```
src/farfan_pipeline/phases/Phase_zero/verified_pipeline_runner.py

"""
Verified Pipeline Runner - Phase 0 Orchestrator
================================================

Implements the P00-EN v2.0 specification for pre-execution validation and
deterministic bootstrap. This is the CANONICAL Phase 0 orchestrator.

Architecture:
    Phase 0 consists of 4 strictly sequenced sub-phases with exit gates:

    P0.0: Bootstrap          ? Runtime config, seed registry, manifest builder
    P0.1: Input Verification ? Cryptographic hash validation (SHA-256)
    P0.2: Boot Checks        ? Dependency validation (PROD: fatal, DEV: warn)
    P0.3: Determinism        ? RNG seeding with mandatory python+numpy seeds

    EXIT GATE: self.errors MUST be empty ? _bootstrap_failed = False

Contract:
    - NEVER proceeds with partial configurations
    - NEVER uses defaults on config errors
    - ALWAYS fails fast and clean
    - ALWAYS generates failure manifest on abort

Questionnaire Access Architecture:
    Phase 0 ONLY validates questionnaire file integrity (SHA-256 hash).

    CRITICAL: Phase 0 does NOT load or parse questionnaire content.

    Questionnaire access hierarchy (per factory.py):
        Level 1: AnalysisPipelineFactory (ONLY owner, loads CanonicalQuestionnaire)
        Level 2: QuestionnaireResourceProvider (scoped access, no I/O)
        Level 3: Orchestrator (accesses via Provider)
        Level 4: Signals (alternative access path)

     Phase 0 validates FILE INTEGRITY only, NOT content. Factory loads after Phase 0
passes.

Author: Phase 0 Compliance Team
Version: 2.0.1
Specification: P00-EN v2.0
"""

from __future__ import annotations

import json
from datetime import datetime
from pathlib import Path
from typing import Any

from canonic_phases.Phase_zero.boot_checks import BootCheckError, run_boot_checks
from canonic_phases.Phase_zero.determinism import initialize_determinism_from_registry
from canonic_phases.Phase_zero.exit_gates import (
```

```
    check_all_gates,
    get_gate_summary,
)
from canonic_phases.Phase_zero.runtime_config import RuntimeConfig


class VerifiedPipelineRunner:
    """
    Phase 0 orchestrator with strict contract enforcement.

    This class implements the complete Phase 0 validation sequence as specified
    in P00-EN v2.0. It MUST be the first component executed in the pipeline.

    Attributes:
        plan_pdf_path: Path to input policy plan PDF
        questionnaire_path: Path to questionnaire monolith JSON
        artifacts_dir: Directory for output artifacts
        execution_id: Unique execution identifier (timestamp-based)
        errors: List of accumulated errors (MUST be empty for success)
        _bootstrap_failed: Flag indicating bootstrap failure
        runtime_config: Validated runtime configuration
        seed_snapshot: Dictionary of applied seeds (populated in P0.3)
        input_pdf_sha256: SHA-256 hash of input PDF (populated in P0.1)
        questionnaire_sha256: SHA-256 hash of questionnaire (populated in P0.1)
    """

    def __init__(
        self,
        plan_pdf_path: Path,
        artifacts_dir: Path,
        questionnaire_path: Path,
    ):
        """
        P0.0: Bootstrap - Initialize core runner infrastructure.

        This method initializes runtime configuration, seed registry, and
        manifest builder. It does NOT seed RNGs (that happens in P0.3).

        CRITICAL: Phase 0 only validates file integrity (hashing).
                  Factory loads questionnaire AFTER Phase 0 passes.

