBACK, ALLOW_PDFPLUMBER_FALLBACK
    Assessment: Safe fallbacks maintaining correctness with different implementation strategies.
    Generally allowed as they don't affect scientific validity.

Environment Variables:
    SAAAAAA_RUNTIME_MODE: Runtime mode (prod/dev/exploratory), default: prod

    # Category A - Critical System Integrity
    ALLOW_CONTRADICTION_FALLBACK: Allow contradiction detection fallback, default: false
    ALLOW_VALIDATOR_DISABLE: Allow wiring validator disabling, default: false
    ALLOW_EXECUTION_ESTIMATES: Allow execution metric estimation, default: false

    # Category B - Quality Degradation
    ALLOW_NETWORKX_FALLBACK: Allow NetworkX unavailability, default: false
    ALLOW_SPACY_FALLBACK: Allow spaCy model fallback, default: false

    # Category C - Development Convenience (FORBIDDEN in PROD)
    ALLOW_DEV_INGESTION_FALLBACKS: Allow dev ingestion fallbacks, default: false
    ALLOW_AGGREGATION_DEFAULTS: Allow aggregation defaults, default: false
    ALLOW_MISSING_BASE_WEIGHTS: Allow missing _base_weights in calibration, default: false

    # Category D - Operational Flexibility
    ALLOW_HASH_FALLBACK: Allow hash algorithm fallback, default: true
    ALLOW_PDFPLUMBER_FALLBACK: Allow pdfplumber unavailability, default: false

    # Calibration and Quality Controls
    STRICT_CALIBRATION: Require complete calibration files, default: true

    # Model and Processing Configuration
    PREFERRED_SPACY_MODEL: Preferred spaCy model, default: es_core_news_lg
    PREFERRED_EMBEDDING_MODEL: Preferred embedding model, default: sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2

    # Path Configuration
    SAAAAAA_PROJECT_ROOT: Project root override
    SAAAAAA_DATA_DIR: Data directory override
    SAAAAAA_OUTPUT_DIR: Output directory override
    SAAAAAA_CACHE_DIR: Cache directory override
    SAAAAAA_LOGS_DIR: Logs directory override

    # External Dependencies
    HF_ONLINE: Allow HuggingFace online access (0 or 1), default: 0

    # Processing Limits
    EXPECTED_QUESTION_COUNT: Expected question count, default: 305
    EXPECTED_METHOD_COUNT: Expected method count, default: 416
    PHASE_TIMEOUT_SECONDS: Phase timeout in seconds, default: 300
    MAX_WORKERS: Maximum worker threads, default: 4
    BATCH_SIZE: Batch size for processing, default: 100

Example:
    >>> config = RuntimeConfig.from_env()
    >>> if config.mode == RuntimeMode.PROD:
    ...     assert not config.allow_dev_ingestion_fallbacks
"""

import os
from dataclasses import dataclass
from enum import Enum
from typing import ClassVar, Optional


class RuntimeMode(Enum):
    """Runtime execution mode with different strictness levels."""

    PROD = "prod"
    """Production mode: strict enforcement, no fallbacks unless explicitly allowed."""

    DEV = "dev"
    """Development mode: permissive with flags, allows controlled degradation."""

    EXPLORATORY = "exploratory"
    """Exploratory mode: maximum flexibility for research and experimentation."""


class FallbackCategory(Enum):
    """Categorization of fallback types by impact."""

    CRITICAL = "critical"
    """Category A: System integrity - failures indicate missing critical dependencies."""

    QUALITY = "quality"
    """Category B: Quality degradation - system continues with reduced quality."""

    DEVELOPMENT = "development"
    """Category C: Development convenience - only allowed in DEV/EXPLORATORY."""

    OPERATIONAL = "operational"
    """Category D: Operational flexibility - safe fallbacks for operational concerns."""


class ConfigurationError(Exception):
    """Raised when runtime configuration is invalid or contains illegal combinations."""

    def __init__(self, message: str, illegal_combo: str | None = None) -> None:
        self.illegal_combo = illegal_combo
        super().__init__(message)


@dataclass(frozen=True)
class RuntimeConfig:
    """
    Immutable runtime configuration parsed from environment variables.

    This configuration controls system behavior across all components, enforcing
    strict policies in PROD mode and allowing controlled degradation in DEV/EXPLORATORY.

    Attributes:
        mode: Runtime execution mode

        # Category A - Critical System Integrity
        allow_contradiction_fallback: Allow fallback when contradiction module unavailable
        allow_validator_disable: Allow disabling wiring validator
        allow_execution_estimates: Allow execution metric estimation

        # Category B - Quality Degradation
        allow_networkx_fallback: Allow NetworkX unavailability
        allow_spacy_fallback: Allow spaCy model fallback

        # Category C - Development Convenience
        allow_dev_ingestion_fallbacks: Allow development ingestion fallbacks
        allow_aggregation_defaults: Allow aggregation default values
        allow_missing_base_weights: Allow missing _base_weights in calibration

        # Category D - Operational Flexibility
        allow_hash_fallback: Allow hash algorithm fallback
        allow_pdfplumber_fallback: Allow pdfplumber unavailability

        # Calibration
        strict_calibration: Require complete calibration files with _base_weights

        # Model Configuration
        preferred_spacy_model: Preferred spaCy model name
        preferred_embedding_model: Preferred embedding model name

        # Path Configuration
        project_root_override: Project root path override
        data_dir_override: Data directory override
        output_dir_override: Output directory override
        cache_dir_override: Cache directory override
        logs_dir_override: Logs directory override

        # External Dependencies
        hf_online: Allow HuggingFace online access

        # Processing Configuration
        expected_question_count: Expected question count for validation
        expected_method_count: Expected method count for validation
        phase_timeout_seconds: Phase timeout in seconds
        max_workers: Maximum worker threads
        batch_size: Batch size for processing
    """

    mode: RuntimeMode

    # Category A - Critical
    allow_contradiction_fallback: bool
    allow_validator_disable: bool
    allow_execution_estimates: bool

    # Category B - Quality
    allow_networkx_fallback: bool
    allow_spacy_fallback: bool

    # Category C - Development
    allow_dev_ingestion_fallbacks: bool
    allow_aggregation_defaults: bool
    allow_missing_base_weights: bool

    # Category D - Operational
    allow_hash_fallback: bool
    allow_pdfplumber_fallback: bool

    # Calibration
    strict_calibration: bool

    # Model Configuration
    preferred_spacy_model: str
    preferred_embedding_model: str

    # Path Configuration
    project_root_override: Optional[str]
    data_dir_override: Optional[str]
    output_dir_override: Optional[str]
    cache_dir_override: Optional[str]
    logs_dir_override: Optional[str]

    # External Dependencies
    hf_online: bool

    # Processing Configuration
    expected_question_count: int
    expected_method_count: int
    phase_timeout_seconds: int
    max_workers: int
    batch_size: int

    # Illegal combinations in PROD mode
    _PROD_ILLEGAL_COMBOS: ClassVar[dict[str, tuple[str, FallbackCategory]]] = {
        "ALLOW_DEV_INGESTION_FALLBACKS": (
            "Development ingestion fallbacks not allowed in PROD - they bypass quality gates",
            FallbackCategory.DEVELOPMENT
        ),
        "ALLOW_EXECUTION_ESTIMATES": (
            "Execution metric estimation not allowed in PROD - actual measurements required",
            FallbackCategory.CRITICAL
        ),
        "ALLOW_AGGREGATION_DEFAULTS": (
            "Aggregation defaults not allowed in PROD - explicit calibration required",
            FallbackCategory.DEVELOPMENT
        ),
        "ALLOW_MISSING_BASE_WEIGHTS": (
            "Missing base weights not allowed in PROD - complete calibration required",
            FallbackCategory.DEVELOPMENT
        ),
    }

    @classmethod
    def from_env(cls) -> "RuntimeConfig":
        """
        Parse runtime configuration from environment variables.

        Returns:
            RuntimeConfig: Validated configuration instance

        Raises:
            ConfigurationError: If configuration is invalid or contains illegal combinations

        Example:
            >>> os.environ['SAAAAAA_RUNTIME_MODE'] = 'prod'
            >>> config = RuntimeConfig.from_env()
            >>> assert config.mode == RuntimeMode.PROD
        """
        # Parse runtime mode
        mode_str = os.getenv("SAAAAAA_RUNTIME_MODE", "prod").lower()
        try:
            mode = RuntimeMode(mode_str)
        except ValueError as e:
            raise ConfigurationError(
                f"Invalid SAAAAAA_RUNTIME_MODE: {mode_str}. "
                f"Must be one of: {’, ’.join(m.value for m in RuntimeMode)}"
            ) from e

        # Parse Category A - Critical Fallbacks
        allow_contradiction_fallback = _parse_bool_env("ALLOW_CONTRADICTION_FALLBACK", False)
        allow_validator_disable = _parse_bool_env("ALLOW_VALIDATOR_DISABLE", False)
        allow_execution_estimates = _parse_bool_env("ALLOW_EXECUTION_ESTIMATES", False)

        # Parse Category B - Quality Fallbacks
        allow_networkx_fallback = _parse_bool_env("ALLOW_NETWORKX_FALLBACK", False)
        allow_spacy_fallback = _parse_bool_env("ALLOW_SPACY_FALLBACK", False)

        # Parse Category C - Development Fallbacks
        allow_dev_ingestion_fallbacks = _parse_bool_env("ALLOW_DEV_INGESTION_FALLBACKS", False)
        allow_aggregation_defaults = _parse_bool_env("ALLOW_AGGREGATION_DEFAULTS", False)
        allow_missing_base_weights = _parse_bool_env("ALLOW_MISSING_BASE_WEIGHTS", False)

        # Parse Category D - Operational Fallbacks
        allow_hash_fallback = _parse_bool_env("ALLOW_HASH_FALLBACK", True)
        allow_pdfplumber_fallback = _parse_bool_env("ALLOW_PDFPLUMBER_FALLBACK", False)

        # Parse calibration config
        strict_calibration = _parse_bool_env("STRICT_CALIBRATION", True)

        # Parse model configuration
        preferred_spacy_model = os.getenv("PREFERRED_SPACY_MODEL", "es_core_news_lg")
        preferred_embedding_model = os.getenv(
            "PREFERRED_EMBEDDING_MODEL",
            "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
        )

        # Parse path configuration
        project_root_override = os.getenv("SAAAAAA_PROJECT_ROOT")
        data_dir_override = os.getenv("SAAAAAA_DATA_DIR")
        output_dir_override = os.getenv("SAAAAAA_OUTPUT_DIR")
        cache_dir_override = os.getenv("SAAAAAA_CACHE_DIR")
        logs_dir_override = os.getenv("SAAAAAA_LOGS_DIR")

        # Parse external dependencies
        hf_online = os.getenv("HF_ONLINE", "0") == "1"

        # Parse processing configuration
        expected_question_count = _parse_int_env("EXPECTED_QUESTION_COUNT", 305)
        expected_method_count = _parse_int_env("EXPECTED_METHOD_COUNT", 416)
        phase_timeout_seconds = _parse_int_env("PHASE_TIMEOUT_SECONDS", 300)
        max_workers = _parse_int_env("MAX_WORKERS", 4)
        batch_size = _parse_int_env("BATCH_SIZE", 100)

        # Create config instance
        config = cls(
            mode=mode,
            allow_contradiction_fallback=allow_contradiction_fallback,
            allow_validator_disable=allow_validator_disable,
            allow_execution_estimates=allow_execution_estimates,
            allow_networkx_fallback=allow_networkx_fallback,
            allow_spacy_fallback=allow_spacy_fallback,
            allow_dev_ingestion_fallbacks=allow_dev_ingestion_fallbacks,
            allow_aggregation_defaults=allow_aggregation_defaults,
            allow_missing_base_weights=allow_missing_base_weights,
            allow_hash_fallback=allow_hash_fallback,
            allow_pdfplumber_fallback=allow_pdfplumber_fallback,
            strict_calibration=strict_calibration,
            preferred_spacy_model=preferred_spacy_model,
            preferred_embedding_model=preferred_embedding_model,
            project_root_override=project_root_override,
            data_dir_override=data_dir_override,
            output_dir_override=output_dir_override,
            cache_dir_override=cache_dir_override,
            logs_dir_override=logs_dir_override,
            hf_online=hf_online,
            expected_question_count=expected_question_count,
            expected_method_count=expected_method_count,
            phase_timeout_seconds=phase_timeout_seconds,
            max_workers=max_workers,
            batch_size=batch_size,
        )

        # Validate configuration
        config._validate()

        return config

    def _validate(self) -> None:
        """
        Validate configuration for illegal combinations.