        Args:
            plan_pdf_path: Path to input policy plan PDF
            artifacts_dir: Directory for output artifacts
            questionnaire_path: Path to questionnaire file for hash validation
                                (Factory will load content after Phase 0)

        Postconditions:
            - self.runtime_config set (or None with _bootstrap_failed=True)
            - self.artifacts_dir exists
            - self.seed_registry initialized
            - self.errors empty (or populated with bootstrap errors)
            - self._bootstrap_failed reflects init status
        """
```

```
        self.plan_pdf_path = plan_pdf_path
        self.artifacts_dir = artifacts_dir
        self.questionnaire_path = questionnaire_path  # For hash validation ONLY
        self.errors: list[str] = []
        self._bootstrap_failed: bool = False

        # Generate execution identifiers
        self.execution_id = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
        self.policy_unit_id = f"policy_unit::{self.plan_pdf_path.stem}"
        self.correlation_id = self.execution_id

        # Initialize seed registry (but don't seed yet - that's P0.3)
        from orchestration.seed_registry import get_global_seed_registry
        self.seed_registry = get_global_seed_registry()
        self.seed_snapshot: dict[str, int] = {}  # Populated in P0.3

        # Initialize attributes for P0.1
        self.input_pdf_sha256: str = ""
        self.questionnaire_sha256: str = ""

        # P0.0.1: Load runtime configuration
        self.runtime_config: RuntimeConfig | None = None
        try:
            self.runtime_config = RuntimeConfig.from_env()
            self._log_bootstrap("runtime_config_loaded", {
                "mode": self.runtime_config.mode.value,
                "strict": self.runtime_config.is_strict_mode(),
            })
        except Exception as e:
            self._log_bootstrap("runtime_config_failed", {"error": str(e)})
            self.errors.append(f"Failed to load runtime config: {e}")
            self._bootstrap_failed = True

        # P0.0.2: Create artifacts directory
        try:
            self.artifacts_dir.mkdir(parents=True, exist_ok=True)
            self._log_bootstrap("artifacts_dir_created", {
                "path": str(self.artifacts_dir)
            })
        except Exception as e:
            self._log_bootstrap("artifacts_dir_failed", {"error": str(e)})
            self.errors.append(f"Failed to create artifacts directory: {e}")
            self._bootstrap_failed = True

        # P0.0.3: Bootstrap complete
        if not self._bootstrap_failed:
            self._log_bootstrap("bootstrap_complete", {
                "execution_id": self.execution_id,
                "policy_unit_id": self.policy_unit_id,
            })

    def _log_bootstrap(self, event: str, data: dict[str, Any]) -> None:
        """Log bootstrap event (minimal logging during bootstrap)."""
        # In production, this would use structured logging
```

```
    # For now, we keep it simple
    print(f"[BOOTSTRAP] {event}: {data}", flush=True)

def _compute_sha256_streaming(self, file_path: Path) -> str:
    """
    Compute SHA-256 hash of file using streaming read.

    Args:
        file_path: Path to file

    Returns:
        Hex-encoded SHA-256 hash
    """
    import hashlib
    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)
    return sha256_hash.hexdigest()

def verify_input(self, expected_hashes: dict[str, str] | None = None) -> bool:
    """
    P0.1: Input Verification - Hash and validate input files.

    Computes SHA-256 hashes for FILE INTEGRITY only:
    - Input policy plan PDF
    - Questionnaire monolith JSON (file integrity, NOT content parsing)

    CRITICAL: Phase 0 does NOT load or parse questionnaire content.
              This validates FILE INTEGRITY only. Factory loads content
              AFTER Phase 0 passes via load_questionnaire().

    Optionally validates against expected hashes for tamper detection.

    Args:
        expected_hashes: Optional dict with keys 'pdf' and 'questionnaire'
                         containing expected SHA-256 hashes (64-char hex)

    Returns:
        True if all inputs verified successfully

    Postconditions:
        - self.input_pdf_sha256 set (or error appended)
        - self.questionnaire_sha256 set (or error appended)
        - Files exist and are readable
        - Hashes match expected values (if provided)
        - Content NOT loaded (deferred to Factory)

    Specification:
        P00-EN v2.0 Section 3.2
    """
    print("[P0.1] Starting input verification", flush=True)

    if expected_hashes:
```

```
                print(f"[P0.1] Hash validation enabled (will verify against expected
values)", flush=True)

        # Verify PDF exists and hash
        if not self.plan_pdf_path.exists():
            error = f"Input PDF not found: {self.plan_pdf_path}"
            self.errors.append(error)
            print(f"[P0.1] ERROR: {error}", flush=True)
            return False

        try:
            self.input_pdf_sha256 = self._compute_sha256_streaming(self.plan_pdf_path)
            print(f"[P0.1] PDF hashed: {self.input_pdf_sha256[:16]}...", flush=True)
        except Exception as e:
            error = f"Failed to hash PDF: {e}"
            self.errors.append(error)
            print(f"[P0.1] ERROR: {error}", flush=True)
            return False

        # Verify questionnaire file exists and hash (FILE INTEGRITY ONLY)
        # NOTE: Phase 0 does NOT parse content. Factory loads after Phase 0 passes.
        if not self.questionnaire_path.exists():
            error = f"Questionnaire file not found: {self.questionnaire_path}"
            self.errors.append(error)
            print(f"[P0.1] ERROR: {error}", flush=True)
            return False

        try:
                                                    self.questionnaire_sha256    =
self._compute_sha256_streaming(self.questionnaire_path)
                        print(f"[P0.1]  Questionnaire  file  hashed  (integrity  only):
{self.questionnaire_sha256[:16]}...", flush=True)
             print(f"[P0.1] ??  Content will be loaded by Factory after Phase 0 passes",
flush=True)
        except Exception as e:
            error = f"Failed to hash questionnaire file: {e}"
            self.errors.append(error)
            print(f"[P0.1] ERROR: {error}", flush=True)
            return False

        # Optional: Validate against expected hashes (tamper detection)
        if expected_hashes:
            validation_passed = True

            if 'pdf' in expected_hashes:
                expected_pdf = expected_hashes['pdf']
                if self.input_pdf_sha256 != expected_pdf:
                    error = (
                        f"PDF hash mismatch! "
                        f"Expected: {expected_pdf[:16]}... "
                        f"Got: {self.input_pdf_sha256[:16]}... "
                        f"(possible tampering detected)"
                    )
                    self.errors.append(error)
```

```
                    print(f"[P0.1] ERROR: {error}", flush=True)
                    validation_passed = False
                else:
                          print(f"[P0.1] ? PDF hash validated against expected value",
flush=True)

            if 'questionnaire' in expected_hashes:
                expected_q = expected_hashes['questionnaire']
                if self.questionnaire_sha256 != expected_q:
                    error = (
                        f"Questionnaire hash mismatch! "
                        f"Expected: {expected_q[:16]}... "
                        f"Got: {self.questionnaire_sha256[:16]}... "
                        f"(possible tampering detected)"
                    )
                    self.errors.append(error)
                    print(f"[P0.1] ERROR: {error}", flush=True)
                    validation_passed = False
                else:
                          print(f"[P0.1] ? Questionnaire hash validated against expected
value", flush=True)

            if not validation_passed:
                return False

        print("[P0.1] Input verification complete", flush=True)
        return True

    def run_boot_checks(self) -> bool:
        """
        P0.2: Boot Checks - Validate system dependencies.

        Runs boot-time validation checks for:
        - Python version compatibility
        - Critical package availability
        - Calibration file presence
        - spaCy model availability

        In PROD mode: Failures are FATAL (raises BootCheckError)
        In DEV mode: Failures log warnings but allow continuation

        Returns:
            True if checks passed or warnings allowed

        Raises:
            BootCheckError: If critical check fails in PROD mode

        Specification:
            P00-EN v2.0 Section 3.3
        """
        print("[P0.2] Starting boot checks", flush=True)

        if self.runtime_config is None:
            error = "Runtime config not available for boot checks"
```

```
            self.errors.append(error)
            raise BootCheckError("runtime_config", error, "CONFIG_MISSING")

        try:
            results = run_boot_checks(self.runtime_config)

            # Print results summary
            passed = sum(1 for v in results.values() if v)
            total = len(results)
            print(f"[P0.2] Boot checks: {passed}/{total} passed", flush=True)

            for check, success in results.items():
                status = "?" if success else "?"
                print(f"[P0.2]   {status} {check}", flush=True)

            return True

        except BootCheckError as e:
            # In PROD, this is fatal
            if self.runtime_config.mode.value == "prod":
                error = f"Boot check failed: {e}"
                self.errors.append(error)
                print(f"[P0.2] FATAL: {error}", flush=True)
                raise

            # In DEV, log warning and continue
             print(f"[P0.2] WARNING: {e} (continuing in {self.runtime_config.mode.value}
mode)", flush=True)
            return False

    def initialize_determinism(self) -> bool:
        """
        P0.3: Determinism Context - Seed all RNGs.