        In PROD mode, certain ALLOW_* flags are prohibited to ensure strict behavior.

        Raises:
            ConfigurationError: If illegal combination detected
        """
        if self.mode != RuntimeMode.PROD:
            return  # DEV/EXPLORATORY modes allow all combinations

        # Check for illegal PROD combinations
        violations = []

        if self.allow_dev_ingestion_fallbacks:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_DEV_INGESTION_FALLBACKS"]
            violations.append(
                f"PROD + ALLOW_DEV_INGESTION_FALLBACKS=true: {msg} [Category: {cat.value}]"
            )

        if self.allow_execution_estimates:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_EXECUTION_ESTIMATES"]
            violations.append(
                f"PROD + ALLOW_EXECUTION_ESTIMATES=true: {msg} [Category: {cat.value}]"
            )

        if self.allow_aggregation_defaults:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_AGGREGATION_DEFAULTS"]
            violations.append(
                f"PROD + ALLOW_AGGREGATION_DEFAULTS=true: {msg} [Category: {cat.value}]"
            )

        if self.allow_missing_base_weights:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_MISSING_BASE_WEIGHTS"]
            violations.append(
                f"PROD + ALLOW_MISSING_BASE_WEIGHTS=true: {msg} [Category: {cat.value}]"
            )

        if violations:
            raise ConfigurationError(
                "Illegal configuration combinations detected:\n" + "\n".join(f"  - {v}" for v in violations),
                illegal_combo="; ".join(violations)
            )

    def is_strict_mode(self) -> bool:
        """Check if running in strict mode (PROD with no fallbacks allowed)."""
        return (
            self.mode == RuntimeMode.PROD
            and not self.allow_contradiction_fallback
            and not self.allow_validator_disable
        )

    def get_fallback_summary(self) -> dict[str, dict[str, bool]]:
        """
        Get summary of all fallback configurations grouped by category.

        Returns:
            Dictionary mapping category names to flag dictionaries
        """
        return {
            "critical": {
                "contradiction_fallback": self.allow_contradiction_fallback,
                "validator_disable": self.allow_validator_disable,
                "execution_estimates": self.allow_execution_estimates,
            },
            "quality": {
                "networkx_fallback": self.allow_networkx_fallback,
                "spacy_fallback": self.allow_spacy_fallback,
            },
            "development": {
                "dev_ingestion_fallbacks": self.allow_dev_ingestion_fallbacks,
                "aggregation_defaults": self.allow_aggregation_defaults,
                "missing_base_weights": self.allow_missing_base_weights,
            },
            "operational": {
                "hash_fallback": self.allow_hash_fallback,
                "pdfplumber_fallback": self.allow_pdfplumber_fallback,
            },
        }

    def __repr__(self) -> str:
        """String representation showing mode and key flags."""
        flags = []
        if self.allow_contradiction_fallback:
            flags.append("contradiction_fallback")
        if self.allow_validator_disable:
            flags.append("validator_disable")
        if self.allow_execution_estimates:
            flags.append("execution_estimates")
        if self.allow_networkx_fallback:
            flags.append("networkx_fallback")
        if self.allow_spacy_fallback:
            flags.append("spacy_fallback")
        if self.allow_dev_ingestion_fallbacks:
            flags.append("dev_ingestion_fallbacks")
        if self.allow_aggregation_defaults:
            flags.append("aggregation_defaults")
        if self.allow_missing_base_weights:
            flags.append("missing_base_weights")
        if not self.strict_calibration:
            flags.append("relaxed_calibration")

        flags_str = f", flags={flags}" if flags else ""
        return f"RuntimeConfig(mode={self.mode.value}{flags_str})"


def _parse_bool_env(var_name: str, default: bool) -> bool:
    """
    Parse boolean environment variable with case-insensitive handling.

    Args:
        var_name: Environment variable name
        default: Default value if not set

    Returns:
        Parsed boolean value

    Raises:
        ConfigurationError: If value is not a valid boolean
    """
    value = os.getenv(var_name)
    if value is None:
        return default

    value_lower = value.lower()
    if value_lower in ("true", "1", "yes", "on"):
        return True
    elif value_lower in ("false", "0", "no", "off"):
        return False
    else:
        raise ConfigurationError(
            f"Invalid boolean value for {var_name}: {value}. "
            f"Must be one of: true/false, 1/0, yes/no, on/off"
        )

```

```

def _parse_int_env(var_name: str, default: int) -> int:
    """
    Parse integer environment variable with validation.

    Args:
        var_name: Environment variable name
        default: Default value if not set

    Returns:
        Parsed integer value

    Raises:
        ConfigurationError: If value is not a valid integer
    """
    value = os.getenv(var_name)
    if value is None:
        return default

    try:
        return int(value)
    except ValueError:
        raise ConfigurationError(
            f"Invalid integer value for {var_name}: {value}. "
            f"Must be a valid integer."
        )


# Global singleton instance (lazy-initialized)
_global_config: RuntimeConfig | None = None


def get_runtime_config() -> RuntimeConfig:
    """
    Get global runtime configuration instance (lazy-initialized).

    Returns:
        RuntimeConfig: Global configuration instance

    Note:
        This is initialized once on first call. For testing, use from_env() directly.
    """
    global _global_config
    if _global_config is None:
        _global_config = RuntimeConfig.from_env()
    return _global_config


def reset_runtime_config() -> None:
    """
    Reset global runtime configuration (for testing only).

    Warning:
        This should only be used in tests. Production code should never reset config.
    """
    global _global_config
    _global_config = None



================================================================================
FILE: src/farfan_pipeline/core/types.py
================================================================================

"""
Core type definitions shared across layers.

This module contains types that need to be referenced by both core and analysis
layers without creating circular dependencies.
"""

from __future__ import annotations

import re
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from pathlib import Path
from typing import Any, Literal

__all__ = [
    "CategoriaCausal",
    "ChunkData",
    "PreprocessedDocument",
    "Provenance",
]


class CategoriaCausal(Enum):
    """
    JerarquÃa axiomÃ¡tica de categorÃas causales en una teorÃa de cambio.
    El orden numÃ©rico impone la secuencia lÃ³gica obligatoria.

    Originally from farfan_core.analysis.teoria_cambio, moved here to break
    architectural dependency (core should not import from analysis).
    """

    INSUMOS = 1
    ACTIVIDADES = 2
    PRODUCTOS = 3
    RESULTADOS = 4
    CAUSALIDAD = 5


@dataclass(frozen=True)
class Provenance:
    """Provenance metadata for a chunk."""

    page_number: int
    section_header: str | None = None
    bbox: tuple[float, float, float, float] | None = None
    span_in_page: tuple[int, int] | None = None
    source_file: str | None = None


@dataclass(frozen=True)
class ChunkData:
    """Single semantic chunk from SPC (Smart Policy Chunks).

    Preserves chunk structure and metadata from the ingestion pipeline,
    enabling chunk-aware executor routing and scoped processing.
    """

    id: int
    text: str
    chunk_type: Literal[
        "diagnostic", "activity", "indicator", "resource", "temporal", "entity"
    ]
    sentences: list[int]
    tables: list[int]
    start_pos: int
    end_pos: int
    confidence: float
    chunk_id: str | None = None
    edges_out: list[int] = field(default_factory=list)
    edges_in: list[int] = field(default_factory=list)
    policy_area_id: str | None = None
    dimension_id: str | None = None
    provenance: Provenance | None = None
    expected_elements: list[dict[str, Any]] = field(default_factory=list)
    document_position: tuple[int, int] | None = None

    _CHUNK_ID_PATTERN = re.compile(r"^PA(0[1-9]|10)-DIM(0[1-6])$")

    def __post_init__(self) -> None:
        """Validate chunk_id presence and format (PA{01-10}-DIM{01-06}) and new fields.

        Enforces Phase 1 output contract:
        - Non-empty text content
        - Valid chunk_id format (PA01-PA10, DIM01-DIM06)
        - Consistency between chunk_id and policy_area_id/dimension_id
        - Valid expected_elements structure
        - Valid document_position range
        """
        import logging

        logger = logging.getLogger(__name__)

        if not isinstance(self.text, str):
            raise ValueError(
                f"ChunkData text must be a string, got {type(self.text).__name__}"
            )

        if not self.text or not self.text.strip():
            raise ValueError(
                "ChunkData text cannot be empty or whitespace-only. "
                "Phase 1 must populate all chunks with non-empty text content."
            )

        self._validate_expected_elements()
        self._validate_document_position(logger)

        chunk_id = self.chunk_id
        if chunk_id is None:
            if self.policy_area_id and self.dimension_id:
                chunk_id = f"{self.policy_area_id}-{self.dimension_id}"
                object.__setattr__(self, "chunk_id", chunk_id)
            else:
                raise ValueError(
                    "chunk_id is required and must follow format PA{01-10}-DIM{01-06}. "
                    "Provide chunk_id explicitly or set both policy_area_id and dimension_id "
                    "to derive it. Phase 1 must populate these fields for all chunks."
                )

        if not self._CHUNK_ID_PATTERN.match(chunk_id):
            raise ValueError(
                f"Invalid chunk_id '{chunk_id}'. Expected format PA{{01-10}}-DIM{{01-06}}. "
                "Valid examples: 'PA01-DIM01', 'PA10-DIM06'. "
                "Phase 1 must generate chunk_id values that match this pattern."
            )

        match = self._CHUNK_ID_PATTERN.match(chunk_id)
        if match:
            pa_code = f"PA{match.group(1)}"
            dim_code = f"DIM{match.group(2)}"
            if self.policy_area_id and self.policy_area_id != pa_code:
                raise ValueError(
                    f"chunk_id '{chunk_id}' mismatches policy_area_id '{self.policy_area_id}'. "
                    f"Expected policy_area_id to be '{pa_code}' based on chunk_id. "
                    "Phase 1 must ensure consistency between chunk_id and metadata fields."
                )
            if self.dimension_id and self.dimension_id != dim_code:
                raise ValueError(
                    f"chunk_id '{chunk_id}' mismatches dimension_id '{self.dimension_id}'. "
                    f"Expected dimension_id to be '{dim_code}' based on chunk_id. "
                    "Phase 1 must ensure consistency between chunk_id and metadata fields."
                )

    def _validate_expected_elements(self) -> None:
        """Validate expected_elements field structure."""
        if not isinstance(self.expected_elements, list):
            raise ValueError(
                f"expected_elements must be a list, got {type(self.expected_elements).__name__}"
            )

        for idx, element in enumerate(self.expected_elements):
            if not isinstance(element, dict):
                raise ValueError(
                    f"expected_elements[{idx}] must be a dict, got {type(element).__name__}"
                )

            if "type" not in element:
                raise ValueError(
                    f"expected_elements[{idx}] missing required 'type' key"
                )

            if not isinstance(element["type"], str):
                raise ValueError(
                    f"expected_elements[{idx}]['type'] must be a string, "
                    f"got {type(element['type']).__name__}"
                )

            if "required" in element:
                if not isinstance(element["required"], bool):
                    raise ValueError(
                        f"expected_elements[{idx}]['required'] must be a boolean, "
                        f"got {type(element['required']).__name__}"
                    )

            if "minimum" in element:
                if not isinstance(element["minimum"], int):
                    raise ValueError(
                        f"expected_elements[{idx}]['minimum'] must be an integer, "
                        f"got {type(element['minimum']).__name__}"
                    )
                if element["minimum"] < 0:
                    raise ValueError(
                        f"expected_elements[{idx}]['minimum'] must be non-negative, "
                        f"got {element['minimum']}"
                    )

    def _validate_document_position(self, logger: Any) -> None:
        """Validate document_position field structure."""
        if self.document_position is None:
            return

        if not isinstance(self.document_position, tuple):
            raise ValueError(
                f"document_position must be a tuple, got {type(self.document_position).__name__}"
            )

        if len(self.document_position) != 2:
            raise ValueError(
                f"document_position must have exactly 2 elements, got {len(self.document_position)}"
            )

        start, end = self.document_position

        if not isinstance(start, int):
            raise ValueError(
                f"document_position[0] (start) must be an integer, got {type(start).__name__}"
            )

        if not isinstance(end, int):
            raise ValueError(
                f"document_position[1] (end) must be an integer, got {type(end).__name__}"
            )

        if start < 0:
            raise ValueError(
                f"document_position start offset must be non-negative, got {start}"
            )

        if end < start:
            raise ValueError(
                f"document_position end offset ({end}) must be >= start offset ({start})"
            )

        if start == end:
            logger.warning(
                f"ChunkData {self.id} has zero-length document_position [{start}, {end})"
            )


@dataclass
class PreprocessedDocument:
    """Orchestrator representation of a processed document.