        Seeds random number generators for:
        - Python random module (MANDATORY)
        - NumPy random state (MANDATORY)
        - Quantum optimizer (optional)
        - Neuromorphic controller (optional)
        - Meta-learner strategy (optional)

        Returns:
            True if all mandatory seeds applied successfully

        Postconditions:
            - self.seed_snapshot populated with seed values
            - Python random.seed() called
            - NumPy np.random.seed() called
            - Errors appended if seeding fails

        Specification:
            P00-EN v2.0 Section 3.4
        """
        print("[P0.3] Initializing determinism context", flush=True)
```

```
        seeds, status, errors = initialize_determinism_from_registry(
            self.seed_registry,
            self.policy_unit_id,
            self.correlation_id
        )

        if errors:
            self.errors.extend(errors)
            self._bootstrap_failed = True
                  print(f"[P0.3] FATAL: Determinism initialization failed: {errors}",
flush=True)
            return False

        # Store snapshot for validation
        self.seed_snapshot = seeds

        # Log seeding success
        print(f"[P0.3] Seeds applied:", flush=True)
        for component, seed_value in seeds.items():
            applied = status.get(component, False)
            marker = "?" if applied else "?"
            print(f"[P0.3]   {marker} {component}: {seed_value}", flush=True)

        print("[P0.3] Determinism context initialized", flush=True)
        return True

    async def run_phase_zero(self) -> bool:
        """
        Execute complete Phase 0 with strict exit gate enforcement.

        Sequence:
            1. Check Gate 1 (Bootstrap)
            2. Execute P0.1 (Input Verification)
            3. Check Gate 2 (Input Verification)
            4. Execute P0.2 (Boot Checks)
            5. Check Gate 3 (Boot Checks)
            6. Execute P0.3 (Determinism)
            7. Check Gate 4 (Determinism)

        Returns:
            True if Phase 0 completed successfully (all gates passed)

        Exit Condition:
            self.errors == [] AND self._bootstrap_failed == False

        Specification:
            P00-EN v2.0 Section 4.1 - Exit Conditions & Guarantees
        """
        import asyncio

        print("\n" + "="*80, flush=True)
        print("PHASE 0: PRE-EXECUTION VALIDATION & DETERMINISTIC BOOTSTRAP", flush=True)
        print("="*80 + "\n", flush=True)
```

```
# Check all gates (bootstrap already happened in __init__)
all_passed, gate_results = check_all_gates(self)

# If bootstrap failed, abort immediately
if gate_results and not gate_results[0].passed:
    print("\n? Gate 1 (Bootstrap) FAILED", flush=True)
    print(get_gate_summary(gate_results), flush=True)
    return False

# P0.1: Input Verification
if not await asyncio.to_thread(self.verify_input):
    print("\n? P0.1 Input Verification FAILED", flush=True)
    return False

# Gate 2: Input Verification
all_passed, gate_results = check_all_gates(self)
if not gate_results[1].passed:
    print("\n? Gate 2 (Input Verification) FAILED", flush=True)
    print(get_gate_summary(gate_results), flush=True)
    return False

# P0.2: Boot Checks
try:
    await asyncio.to_thread(self.run_boot_checks)
except BootCheckError:
    print("\n? P0.2 Boot Checks FAILED (PROD mode)", flush=True)
    return False

# Gate 3: Boot Checks
all_passed, gate_results = check_all_gates(self)
if not gate_results[2].passed:
    print("\n? Gate 3 (Boot Checks) FAILED", flush=True)
    print(get_gate_summary(gate_results), flush=True)
    return False

# P0.3: Determinism
if not await asyncio.to_thread(self.initialize_determinism):
    print("\n? P0.3 Determinism Initialization FAILED", flush=True)
    return False

# Gate 4: Determinism (FINAL GATE)
all_passed, gate_results = check_all_gates(self)
if not gate_results[3].passed:
    print("\n? Gate 4 (Determinism) FAILED", flush=True)
    print(get_gate_summary(gate_results), flush=True)
    return False

# ALL GATES PASSED
print("\n" + "="*80, flush=True)
print("? PHASE 0 COMPLETE - ALL GATES PASSED", flush=True)
print("="*80, flush=True)
print(get_gate_summary(gate_results), flush=True)
print("="*80 + "\n", flush=True)
```

```
        return True

    def generate_failure_manifest(self) -> Path:
        """
        Generate failure manifest when Phase 0 fails.