    This is the normalized document format used internally by the orchestrator.
    It can be constructed from ingestion payloads or created directly.
    """

    document_id: str
    raw_text: str
    sentences: list[Any]
    tables: list[Any]
    metadata: dict[str, Any]
    source_path: Path | None = None
    sentence_metadata: list[Any] = field(default_factory=list)
    indexes: dict[str, Any] | None = None
    structured_text: dict[str, Any] | None = None
    language: str | None = None
    ingested_at: datetime | None = None
    full_text: str | None = None

    chunks: list[ChunkData] = field(default_factory=list)
    chunk_index: dict[str, int] = field(default_factory=dict)
    chunk_graph: dict[str, Any] = field(default_factory=dict)
    processing_mode: Literal["flat", "chunked"] = "chunked"

    def __post_init__(self) -> None:
        """Validate document fields after initialization.

        Raises:
            ValueError: If raw_text is empty, chunks are missing, or processing mode is invalid
        """
        if (not self.raw_text or not self.raw_text.strip()) and self.full_text:
            self.raw_text = self.full_text
        if not self.raw_text or not self.raw_text.strip():
            raise ValueError(
                "PreprocessedDocument cannot have empty raw_text. "
                "Use PreprocessedDocument.ensure() to create from SPC pipeline. "
                "Phase 1 must populate raw_text field."
            )
        if self.processing_mode != "chunked":
            raise ValueError(
                f"processing_mode must be 'chunked' for irrigation; got {self.processing_mode!r}. "
                "Phase 1 must set processing_mode to 'chunked' for chunk-based routing."
            )

        if not isinstance(self.chunks, list):
            raise ValueError(
                f"PreprocessedDocument.chunks must be a list, got {type(self.chunks).__name__}. "
                "Phase 1 must populate chunks as a list of ChunkData instances."
            )

        for idx, chunk in enumerate(self.chunks):
            if not isinstance(chunk, ChunkData):
                raise ValueError(
                    f"PreprocessedDocument.chunks[{idx}] must be ChunkData instance, "
                    f"got {type(chunk).__name__}. "
                    "Phase 1 must produce only ChunkData instances."
                )

    @staticmethod
    def _dataclass_to_dict(value: Any) -> Any:
        """Convert a dataclass to a dictionary if applicable."""
        from dataclasses import asdict, is_dataclass

        if is_dataclass(value):
            return asdict(value)
        return value

    @classmethod
    def ensure(
        cls,
        document: Any,
        *,
        document_id: str | None = None,
        use_spc_ingestion: bool = True,
    ) -> PreprocessedDocument:
        """Normalize arbitrary ingestion payloads into orchestrator documents.

        Args:
            document: Document to normalize (PreprocessedDocument or CanonPolicyPackage)
            document_id: Optional document ID override
            use_spc_ingestion: Must be True (SPC is now the only supported ingestion method)

        Returns:
            PreprocessedDocument instance

        Raises:
            ValueError: If use_spc_ingestion is False
            TypeError: If document type is not supported
        """
        import logging

        logger = logging.getLogger(__name__)

        if not use_spc_ingestion:
            raise ValueError(
                "SPC ingestion is now required. Set use_spc_ingestion=True or remove the parameter. "
                "Legacy ingestion methods (document_ingestion module) are no longer supported."
            )

        if isinstance(document, type):
            class_name = getattr(document, "__name__", str(document))
            raise TypeError(
                f"Expected document instance, got class type '{class_name}'. "
                "Pass an instance of the document, not the class itself."
            )

        if isinstance(document, cls):
            return document

        if hasattr(document, "chunk_graph"):
            chunk_graph = getattr(document, "chunk_graph", None)
            if chunk_graph is None:
                raise ValueError(
                    "Document has chunk_graph attribute but it is None. "
                    "Ensure SPC ingestion pipeline completed successfully."
                )

            if not hasattr(chunk_graph, "chunks") or not chunk_graph.chunks:
                raise ValueError(
                    "Document chunk_graph is empty. "
                    "Ensure SPC ingestion pipeline completed successfully and extracted chunks."
                )

            try:
                from farfan_pipeline.utils.spc_adapter import SPCAdapter

                adapter = SPCAdapter()
                preprocessed = adapter.to_preprocessed_document(
                    document, document_id=document_id
                )

                validation_results = []

                if not preprocessed.raw_text or not preprocessed.raw_text.strip():
                    raise ValueError(
                        "SPC ingestion produced empty document. "
                        "Check that the source document contains extractable text."
                    )
                text_length = len(preprocessed.raw_text)
                validation_results.append(f"raw_text: {text_length} chars")

                sentence_count = (
                    len(preprocessed.sentences) if preprocessed.sentences else 0
                )
                if sentence_count == 0:
                    logger.warning(
                        "SPC ingestion produced zero sentences - document may be malformed"
                    )
                validation_results.append(f"sentences: {sentence_count}")

                chunk_count = preprocessed.metadata.get("chunk_count", 0)
                validation_results.append(f"chunks: {chunk_count}")

                logger.info(
                    f"SPC ingestion validation passed: {', '.join(validation_results)}"
                )

                return preprocessed
            except ImportError as e:
                raise ImportError(
                    "SPC ingestion requires spc_adapter module. "
                    "Ensure farfan_core.utils.spc_adapter is available."
                ) from e
            except ValueError:
                raise
            except Exception as e:
                raise TypeError(
                    f"Failed to adapt SPC document: {e}. "
                    "Ensure document is a valid CanonPolicyPackage instance from SPC pipeline."
                ) from e

        raise TypeError(
            "Unsupported preprocessed document payload. "
            f"Expected PreprocessedDocument or CanonPolicyPackage with chunk_graph, got {type(document)!r}. "
            "Documents must be processed through the SPC ingestion pipeline first."
        )



================================================================================
FILE: src/farfan_pipeline/core/wiring/__init__.py
================================================================================

"""Wiring System - Fine-Grained Module Connection and Contract Validation.

This package implements the complete wiring architecture for SAAAAAA,
providing deterministic initialization, contract validation, and observability
for all module connections.

Architecture:
- Ports and adapters (hexagonal architecture)
- Dependency injection via constructors
- Feature flags for conditional wiring
- Contract validation between all links
- OpenTelemetry observability
- Deterministic initialization order

Key Modules:
- errors: Typed error classes for wiring failures
- contracts: Pydantic models for link contracts
- feature_flags: Typed feature flags
- bootstrap: Deterministic initialization engine
- validation: Contract validation between links
- observability: Tracing and metrics
- analysis_factory: Factory for analysis module dependency injection
"""

from farfan_pipeline.core.wiring.analysis_factory import (
    create_analysis_components,
    create_bayesian_confidence_calculator,
    create_contradiction_detector,
    create_document_loader,
    create_municipal_analyzer,
    create_municipal_ontology,
    create_performance_analyzer,
    create_semantic_analyzer,
    create_temporal_logic_verifier,
)

__all__ = [
    'create_analysis_components',
    'create_bayesian_confidence_calculator',
    'create_contradiction_detector',
    'create_document_loader',
    'create_municipal_analyzer',
    'create_municipal_ontology',
    'create_performance_analyzer',
    'create_semantic_analyzer',
    'create_temporal_logic_verifier',
]



================================================================================
FILE: src/farfan_pipeline/core/wiring/analysis_factory.py
================================================================================






================================================================================
FILE: src/farfan_pipeline/core/wiring/bootstrap.py
================================================================================

"""Bootstrap module for deterministic wiring initialization.

Implements the complete initialization sequence with:
1. Resource loading (QuestionnaireResourceProvider)
2. Signal system setup (memory:// by default, HTTP optional)
3. CoreModuleFactory with DI
4. ArgRouterExtended (â\211¥30 routes)
5. Orchestrator assembly

All initialization is deterministic and observable.
"""

from __future__ import annotations

import json
import time
from collections import OrderedDict
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

import structlog

from farfan_pipeline.config.paths import CONFIG_DIR, DATA_DIR
from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
from farfan_pipeline.core.orchestrator.factory import CoreModuleFactory
from farfan_pipeline.core.orchestrator.signals import (
    InMemorySignalSource,
    SignalClient,
    SignalPack,
    SignalRegistry,
)

@dataclass
class QuestionnaireResourceProvider:
    """Provider for questionnaire resources."""
    questionnaire_path: Path | None = None
    data_dir: Path = field(default_factory=lambda: DATA_DIR)

try:  # Optional dependency: calibration orchestrator
    from farfan_pipeline.core.calibration.orchestrator import CalibrationOrchestrator as _CalibrationOrchestrator
    from farfan_pipeline.core.calibration.config import DEFAULT_CALIBRATION_CONFIG as _DEFAULT_CALIBRATION_CONFIG
    _HAS_CALIBRATION = True
except Exception:  # pragma: no cover - only during stripped installs
    _CalibrationOrchestrator = None  # type: ignore[assignment]
    _DEFAULT_CALIBRATION_CONFIG = None  # type: ignore[assignment]
    _HAS_CALIBRATION = False

from farfan_pipeline.core.wiring.errors import MissingDependencyError, WiringInitializationError
from farfan_pipeline.core.wiring.feature_flags import WiringFeatureFlags
from farfan_pipeline.core.wiring.phase_0_validator import Phase0Validator
from farfan_pipeline.core.wiring.validation import WiringValidator

logger = structlog.get_logger(__name__)


@dataclass
class WiringComponents:
    """Container for all wired components.

    Attributes:
        provider: QuestionnaireResourceProvider
        signal_client: SignalClient (memory:// or HTTP)
        signal_registry: SignalRegistry with TTL and LRU
        executor_config: ExecutorConfig with defaults
        factory: CoreModuleFactory with DI
        arg_router: ExtendedArgRouter with special routes
        class_registry: Class registry for routing
        validator: WiringValidator for contract checking
        flags: Feature flags used during initialization
        init_hashes: Hashes computed during initialization
    """

    provider: QuestionnaireResourceProvider
    signal_client: SignalClient
    signal_registry: SignalRegistry
    executor_config: ExecutorConfig
    factory: CoreModuleFactory
    arg_router: ExtendedArgRouter
    class_registry: dict[str, type]
    validator: WiringValidator
    flags: WiringFeatureFlags
    calibration_orchestrator: "_CalibrationOrchestrator | None" = None
    init_hashes: dict[str, str] = field(default_factory=dict)


CANONICAL_POLICY_AREA_DEFINITIONS: "OrderedDict[str, dict[str, list[str] | str]]" = OrderedDict(
    [
        (
            "PA01",
            {
                "name": "Derechos de las mujeres e igualdad de gÃ©nero",
                "slug": "genero_mujeres",
                "aliases": ["fiscal"],
            },
        ),
        (
            "PA02",
            {
                "name": "PrevenciÃ³n de la violencia y protecciÃ³n",
                "slug": "seguridad_violencia",
                "aliases": ["salud"],
            },
        ),
        (
            "PA03",
            {
                "name": "Ambiente sano y cambio climÃ¡tico",
                "slug": "ambiente",
                "aliases": ["ambiente"],
            },
        ),
        (
            "PA04",
            {
                "name": "Derechos econÃ³micos, sociales y culturales",
                "slug": "derechos_sociales",
                "aliases": ["energÃ-a"],
            },
        ),
        (
            "PA05",
            {
                "name": "Derechos de las vÃ-ctimas y construcciÃ³n de paz",
                "slug": "paz_victimas",
                "aliases": ["transporte"],
            },
        ),
        (
            "PA06",
            {
                "name": "Derecho al futuro de la niÃ±ez y juventud",
                "slug": "ninez_juventud",
                "aliases": [],
            },
        ),
        (
            "PA07",
            {
                "name": "Tierras y territorios",
                "slug": "tierras_territorios",
                "aliases": [],
            },
        ),
        (
            "PA08",
            {
                "name": "LÃderes, lideresas y defensores de DD. HH.",
                "slug": "liderazgos_ddhh",
                "aliases": [],
            },
        ),
        (
            "PA09",
            {
                "name": "Derechos de personas privadas de libertad",
                "slug": "privados_libertad",
                "aliases": [],
            },
        ),
        (
            "PA10",
            {
                "name": "MigraciÃ³n transfronteriza",
                "slug": "migracion",
                "aliases": [],
            },
        ),
    ]
)

SIGNAL_PACK_VERSION = "1.0.0"
MAX_PATTERNS_PER_POLICY_AREA = 32

class WiringBootstrap:
    """Bootstrap engine for deterministic wiring initialization.

    Follows strict initialization order:
    1. Load resources (questionnaire)
    2. Build signal system (memory:// or HTTP)
    3. Create factory with DI
    4. Initialize arg router
    5. Validate all contracts
    """

    def __init__(
        self,
        questionnaire_path: str | Path,
        questionnaire_hash: str,
        executor_config_path: str | Path,
        calibration_profile: str,
        abort_on_insufficient: bool,
        resource_limits: dict[str, int],
        flags: WiringFeatureFlags | None = None,
    ) -> None:
        """Initialize bootstrap engine.

        Args:
            questionnaire_path: Path to questionnaire monolith JSON.
            questionnaire_hash: Expected SHA-256 hash of the monolith.
            executor_config_path: Path to the executor configuration.
            calibration_profile: The calibration profile to use.
            abort_on_insufficient: Flag to abort on insufficient data.
            resource_limits: Resource limit settings.
            flags: Feature flags (defaults to environment).
        """
        self.questionnaire_path = questionnaire_path
        self.questionnaire_hash = questionnaire_hash
        self.executor_config_path = executor_config_path
        self.calibration_profile = calibration_profile
        self.abort_on_insufficient = abort_on_insufficient
        self.resource_limits = resource_limits
        self.flags = flags or WiringFeatureFlags.from_env()
        self._start_time = time.time()

        # Validate flags
        warnings = self.flags.validate()
        for warning in warnings:
            logger.warning("feature_flag_warning", message=warning)

        logger.info(
            "wiring_bootstrap_initialized",
            questionnaire_path=str(questionnaire_path) if questionnaire_path else None,
            flags=self.flags.to_dict(),
        )

    def bootstrap(self) -> WiringComponents:
        """Execute complete bootstrap sequence.

        Returns:
            WiringComponents with all initialized modules

        Raises:
            WiringInitializationError: If any phase fails
        """
        logger.info("wiring_bootstrap_start")

        try:
            # Phase 0: Validate configuration contract
            logger.info("wiring_init_phase", phase="phase_0_validation")
            phase_0_validator = Phase0Validator()
            raw_config = {
                "monolith_path": self.questionnaire_path,
                "questionnaire_hash": self.questionnaire_hash,
                "executor_config_path": self.executor_config_path,
                "calibration_profile": self.calibration_profile,
                "abort_on_insufficient": self.abort_on_insufficient,
                "resource_limits": self.resource_limits,
            }
            phase_0_validator.validate(raw_config)
            logger.info("phase_0_validation_passed")

            # Phase 1: Load resources
            provider = self._load_resources()

            # Phase 2: Build signal system
            signal_client, signal_registry = self._build_signal_system(provider)

            # Phase 3: Create executor config
            executor_config = self._create_executor_config()

            # Phase 4: Create factory with DI
            factory = self._create_factory(provider, signal_registry, executor_config)

            # Phase 5: Build class registry
            class_registry = self._build_class_registry()

            # Phase 6: Initialize arg router
            arg_router = self._create_arg_router(class_registry)

            # Phase 7: Create validator
            validator = WiringValidator()

            # Phase 8: Create calibration orchestrator (optional enhancement)
            calibration_orchestrator = self._create_calibration_orchestrator()

            # Phase 9: Seed signals (if memory mode)
            if signal_client._transport == "memory":
                metrics = self._seed_canonical_policy_area_signals(
                    signal_client._memory_source,
                    signal_registry,
                    provider,
                )
                logger.info(
                    "signals_seeded",
                    areas=metrics["canonical_areas"],
                    aliases=metrics["legacy_aliases"],
                    hit_rate=metrics["hit_rate"],
                )

            # Compute initialization hashes
            init_hashes = self._compute_init_hashes(
                provider, signal_registry, factory, arg_router
            )

            components = WiringComponents(
                provider=provider,
                signal_client=signal_client,
                signal_registry=signal_registry,
                executor_config=executor_config,
                factory=factory,
                arg_router=arg_router,
                class_registry=class_registry,
                validator=validator,
                calibration_orchestrator=calibration_orchestrator,
                flags=self.flags,
                init_hashes=init_hashes,
            )

            elapsed = time.time() - self._start_time

            logger.info(
                "wiring_bootstrap_complete",
                elapsed_s=elapsed,
                factory_instances=19,  # Expected count
                argrouter_routes=arg_router.get_special_route_coverage(),
                signals_mode=signal_client._transport,
                init_hashes={k: v[:16] for k, v in init_hashes.items()},
            )

            return components

        except Exception as e:
            elapsed = time.time() - self._start_time
            logger.error(
                "wiring_bootstrap_failed",
                elapsed_s=elapsed,
                error=str(e),
                error_type=type(e).__name__,
            )
            raise

    def _load_resources(self) -> QuestionnaireResourceProvider:
        """Load questionnaire resources.

        Returns:
            QuestionnaireResourceProvider instance

        Raises:
            WiringInitializationError: If loading fails
        """
        logger.info("wiring_init_phase", phase="load_resources")

        try:
            if self.questionnaire_path:
                path = Path(self.questionnaire_path)
                if not path.exists():
                    raise MissingDependencyError(
                        dependency=str(path),
                        required_by="WiringBootstrap",
                        fix=f"Ensure questionnaire file exists at {path}",
                    )

                with open(path, encoding="utf-8") as f:
                    data = json.load(f)

                provider = QuestionnaireResourceProvider(data)
            else:
                # Use default/empty provider
                provider = QuestionnaireResourceProvider({})

            logger.info(
                "questionnaire_loaded",
                path=str(self.questionnaire_path) if self.questionnaire_path else "default",
            )

            return provider

        except Exception as e:
            raise WiringInitializationError(
                phase="load_resources",
                component="QuestionnaireResourceProvider",
                reason=str(e),
            ) from e

    def _build_signal_system(
        self,
        provider: QuestionnaireResourceProvider,
    ) -> tuple[SignalClient, SignalRegistry]:
        """Build signal system (memory:// or HTTP).

        Args:
            provider: QuestionnaireResourceProvider for signal data

        Returns:
            Tuple of (SignalClient, SignalRegistry)

        Raises:
            WiringInitializationError: If setup fails
        """
        logger.info("wiring_init_phase", phase="build_signal_system")

        try:
            # Create registry first
            registry = SignalRegistry(
                max_size=100,
                default_ttl_s=3600,
            )

            # Create signal source
            if self.flags.enable_http_signals:
                # HTTP mode (requires explicit configuration)
                base_url = "http://127.0.0.1:8000"  # Default, should be configurable
                logger.info("signal_client_http_mode", base_url=base_url)

                client = SignalClient(
                    base_url=base_url,
                    enable_http_signals=True,
                )
            else:
                # Memory mode (default)
                memory_source = InMemorySignalSource()

                client = SignalClient(
                    base_url="memory://",
                    enable_http_signals=False,
                    memory_source=memory_source,
                )

                logger.info("signal_client_memory_mode")

            return client, registry

        except Exception as e:
            raise WiringInitializationError(
                phase="build_signal_system",
                component="SignalClient/SignalRegistry",
                reason=str(e),
            ) from e

    def _create_executor_config(self) -> ExecutorConfig:
        """Create executor configuration.

        Returns:
            ExecutorConfig with defaults
        """
        logger.info("wiring_init_phase", phase="create_executor_config")

        config = ExecutorConfig(
            max_tokens=2048,
            temperature=0.0,  # Deterministic
            timeout_s=30.0,
            retry=2,
            seed=0 if self.flags.deterministic_mode else None,
        )

        logger.info(
            "executor_config_created",
            deterministic=self.flags.deterministic_mode,
            seed=config.seed,
        )

        return config

    def _create_factory(
        self,
        provider: QuestionnaireResourceProvider,
        registry: SignalRegistry,
        config: ExecutorConfig,
    ) -> CoreModuleFactory:
        """Create CoreModuleFactory with DI.

        Args:
            provider: QuestionnaireResourceProvider
            registry: SignalRegistry for injection
            config: ExecutorConfig for injection

        Returns:
            CoreModuleFactory instance

        Raises:
            WiringInitializationError: If creation fails
        """
        logger.info("wiring_init_phase", phase="create_factory")

        try:
            factory = CoreModuleFactory(
                data_dir=provider.data_dir,
            )

            logger.info(
                "factory_created",
                data_dir=str(provider.data_dir),
            )

            return factory

        except Exception as e:
            raise WiringInitializationError(
                phase="create_factory",
                component="CoreModuleFactory",
                reason=str(e),
            ) from e

    def _build_class_registry(self) -> dict[str, type]:
        """Build class registry for arg router.

        Returns:
            Class registry mapping names to types

        Raises:
            WiringInitializationError: If build fails
        """
        logger.info("wiring_init_phase", phase="build_class_registry")

        try:
            registry = build_class_registry()

            logger.info(
                "class_registry_built",
                class_count=len(registry),
            )

            return registry

        except Exception as e:
            raise WiringInitializationError(
                phase="build_class_registry",
                component="ClassRegistry",
                reason=str(e),
            ) from e

    def _create_arg_router(
        self,
        class_registry: dict[str, type],
    ) -> ExtendedArgRouter:
        """Create ExtendedArgRouter with special routes.

        Args:
            class_registry: Class registry for routing

        Returns:
            ExtendedArgRouter instance

        Raises:
            WiringInitializationError: If creation fails
        """
        logger.info("wiring_init_phase", phase="create_arg_router")

        try:
            router = ExtendedArgRouter(class_registry)

            route_count = router.get_special_route_coverage()

            if route_count < 30:
                logger.warning(
                    "argrouter_coverage_low",
                    count=route_count,
                    expected=30,
                )

            logger.info(
                "arg_router_created",
                special_routes=route_count,
            )

            return router

        except Exception as e:
            raise WiringInitializationError(
                phase="create_arg_router",
                component="ExtendedArgRouter",
                reason=str(e),
            ) from e

    def _create_calibration_orchestrator(self) -> "_CalibrationOrchestrator | None":
        """
        Create CalibrationOrchestrator when calibration stack is available.

        Returns:
            CalibrationOrchestrator instance or None if unavailable.
        """
        if not _HAS_CALIBRATION or _CalibrationOrchestrator is None or _DEFAULT_CALIBRATION_CONFIG is None:
            logger.info("calibration_system_unavailable")
            return None

        data_dir = DATA_DIR
        config_dir = CONFIG_DIR

        kwargs: dict[str, Any] = {"config": _DEFAULT_CALIBRATION_CONFIG}

        intrinsic_path = config_dir / "intrinsic_calibration.json"
        if intrinsic_path.exists():
            kwargs["intrinsic_calibration_path"] = intrinsic_path

        compatibility_path = data_dir / "method_compatibility.json"
        if compatibility_path.exists():
            kwargs["compatibility_path"] = compatibility_path

        registry_path = data_dir / "method_registry.json"
        if registry_path.exists():
            kwargs["method_registry_path"] = registry_path

        signatures_path = data_dir / "method_signatures.json"
        if signatures_path.exists():
            kwargs["method_signatures_path"] = signatures_path

        try:
            orchestrator = _CalibrationOrchestrator(**kwargs)
            logger.info(
                "calibration_orchestrator_ready",
                intrinsic=str(intrinsic_path),
                compatibility=str(compatibility_path),
            )
            return orchestrator
        except Exception as exc:  # pragma: no cover - defensive guardrail
            logger.warning(
                "calibration_orchestrator_initialization_failed",
                error=str(exc),
            )
            return None

    def _build_signal_pack(
        self,
        provider: QuestionnaireResourceProvider,
        canonical_id: str,
        meta: dict[str, Any],
        *,
        alias: str | None = None,
    ) -> SignalPack:
        """Build a SignalPack for a canonical policy area (and optional alias)."""
        pattern_source = getattr(provider, "get_patterns_for_area", None)
        patterns = pattern_source(canonical_id, MAX_PATTERNS_PER_POLICY_AREA) if callable(pattern_source) else []

        pack = SignalPack(
            version=SIGNAL_PACK_VERSION,
            policy_area=alias or canonical_id,  # type: ignore[arg-type]
            patterns=patterns,
            metadata={
                "canonical_id": canonical_id,
                "display_name": meta["name"],
                "slug": meta["slug"],
                "alias": alias,
            },
        )
        fingerprint = pack.compute_hash()
        return pack.model_copy(update={"source_fingerprint": fingerprint})

    @staticmethod
    def _register_signal_pack(
        memory_source: InMemorySignalSource,
        registry: SignalRegistry,
        pack: SignalPack,
    ) -> None:
        """Register pack in both memory source and registry."""
        memory_source.register(pack.policy_area, pack)
        registry.put(pack.policy_area, pack)
        logger.debug(
            "signal_seeded",
            policy_area=pack.policy_area,
            canonical_id=pack.metadata.get("canonical_id"),
            patterns=len(pack.patterns),
        )

    def _seed_canonical_policy_area_signals(
        self,
        memory_source: InMemorySignalSource,
        registry: SignalRegistry,
        provider: QuestionnaireResourceProvider,
    ) -> dict[str, Any]:
        """
        Seed signal registry with canonical (PA01-PA10) policy areas.

        Returns:
            Metrics dict with coverage and legacy alias info.
        """
        canonical_count = 0
        alias_count = 0

        for area_id, meta in CANONICAL_POLICY_AREA_DEFINITIONS.items():
            pack = self._build_signal_pack(provider, area_id, meta)
            self._register_signal_pack(memory_source, registry, pack)
            canonical_count += 1

            for alias in meta["aliases"]:  # type: ignore[index]
                alias_pack = self._build_signal_pack(
                    provider,
                    area_id,
                    meta,
                    alias=alias,
                )
                self._register_signal_pack(memory_source, registry, alias_pack)
                alias_count += 1

        hits = sum(
            1
            for area_id in CANONICAL_POLICY_AREA_DEFINITIONS
            if registry.get(area_id) is not None
        )
        total_required = len(CANONICAL_POLICY_AREA_DEFINITIONS)
        hit_rate = hits / total_required if total_required else 0.0

        return {
            "canonical_areas": canonical_count,
            "legacy_aliases": alias_count,
            "hit_rate": hit_rate,
            "required_hit_rate": 0.95,
        }

    def seed_signals_public(
        self,
        client: SignalClient,
        registry: SignalRegistry,
        provider: QuestionnaireResourceProvider,
    ) -> dict[str, Any]:
        """Seed initial signals in memory mode (PUBLIC API).

        This replaces the private _seed_signals method with a public API that:
        1. Validates the SignalClient is using memory transport
        2. Returns deterministic metrics for validation
        3. Enforces the â\211¥95% hit rate requirement

        Args:
            client: SignalClient to seed (must be in memory mode)
            registry: SignalRegistry to populate
            provider: QuestionnaireResourceProvider for patterns

        Returns:
            Dict with seeding metrics (areas_seeded, total_signals, hit_rate)

        Raises:
            ValueError: If client is not in memory mode
            WiringInitializationError: If hit rate requirement is not met
        """
        logger.info("wiring_init_phase", phase="seed_signals_public")

        if getattr(client, "_transport", None) != "memory":
            raise ValueError(
                "Signal seeding requires memory mode. "
                "Set enable_http_signals=False in WiringFeatureFlags."
            )

        memory_source = getattr(client, "_memory_source", None)
        if memory_source is None:
            raise ValueError("Signal client memory source not initialized.")

        metrics = self._seed_canonical_policy_area_signals(
            memory_source,
            registry,
            provider,
        )

        if metrics["hit_rate"] < metrics["required_hit_rate"]:
            raise WiringInitializationError(
                phase="seed_signals",
                component="SignalRegistry",
                reason=(
                    f"Signal hit rate {metrics['hit_rate']:.2%} below "
                    f"required threshold {metrics['required_hit_rate']:.2%}"
                ),
            )

        return metrics



    def _compute_init_hashes(
        self,
        provider: QuestionnaireResourceProvider,
        registry: SignalRegistry,
        factory: CoreModuleFactory,
        router: ExtendedArgRouter,
    ) -> dict[str, str]:
        """Compute hashes for initialized components.

        Args:
            provider: QuestionnaireResourceProvider
            registry: SignalRegistry
            factory: CoreModuleFactory
            router: ExtendedArgRouter

        Returns:
            Dict of component names to their hashes
        """
        import blake3

        hashes = {}

        # Provider hash (based on data keys)
        provider_keys = sorted(provider._data.keys()) if hasattr(provider, '_data') else []
        hashes["provider"] = blake3.blake3(
            json.dumps(provider_keys, sort_keys=True).encode('utf-8')
        ).hexdigest()

        # Registry hash (based on metrics)
        registry_metrics = registry.get_metrics()
        hashes["registry"] = blake3.blake3(
            json.dumps(registry_metrics, sort_keys=True).encode('utf-8')
        ).hexdigest()

        # Router hash (based on special routes count)
        router_data = {"route_count": router.get_special_route_coverage()}
        hashes["router"] = blake3.blake3(
            json.dumps(router_data, sort_keys=True).encode('utf-8')
        ).hexdigest()

        return hashes


__all__ = [
    'WiringComponents',
    'WiringBootstrap',
]



================================================================================
FILE: src/farfan_pipeline/core/wiring/contracts.py
================================================================================

"""Contract models for wiring validation.

Defines Pydantic models for each link's deliverable and expectation.
Validation ensures type safety and completeness at every boundary.
"""

from __future__ import annotations

from typing import Any

from pydantic import BaseModel, Field, field_validator


class CPPDeliverable(BaseModel):
    """Contract for CPP ingestion output (Deliverable).

    DEPRECATED: Use SPCDeliverable instead. This model is kept for backward compatibility.

    Note: CPP (Canon Policy Package) is the legacy name for SPC (Smart Policy Chunks).
    """

    chunk_graph: dict[str, Any] = Field(
        description="Chunk graph with all chunks"
    )
    policy_manifest: dict[str, Any] = Field(
        description="Policy metadata manifest"
    )
    provenance_completeness: float = Field(
        ge=0.0,
        le=1.0,
        description="Provenance completeness score (must be 1.0)"
    )
    schema_version: str = Field(
        description="CPP schema version"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }

    def __init__(self, **data: Any) -> None:
        import warnings
        warnings.warn(
            "CPPDeliverable is deprecated. Use SPCDeliverable instead.",
            DeprecationWarning,
            stacklevel=2
        )
        super().__init__(**data)

    @field_validator("provenance_completeness")
    @classmethod
    def validate_completeness(cls, v: float) -> float:
        """Ensure provenance is 100% complete."""
        if v != 1.0:
            raise ValueError(
                f"provenance_completeness must be 1.0, got {v}. "
                "Ensure ingestion pipeline completed successfully."
            )
        return v


class SPCDeliverable(BaseModel):
    """Contract for SPC (Smart Policy Chunks) ingestion output (Deliverable).

    This is the preferred terminology for new code. SPC is the successor to CPP.
    """

    chunk_graph: dict[str, Any] = Field(
        description="Chunk graph with all chunks"
    )
    policy_manifest: dict[str, Any] = Field(
        description="Policy metadata manifest"
    )
    provenance_completeness: float = Field(
        ge=0.0,
        le=1.0,
        description="Provenance completeness score (must be 1.0)"
    )
    schema_version: str = Field(
        description="SPC schema version"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }

    @field_validator("provenance_completeness")
    @classmethod
    def validate_completeness(cls, v: float) -> float:
        """Ensure provenance is 100% complete."""
        if v != 1.0:
            raise ValueError(
                f"provenance_completeness must be 1.0, got {v}. "
                "Ensure SPC ingestion pipeline completed successfully."
            )
        return v


class AdapterExpectation(BaseModel):
    """Contract for CPPAdapter input (Expectation)."""

    chunk_graph: dict[str, Any] = Field(
        description="Must have chunk_graph with chunks"
    )
    policy_manifest: dict[str, Any] = Field(
        description="Must have policy_manifest"
    )
    provenance_completeness: float = Field(
        ge=1.0,
        le=1.0,
        description="Must be exactly 1.0"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",  # Allow additional fields
    }


class PreprocessedDocumentDeliverable(BaseModel):
    """Contract for CPPAdapter output (Deliverable)."""

    sentence_metadata: list[dict[str, Any]] = Field(
        min_length=1,
        description="Must have at least one sentence"
    )
    resolution_index: dict[str, Any] = Field(
        description="Resolution index must be consistent"
    )
    provenance_completeness: float = Field(
        ge=1.0,
        le=1.0,
        description="Must maintain 1.0 completeness"
    )
    document_id: str = Field(
        min_length=1,
        description="Document ID must be non-empty"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }


class OrchestratorExpectation(BaseModel):
    """Contract for Orchestrator input (Expectation)."""

    sentence_metadata: list[dict[str, Any]] = Field(
        min_length=1,
        description="Requires sentence_metadata"
    )
    document_id: str = Field(
        min_length=1,
        description="Requires document_id"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class ArgRouterPayloadDeliverable(BaseModel):
    """Contract for Orchestrator to ArgRouter (Deliverable)."""

    class_name: str = Field(
        min_length=1,
        description="Target class name"
    )
    method_name: str = Field(
        min_length=1,
        description="Target method name"
    )
    payload: dict[str, Any] = Field(
        description="Method arguments payload"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }


class ArgRouterExpectation(BaseModel):
    """Contract for ArgRouter input (Expectation)."""

    class_name: str = Field(
        min_length=1,
        description="Class must exist in registry"
    )
    method_name: str = Field(
        min_length=1,
        description="Method must exist on class"
    )
    payload: dict[str, Any] = Field(
        description="Payload with required arguments"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class ExecutorInputDeliverable(BaseModel):
    """Contract for ArgRouter to Executor (Deliverable)."""

    args: tuple[Any, ...] = Field(
        description="Positional arguments"
    )
    kwargs: dict[str, Any] = Field(
        description="Keyword arguments"
    )
    method_signature: str = Field(
        description="Target method signature for validation"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }


class SignalPackDeliverable(BaseModel):
    """Contract for SignalsClient output (Deliverable)."""

    version: str = Field(
        description="Signal pack version (must be present)"
    )
    policy_area: str = Field(
        description="Policy area for signals"
    )
    patterns: list[str] = Field(
        default_factory=list,
        description="Text patterns"
    )
    indicators: list[str] = Field(
        default_factory=list,
        description="KPI indicators"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",  # Allow additional signal fields
    }

    @field_validator("version")
    @classmethod
    def validate_version(cls, v: str) -> str:
        """Validate version format."""
        if not v or v.strip() == "":
            raise ValueError("version must be non-empty")
        return v


class SignalRegistryExpectation(BaseModel):
    """Contract for SignalRegistry input (Expectation)."""

    version: str = Field(
        min_length=1,
        description="Requires version"
    )
    policy_area: str = Field(
        min_length=1,
        description="Requires policy_area"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class EnrichedChunkDeliverable(BaseModel):
    """Contract for Executor output (Deliverable)."""

    chunk_id: str = Field(
        min_length=1,
        description="Chunk identifier"
    )
    used_signals: list[str] = Field(
        default_factory=list,
        description="Signals used during execution"
    )
    enrichment: dict[str, Any] = Field(
        description="Enrichment data"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class AggregateExpectation(BaseModel):
    """Contract for Aggregate input (Expectation)."""

    enriched_chunks: list[dict[str, Any]] = Field(
        min_length=1,
        description="Must have at least one enriched chunk"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class FeatureTableDeliverable(BaseModel):
    """Contract for Aggregate output (Deliverable)."""

    table_type: str = Field(
        description="Must be 'pyarrow.Table'"
    )
    num_rows: int = Field(
        ge=1,
        description="Must have at least one row"
    )
    column_names: list[str] = Field(
        min_length=1,
        description="Must have required columns"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }


class ScoreExpectation(BaseModel):
    """Contract for Score input (Expectation)."""

    table_type: str = Field(
        description="Must be pa.Table"
    )
    required_columns: list[str] = Field(
        min_length=1,
        description="Required columns for scoring"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class ScoresDeliverable(BaseModel):
    """Contract for Score output (Deliverable)."""

    dataframe_type: str = Field(
        description="Must be 'polars.DataFrame'"
    )
    num_rows: int = Field(
        ge=1,
        description="Must have at least one row"
    )
    metrics_computed: list[str] = Field(
        min_length=1,
        description="Metrics that were computed"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }


class ReportExpectation(BaseModel):
    """Contract for Report input (Expectation)."""

    dataframe_type: str = Field(
        description="Must be pl.DataFrame"
    )
    metrics_present: list[str] = Field(
        min_length=1,
        description="Metrics must be present"
    )
    manifest_present: bool = Field(
        description="Manifest must be provided"
    )

    model_config = {
        "frozen": True,
        "extra": "allow",
    }


class ReportDeliverable(BaseModel):
    """Contract for Report output (Deliverable)."""

    report_uris: dict[str, str] = Field(
        min_length=1,
        description="Mapping of report name to URI"
    )
    all_reports_generated: bool = Field(
        description="All declared reports generated"
    )

    model_config = {
        "frozen": True,
        "extra": "forbid",
    }


__all__ = [
    'CPPDeliverable',
    'SPCDeliverable',
    'AdapterExpectation',
    'PreprocessedDocumentDeliverable',
    'OrchestratorExpectation',
    'ArgRouterPayloadDeliverable',
    'ArgRouterExpectation',
    'ExecutorInputDeliverable',
    'SignalPackDeliverable',
    'SignalRegistryExpectation',
    'EnrichedChunkDeliverable',
    'AggregateExpectation',
    'FeatureTableDeliverable',
    'ScoreExpectation',
    'ScoresDeliverable',
    'ReportExpectation',
    'ReportDeliverable',
]



================================================================================
FILE: src/farfan_pipeline/core/wiring/errors.py
================================================================================

"""Typed error classes for wiring system.

All wiring errors include prescriptive fix information to guide remediation.
Errors are loud and explicit - no silent degradation is permitted.
"""
```

```

from __future__ import annotations

from typing import Any


class WiringError(Exception):
    """Base class for all wiring errors."""

    def __init__(self, message: str, details: dict[str, Any] | None = None) -> None:
        super().__init__(message)
        self.details = details or {}


class WiringContractError(WiringError):
    """Raised when a contract between two links is violated.

    Attributes:
        link: Name of the violated link (e.g., "cpp->adapter")
        expected_schema: Expected schema/type
        received_schema: Actual schema/type received
        field: Specific field that failed (if applicable)
        fix: Prescriptive fix instructions
    """

    def __init__(
        self,
        link: str,
        expected_schema: str,
        received_schema: str,
        field: str | None = None,
        fix: str | None = None,
    ) -> None:
        field_info = f" (field: {field})" if field else ""
        fix_info = f"\n\nFix: {fix}" if fix else ""

        message = (
            f"Contract violation in link '{link}'{field_info}\n"
            f"Expected: {expected_schema}\n"
            f"Received: {received_schema}"
            f"{fix_info}"
        )

        super().__init__(
            message,
            details={
                "link": link,
                "expected_schema": expected_schema,
                "received_schema": received_schema,
                "field": field,
                "fix": fix,
            }
        )


class MissingDependencyError(WiringError):
    """Raised when a required dependency is not available.

    Attributes:
        dependency: Name of missing dependency
        required_by: Module/component that requires it
        fix: How to resolve the missing dependency
    """

    def __init__(
        self,
        dependency: str,
        required_by: str,
        fix: str | None = None,
    ) -> None:
        fix_info = f"\n\nFix: {fix}" if fix else ""

        message = (
            f"Missing dependency '{dependency}' required by '{required_by}'"
            f"{fix_info}"
        )

        super().__init__(
            message,
            details={
                "dependency": dependency,
                "required_by": required_by,
                "fix": fix,
            }
        )


class WiringArgumentValidationError(WiringError):
    """Raised when argument routing validation fails.

    Attributes:
        class_name: Class being routed to
        method_name: Method being called
        issue: Description of validation issue
        provided_args: Arguments that were provided
        expected_args: Arguments that were expected
        fix: How to fix the argument mismatch
    """

    def __init__(
        self,
        class_name: str,
        method_name: str,
        issue: str,
        provided_args: list[str] | None = None,
        expected_args: list[str] | None = None,
        fix: str | None = None,
    ) -> None:
        fix_info = f"\n\nFix: {fix}" if fix else ""

        message = (
            f"Argument validation failed for {class_name}.{method_name}\n"
            f"Issue: {issue}"
        )

        if provided_args is not None:
            message += f"\nProvided: {', '.join(provided_args)}"
        if expected_args is not None:
            message += f"\nExpected: {', '.join(expected_args)}"

        message += fix_info

        super().__init__(
            message,
            details={
                "class_name": class_name,
                "method_name": method_name,
                "issue": issue,
                "provided_args": provided_args,
                "expected_args": expected_args,
                "fix": fix,
            }
        )


class WiringSignalUnavailableError(WiringError):
    """Raised when required signals are unavailable.

    Attributes:
        policy_area: Policy area for which signals were requested
        reason: Why signals are unavailable
        breaker_state: Circuit breaker state if applicable
    """

    def __init__(
        self,
        policy_area: str,
        reason: str,
        breaker_state: str | None = None,
    ) -> None:
        breaker_info = f" (breaker: {breaker_state})" if breaker_state else ""

        message = (
            f"Signals unavailable for policy area '{policy_area}'{breaker_info}\n"
            f"Reason: {reason}"
        )

        super().__init__(
            message,
            details={
                "policy_area": policy_area,
                "reason": reason,
                "breaker_state": breaker_state,
            }
        )


class WiringSignalSchemaError(WiringError):
    """Raised when signal pack schema is invalid.

    Attributes:
        pack_version: Signal pack version
        schema_issue: Description of schema problem
        field: Field with schema issue
    """

    def __init__(
        self,
        pack_version: str,
        schema_issue: str,
        field: str | None = None,
    ) -> None:
        field_info = f" (field: {field})" if field else ""

        message = (
            f"Invalid signal pack schema{field_info}\n"
            f"Version: {pack_version}\n"
            f"Issue: {schema_issue}"
        )

        super().__init__(
            message,
            details={
                "pack_version": pack_version,
                "schema_issue": schema_issue,
                "field": field,
            }
        )


class WiringInitializationError(WiringError):
    """Raised when wiring initialization fails.

    Attributes:
        phase: Initialization phase that failed
        component: Component being initialized
        reason: Why initialization failed
    """

    def __init__(
        self,
        phase: str,
        component: str,
        reason: str,
    ) -> None:
        message = (
            f"Wiring initialization failed in phase '{phase}'\n"
            f"Component: {component}\n"
            f"Reason: {reason}"
        )

        super().__init__(
            message,
            details={
                "phase": phase,
                "component": component,
                "reason": reason,
            }
        )


__all__ = [
    'WiringError',
    'WiringContractError',
    'MissingDependencyError',
    'WiringArgumentValidationError',
    'WiringSignalUnavailableError',
    'WiringSignalSchemaError',
    'WiringInitializationError',
]



================================================================================
FILE: src/farfan_pipeline/core/wiring/feature_flags.py
================================================================================

"""Feature flags for wiring system configuration.

All flags are typed and have explicit defaults. Flags control conditional
wiring paths and validation strictness.
"""

from __future__ import annotations

import os
from dataclasses import dataclass


@dataclass(frozen=True)
class WiringFeatureFlags:
    """Feature flags for wiring configuration.

    Attributes:
        use_spc_ingestion: Use SPC (Smart Policy Chunks) ingestion pipeline - canonical phase-one (default: True)
        enable_http_signals: Enable HTTP signal fetching (default: False)
        allow_threshold_override: Allow runtime threshold overrides (default: False)
        wiring_strict_mode: Enforce strict contract validation (default: True)
        enable_observability: Enable OpenTelemetry tracing (default: True)
        enable_metrics: Enable metrics collection (default: True)
        deterministic_mode: Force deterministic execution (default: True)
    """

    use_spc_ingestion: bool = True
    # Legacy alias for backwards compatibility
    use_cpp_ingestion: bool = True
    enable_http_signals: bool = False
    allow_threshold_override: bool = False
    wiring_strict_mode: bool = True
    enable_observability: bool = True
    enable_metrics: bool = True
    deterministic_mode: bool = True

    @classmethod
    def from_env(cls) -> WiringFeatureFlags:
        """Load feature flags from environment variables.

        Environment variables:
        - SAAAAAA_USE_SPC_INGESTION: "true" or "false" (canonical phase-one)
        - SAAAAAA_USE_CPP_INGESTION: "true" or "false" (legacy alias)
        - SAAAAAA_ENABLE_HTTP_SIGNALS: "true" or "false"
        - SAAAAAA_ALLOW_THRESHOLD_OVERRIDE: "true" or "false"
        - SAAAAAA_WIRING_STRICT_MODE: "true" or "false"
        - SAAAAAA_ENABLE_OBSERVABILITY: "true" or "false"
        - SAAAAAA_ENABLE_METRICS: "true" or "false"
        - SAAAAAA_DETERMINISTIC_MODE: "true" or "false"

        Returns:
            WiringFeatureFlags with values from environment
        """
        def get_bool(key: str, default: bool) -> bool:
            value = os.getenv(key, str(default)).lower()
            return value in ("true", "1", "yes", "on")

        # Prefer new SPC name, fallback to legacy CPP name
        spc_flag = get_bool("SAAAAAA_USE_SPC_INGESTION",
                           get_bool("SAAAAAA_USE_CPP_INGESTION", True))

        return cls(
            use_spc_ingestion=spc_flag,
            use_cpp_ingestion=spc_flag,  # Keep in sync for backwards compatibility
            enable_http_signals=get_bool("SAAAAAA_ENABLE_HTTP_SIGNALS", False),
            allow_threshold_override=get_bool("SAAAAAA_ALLOW_THRESHOLD_OVERRIDE", False),
            wiring_strict_mode=get_bool("SAAAAAA_WIRING_STRICT_MODE", True),
            enable_observability=get_bool("SAAAAAA_ENABLE_OBSERVABILITY", True),
            enable_metrics=get_bool("SAAAAAA_ENABLE_METRICS", True),
            deterministic_mode=get_bool("SAAAAAA_DETERMINISTIC_MODE", True),
        )

    def to_dict(self) -> dict[str, bool]:
        """Convert flags to dictionary.

        Returns:
            Dictionary of flag names to values
        """
        return {
            "use_spc_ingestion": self.use_spc_ingestion,
            "use_cpp_ingestion": self.use_cpp_ingestion,  # Legacy compatibility
            "enable_http_signals": self.enable_http_signals,
            "allow_threshold_override": self.allow_threshold_override,
            "wiring_strict_mode": self.wiring_strict_mode,
            "enable_observability": self.enable_observability,
            "enable_metrics": self.enable_metrics,
            "deterministic_mode": self.deterministic_mode,
        }

    def validate(self) -> list[str]:
        """Validate flag combinations for conflicts.

        Returns:
            List of validation warnings (empty if valid)
        """
        warnings = []

        if self.enable_http_signals and self.deterministic_mode:
            warnings.append(
                "enable_http_signals=True with deterministic_mode=True may cause "
                "non-determinism due to HTTP variability. Consider using memory:// only."
            )

        if not self.wiring_strict_mode:
            warnings.append(
                "wiring_strict_mode=False disables contract validation. "
                "This is NOT recommended for production."
            )

        if not self.enable_observability and not self.enable_metrics:
            warnings.append(
                "Both observability and metrics are disabled. "
                "Debugging will be difficult without instrumentation."
            )

        return warnings


# Default flags instance for convenience
DEFAULT_FLAGS = WiringFeatureFlags()


__all__ = [
    'WiringFeatureFlags',
    'DEFAULT_FLAGS',
]



================================================================================
FILE: src/farfan_pipeline/core/wiring/observability.py
================================================================================

"""Observability instrumentation for wiring system.

Provides OpenTelemetry tracing and structured logging for all wiring operations.
"""

from __future__ import annotations

import time
from contextlib import contextmanager
from typing import TYPE_CHECKING, Any

import structlog
```

```

if TYPE_CHECKING:
    from collections.abc import Iterator

try:
    from opentelemetry import trace
    from opentelemetry.trace import Status, StatusCode

    HAS_OTEL = True
    tracer = trace.get_tracer("farfan_core.wiring")
except ImportError:
    HAS_OTEL = False
    tracer = None


logger = structlog.get_logger(__name__)


@contextmanager
def trace_wiring_link(
    link_name: str,
    **attributes: Any,
) -> Iterator[dict[str, Any]]:
    """Trace a wiring link operation.

    Creates an OpenTelemetry span (if available) and logs structured messages.

    Args:
        link_name: Name of the wiring link (e.g., "cpp->adapter")
        **attributes: Additional attributes to include in span/log

    Yields:
        Dict for adding dynamic attributes during operation

    Example:
        with trace_wiring_link("cpp->adapter", document_id="doc123") as attrs:
            result = adapter.convert(cpp)
            attrs["chunk_count"] = len(result.chunks)
    """
    start_time = time.time()
    dynamic_attrs: dict[str, Any] = {}

    # Start span if OpenTelemetry is available
    span = None
    if HAS_OTEL and tracer:
        span = tracer.start_span(f"wiring.link.{link_name}")
        span.set_attribute("link", link_name)
        for key, value in attributes.items():
            if isinstance(value, (str, int, float, bool)):
                span.set_attribute(key, value)

    # Log start
    logger.info(
        "wiring_link_start",
        link=link_name,
        **attributes,
    )

    try:
        yield dynamic_attrs

        # Success
        latency_ms = (time.time() - start_time) * 1000

        if span:
            span.set_attribute("latency_ms", latency_ms)
            span.set_attribute("ok", True)
            for key, value in dynamic_attrs.items():
                if isinstance(value, (str, int, float, bool)):
                    span.set_attribute(key, value)
            span.set_status(Status(StatusCode.OK))

        logger.info(
            "wiring_link_complete",
            link=link_name,
            latency_ms=latency_ms,
            ok=True,
            **attributes,
            **dynamic_attrs,
        )

    except Exception as e:
        # Failure
        latency_ms = (time.time() - start_time) * 1000

        if span:
            span.set_attribute("latency_ms", latency_ms)
            span.set_attribute("ok", False)
            span.set_attribute("error_type", type(e).__name__)
            span.set_attribute("error_message", str(e))
            span.set_status(Status(StatusCode.ERROR, str(e)))

        logger.error(
            "wiring_link_failed",
            link=link_name,
            latency_ms=latency_ms,
            ok=False,
            error_type=type(e).__name__,
            error_message=str(e),
            **attributes,
        )

        raise

    finally:
        if span:
            span.end()


@contextmanager
def trace_wiring_init(
    phase: str,
    **attributes: Any,
) -> Iterator[dict[str, Any]]:
    """Trace a wiring initialization phase.

    Args:
        phase: Name of initialization phase
        **attributes: Additional attributes

    Yields:
        Dict for adding dynamic attributes
    """
    start_time = time.time()
    dynamic_attrs: dict[str, Any] = {}

    span = None
    if HAS_OTEL and tracer:
        span = tracer.start_span(f"wiring.init.{phase}")
        span.set_attribute("phase", phase)
        for key, value in attributes.items():
            if isinstance(value, (str, int, float, bool)):
                span.set_attribute(key, value)

    logger.info(
        "wiring_init_start",
        phase=phase,
        **attributes,
    )

    try:
        yield dynamic_attrs

        latency_ms = (time.time() - start_time) * 1000

        if span:
            span.set_attribute("latency_ms", latency_ms)
            span.set_attribute("ok", True)
            for key, value in dynamic_attrs.items():
                if isinstance(value, (str, int, float, bool)):
                    span.set_attribute(key, value)
            span.set_status(Status(StatusCode.OK))

        logger.info(
            "wiring_init_complete",
            phase=phase,
            latency_ms=latency_ms,
            ok=True,
            **attributes,
            **dynamic_attrs,
        )

    except Exception as e:
        latency_ms = (time.time() - start_time) * 1000

        if span:
            span.set_attribute("latency_ms", latency_ms)
            span.set_attribute("ok", False)
            span.set_attribute("error_type", type(e).__name__)
            span.set_attribute("error_message", str(e))
            span.set_status(Status(StatusCode.ERROR, str(e)))

        logger.error(
            "wiring_init_failed",
            phase=phase,
            latency_ms=latency_ms,
            ok=False,
            error_type=type(e).__name__,
            error_message=str(e),
            **attributes,
        )

        raise

    finally:
        if span:
            span.end()


def log_wiring_metric(
    metric_name: str,
    value: float | int,
    **labels: Any,
) -> None:
    """Log a wiring metric.

    Args:
        metric_name: Name of the metric
        value: Metric value
        **labels: Metric labels
    """
    logger.info(
        "wiring_metric",
        metric=metric_name,
        value=value,
        **labels,
    )


__all__ = [
    'trace_wiring_link',
    'trace_wiring_init',
    'log_wiring_metric',
    'HAS_OTEL',
]



================================================================================
FILE: src/farfan_pipeline/core/wiring/phase_0_validator.py
================================================================================

"""
Phase 0 Configuration Validator
```

```

This module provides a dedicated validator to enforce the C0-CONFIG-V1.0 contract,
as specified in docs/contracts/C0-CONFIG-V1.0.md. It is executed at the very
beginning of the wiring bootstrap process to ensure the system starts in a
known, valid state.
"""

import os
from pathlib import Path
from typing import Any, Dict, List

class Phase0ValidationError(ValueError):
    """Custom exception for Phase 0 validation errors."""
    def __init__(self, message: str, missing_keys: List[str] | None = None, invalid_paths: Dict[str, str] | None = None):
        super().__init__(message)
        self.missing_keys = missing_keys or []
        self.invalid_paths = invalid_paths or {}

class Phase0Validator:
    """
    Enforces the Phase 0 configuration contract.
    """
    REQUIRED_KEYS = {
        "monolith_path",
        "questionnaire_hash",
        "executor_config_path",
        "calibration_profile",
        "abort_on_insufficient",
        "resource_limits",
    }

    def validate(self, config: Dict[str, Any]) -> None:
        """
        Validates the raw configuration dictionary against the Phase 0 contract.

        Args:
            config: The raw configuration dictionary.

        Raises:
            Phase0ValidationError: If the configuration is invalid.
        """
        self._check_mandatory_keys(config)
        self._check_paths_and_permissions(config)

    def _check_mandatory_keys(self, config: Dict[str, Any]) -> None:
        """Ensures all required configuration keys are present."""
        missing_keys = self.REQUIRED_KEYS - set(config.keys())
        if missing_keys:
            raise Phase0ValidationError(
                "Missing mandatory configuration keys.",
                missing_keys=sorted(list(missing_keys))
            )

    def _check_paths_and_permissions(self, config: Dict[str, Any]) -> None:
        """Validates that file paths exist and have the correct permissions."""
        monolith_path = Path(config["monolith_path"])
        executor_path = Path(config["executor_config_path"])
        invalid_paths = {}

        # Check for existence
        if not monolith_path.exists():
            invalid_paths["monolith_path"] = f"File not found at {monolith_path}"
        if not executor_path.exists():
            invalid_paths["executor_config_path"] = f"File not found at {executor_path}"

        if invalid_paths:
            raise Phase0ValidationError("Invalid file paths in configuration.", invalid_paths=invalid_paths)

        # Check monolith permissions (must be read-only)
        if not os.access(monolith_path, os.R_OK):
            invalid_paths["monolith_path"] = f"File at {monolith_path} is not readable."
            raise Phase0ValidationError(
                "Invalid file permissions in configuration.",
                invalid_paths=invalid_paths
            )
        elif os.access(monolith_path, os.W_OK):
            invalid_paths["monolith_path"] = f"File at {monolith_path} must be read-only."
            raise Phase0ValidationError(
                "Invalid file permissions in configuration.",
                invalid_paths=invalid_paths
            )



================================================================================
FILE: src/farfan_pipeline/core/wiring/validation.py
================================================================================

"""Contract validation between wiring links.

Validates that deliverables from one module match expectations of the next.
All validations use Pydantic models for type safety and prescriptive errors.
"""

from __future__ import annotations

from typing import Any

import blake3
import structlog
from pydantic import BaseModel, ValidationError

from farfan_pipeline.core.wiring.contracts import (
    AdapterExpectation,
    AggregateExpectation,
    ArgRouterExpectation,
    ArgRouterPayloadDeliverable,
    CPPDeliverable,
    SPCDeliverable,
    EnrichedChunkDeliverable,
    ExecutorInputDeliverable,
    FeatureTableDeliverable,
    OrchestratorExpectation,
    PreprocessedDocumentDeliverable,
    ReportExpectation,
    ScoreExpectation,
    ScoresDeliverable,
    SignalPackDeliverable,
    SignalRegistryExpectation,
)
from farfan_pipeline.core.wiring.errors import WiringContractError

logger = structlog.get_logger(__name__)


class LinkValidator:
    """Validator for individual wiring links.

    Validates deliverableâ\206\222expectation contracts and computes hashes for determinism.
    """

    def __init__(self, link_name: str) -> None:
        """Initialize validator for a specific link.

        Args:
            link_name: Name of the link (e.g., "cpp->adapter")
        """
        self.link_name = link_name
        self._validation_count = 0
        self._failure_count = 0

    def validate(
        self,
        deliverable_data: dict[str, Any],
        deliverable_model: type[BaseModel],
        expectation_model: type[BaseModel],
    ) -> None:
        """Validate deliverable matches expectation.

        Args:
            deliverable_data: Actual data being delivered
            deliverable_model: Pydantic model for deliverable
            expectation_model: Pydantic model for expectation

        Raises:
            WiringContractError: If validation fails
        """
        self._validation_count += 1

        # Validate deliverable schema
        try:
            deliverable = deliverable_model.model_validate(deliverable_data)
        except ValidationError as e:
            self._failure_count += 1

            errors = e.errors()
            first_error = errors[0] if errors else {}
            field = ".".join(str(loc) for loc in first_error.get("loc", []))

            raise WiringContractError(
                link=self.link_name,
                expected_schema=deliverable_model.__name__,
                received_schema=type(deliverable_data).__name__,
                field=field or None,
                fix=f"Ensure {self.link_name} produces valid {deliverable_model.__name__}. "
                    f"Error: {first_error.get('msg', 'Unknown')}",
            ) from e

        # Validate expectation schema
        # (This ensures the downstream consumer can handle the deliverable)
        try:
            expectation_model.model_validate(deliverable.model_dump())
        except ValidationError as e:
            self._failure_count += 1

            errors = e.errors()
            first_error = errors[0] if errors else {}
            field = ".".join(str(loc) for loc in first_error.get("loc", []))

            raise WiringContractError(
                link=self.link_name,
                expected_schema=expectation_model.__name__,
                received_schema=deliverable_model.__name__,
                field=field or None,
                fix=f"Deliverable from {self.link_name} does not meet expectations. "
                    f"Error: {first_error.get('msg', 'Unknown')}",
            ) from e

        logger.debug(
            "contract_validated",
            link=self.link_name,
            deliverable=deliverable_model.__name__,
            expectation=expectation_model.__name__,
        )

    def compute_hash(self, data: dict[str, Any]) -> str:
        """Compute deterministic hash of data for this link.

        Args:
            data: Data to hash

        Returns:
            BLAKE3 hash hex string
        """
        import json

        # Sort keys for deterministic hashing
        json_str = json.dumps(data, sort_keys=True, separators=(',', ':'))
        hash_value = blake3.blake3(json_str.encode('utf-8')).hexdigest()

        logger.debug(
            "link_hash_computed",
            link=self.link_name,
            hash=hash_value[:16],
        )

        return hash_value

    def get_metrics(self) -> dict[str, Any]:
        """Get validation metrics.

        Returns:
            Dict with validation_count and failure_count
        """
        return {
            "validation_count": self._validation_count,
            "failure_count": self._failure_count,
            "success_rate": (
                (self._validation_count - self._failure_count) / self._validation_count
                if self._validation_count > 0
                else 1.0
            ),
        }


class WiringValidator:
    """Central validator for all wiring links.

    Provides validation methods for each iâ\206\222i+1 link in the system.
    """

    def __init__(self) -> None:
        """Initialize wiring validator."""
        self._validators = {
            "cpp->adapter": LinkValidator("cpp->adapter"),
            "spc->adapter": LinkValidator("spc->adapter"),
            "adapter->orchestrator": LinkValidator("adapter->orchestrator"),
            "orchestrator->argrouter": LinkValidator("orchestrator->argrouter"),
            "argrouter->executors": LinkValidator("argrouter->executors"),
            "signals->registry": LinkValidator("signals->registry"),
            "executors->aggregate": LinkValidator("executors->aggregate"),
            "aggregate->score": LinkValidator("aggregate->score"),
            "score->report": LinkValidator("score->report"),
        }

        logger.info("wiring_validator_initialized", links=len(self._validators))

    def validate_spc_to_adapter(self, spc_data: dict[str, Any]) -> None:
        """Validate SPC â\206\222 Adapter link.

        Args:
            spc_data: SPC deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        from farfan_pipeline.core.wiring.contracts import SPCDeliverable

        validator = self._validators["spc->adapter"]
        validator.validate(
            deliverable_data=spc_data,
            deliverable_model=SPCDeliverable,
            expectation_model=AdapterExpectation,
        )

    def validate_cpp_to_adapter(self, cpp_data: dict[str, Any]) -> None:
        """Validate CPP â\206\222 Adapter link.

        DEPRECATED: Use validate_spc_to_adapter instead.

        Args:
            cpp_data: CPP deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        # Forward to new validator if possible, but keep legacy link name for now
        # to avoid breaking existing hashes if they depend on link name.
        # However, we should warn.
        import warnings
        warnings.warn(
            "validate_cpp_to_adapter is deprecated. Use validate_spc_to_adapter instead.",
            DeprecationWarning,
            stacklevel=2
        )

        validator = self._validators["cpp->adapter"]
        validator.validate(
            deliverable_data=cpp_data,
            deliverable_model=CPPDeliverable,
            expectation_model=AdapterExpectation,
        )

    def validate_adapter_to_orchestrator(
        self,
        preprocessed_doc_data: dict[str, Any],
    ) -> None:
        """Validate Adapter â\206\222 Orchestrator link.

        Args:
            preprocessed_doc_data: PreprocessedDocument deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        validator = self._validators["adapter->orchestrator"]
        validator.validate(
            deliverable_data=preprocessed_doc_data,
            deliverable_model=PreprocessedDocumentDeliverable,
            expectation_model=OrchestratorExpectation,
        )

    def validate_orchestrator_to_argrouter(
        self,
        payload_data: dict[str, Any],
    ) -> None:
        """Validate Orchestrator â\206\222 ArgRouter link.

        Args:
            payload_data: ArgRouter payload deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        validator = self._validators["orchestrator->argrouter"]
        validator.validate(
            deliverable_data=payload_data,
            deliverable_model=ArgRouterPayloadDeliverable,
            expectation_model=ArgRouterExpectation,
        )

    def validate_argrouter_to_executors(
        self,
        executor_input_data: dict[str, Any],
    ) -> None:
        """Validate ArgRouter â\206\222 Executors link.

        Args:
            executor_input_data: Executor input deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        self._validators["argrouter->executors"]
        # Note: ExecutorInput doesn't have a matching expectation model yet
        # For now, just validate the deliverable
        from pydantic import ValidationError

        try:
            ExecutorInputDeliverable.model_validate(executor_input_data)
        except ValidationError as e:
            raise WiringContractError(
                link="argrouter->executors",
                expected_schema=ExecutorInputDeliverable.__name__,
                received_schema=type(executor_input_data).__name__,
                field=str(e.errors()[0].get("loc", [])) if e.errors() else None,
                fix="Ensure ArgRouter produces valid ExecutorInputDeliverable",
            ) from e

    def validate_signals_to_registry(
        self,
        signal_pack_data: dict[str, Any],
    ) -> None:
        """Validate Signals â\206\222 Registry link.

        Args:
            signal_pack_data: SignalPack deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        validator = self._validators["signals->registry"]
        validator.validate(
            deliverable_data=signal_pack_data,
            deliverable_model=SignalPackDeliverable,
            expectation_model=SignalRegistryExpectation,
        )

    def validate_executors_to_aggregate(
        self,
        enriched_chunks_data: list[dict[str, Any]],
    ) -> None:
        """Validate Executors â\206\222 Aggregate link.

        Args:
            enriched_chunks_data: List of enriched chunk deliverables

        Raises:
            WiringContractError: If validation fails
        """
        validator = self._validators["executors->aggregate"]

        # Validate each chunk
        for i, chunk_data in enumerate(enriched_chunks_data):
            try:
                EnrichedChunkDeliverable.model_validate(chunk_data)
            except ValidationError as e:
                raise WiringContractError(
                    link="executors->aggregate",
                    expected_schema=EnrichedChunkDeliverable.__name__,
                    received_schema=type(chunk_data).__name__,
                    field=f"chunk[{i}]",
                    fix=f"Ensure all enriched chunks are valid. Chunk {i} failed validation.",
                ) from e

        # Validate aggregate expectation
        validator.validate(
            deliverable_data={"enriched_chunks": enriched_chunks_data},
            deliverable_model=AggregateExpectation,
            expectation_model=AggregateExpectation,  # Same for now
        )

    def validate_aggregate_to_score(
        self,
        feature_table_data: dict[str, Any],
    ) -> None:
        """Validate Aggregate â\206\222 Score link.

        Args:
            feature_table_data: Feature table deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        validator = self._validators["aggregate->score"]
        validator.validate(
            deliverable_data=feature_table_data,
            deliverable_model=FeatureTableDeliverable,
            expectation_model=ScoreExpectation,
        )

    def validate_score_to_report(
        self,
        scores_data: dict[str, Any],
    ) -> None:
        """Validate Score â\206\222 Report link.

        Args:
            scores_data: Scores deliverable data

        Raises:
            WiringContractError: If validation fails
        """
        validator = self._validators["score->report"]
        validator.validate(
            deliverable_data=scores_data,
            deliverable_model=ScoresDeliverable,
            expectation_model=ReportExpectation,
        )

    def compute_link_hash(self, link_name: str, data: dict[str, Any]) -> str:
        """Compute hash for a specific link.

        Args:
            link_name: Name of the link
            data: Data to hash

        Returns:
            BLAKE3 hash hex string

        Raises:
            KeyError: If link_name is not recognized
        """
        validator = self._validators[link_name]
        return validator.compute_hash(data)

    def get_all_metrics(self) -> dict[str, dict[str, Any]]:
        """Get metrics for all links.

        Returns:
            Dict mapping link names to their metrics
        """
        return {
            link_name: validator.get_metrics()
            for link_name, validator in self._validators.items()
        }

    def get_summary(self) -> dict[str, Any]:
        """Get summary of all validation activity.

        Returns:
            Summary dict with total counts and success rate
        """
        all_metrics = self.get_all_metrics()

        total_validations = sum(m["validation_count"] for m in all_metrics.values())
        total_failures = sum(m["failure_count"] for m in all_metrics.values())

        return {
            "total_validations": total_validations,
            "total_failures": total_failures,
            "overall_success_rate": (
                (total_validations - total_failures) / total_validations
                if total_validations > 0
                else 1.0
            ),
            "links": all_metrics,
        }


__all__ = [
    'LinkValidator',
    'WiringValidator',
]



================================================================================
FILE: src/farfan_pipeline/dashboard_atroz/__init__.py
================================================================================

"""Dashboard Atroz package

This package groups and labels the modules orchestrating the current dashboard
without moving original sources. It provides stable import paths while
preserving existing module locations.
"""

# Re-export key orchestrator components for convenience
from ..api.api_server import app as flask_app  # type: ignore



================================================================================
FILE: src/farfan_pipeline/dashboard_atroz/api_server.py
================================================================================

"""Shim to run the current dashboard Flask API server.

Keeps import paths stable by referencing the existing server implementation
under `farfan_pipeline.api.api_server`.
"""

from ..api.api_server import app

if __name__ == "__main__":
    # Delegate to the existing Flask app run configuration
    app.run(host="0.0.0.0", port=5000, debug=True)

```

```

================================================================================
FILE: src/farfan_pipeline/dashboard_atroz/data_service.py
================================================================================

"""Shim for the dashboard transformer service.

References the implementation in `farfan_pipeline.api.dashboard_data_service`.
"""

from ..api.dashboard_data_service import DashboardDataService

__all__ = ["DashboardDataService"]


```