        Returns:
            Path to verification_manifest.json

        Specification:
            P00-EN v2.0 Section 4.2 - Failure Manifest Generation
        """
        manifest = {
            "success": False,
            "execution_id": self.execution_id,
            "start_time": datetime.utcnow().isoformat(),
            "end_time": datetime.utcnow().isoformat(),
            "errors": self.errors,
            "phases_completed": 0,
            "phases_failed": 1,
            "artifacts_generated": [],
            "artifact_hashes": {},
            "input_pdf_path": str(self.plan_pdf_path),
            "input_pdf_sha256": self.input_pdf_sha256 or "NOT_COMPUTED",
            "questionnaire_file_path": str(self.questionnaire_path),
            "questionnaire_file_sha256": self.questionnaire_sha256 or "NOT_COMPUTED",
             "note": "Phase 0 validates file integrity only. Factory loads content after
Phase 0 passes.",
        }

        manifest_path = self.artifacts_dir / "verification_manifest.json"
        with open(manifest_path, "w") as f:
            json.dump(manifest, f, indent=2)

        print(f"\nFailure manifest written to: {manifest_path}", flush=True)
        print("PIPELINE_VERIFIED=0", flush=True)

        return manifest_path


__all__ = ["VerifiedPipelineRunner"]
```

src/farfan_pipeline/phases/__init__.py

```
"""Canonical pipeline phases (Phase 0-9).

This package was formerly exposed as the top-level `canonic_phases` package.
New code should import from `farfan_pipeline.phases`.
"""

__all__ = [
    "Phase_zero",
    "Phase_one",
    "Phase_two",
    "Phase_three",
    "Phase_four_five_six_seven",
    "Phase_eight",
    "Phase_nine",
]
```

src/farfan_pipeline/processing/__init__.py

src/farfan_pipeline/processing/aggregation_provenance.py

```
"""Stub for aggregation_provenance to allow imports."""

from dataclasses import dataclass, field
from typing import Any

@dataclass
class ProvenanceNode:
    node_id: str
    level: str
    score: float
    quality_level: str
    metadata: dict[str, Any] = field(default_factory=dict)

class AggregationDAG:
    def __init__(self):
        self.nodes = {}

    def add_node(self, node):
        self.nodes[node.node_id] = node

    def add_aggregation_edge(self, source_ids, target_id, operation, weights, metadata):
        pass
```

src/farfan_pipeline/processing/choquet_adapter.py

```
"""Stub for choquet_adapter to allow imports."""

class ChoquetProcessingAdapter:
    def __init__(self, n_layers):
        self.n_layers = n_layers

    def aggregate(self, scores, weights=None):
        if weights is None:
            return sum(scores) / len(scores) if scores else 0.0
        return sum(s * w for s, w in zip(scores, weights))

def create_default_choquet_adapter(n_layers):
    return ChoquetProcessingAdapter(n_layers)
```

src/farfan_pipeline/processing/uncertainty_quantification.py

```
"""Stub for uncertainty_quantification to allow imports."""

from dataclasses import dataclass
from typing import Any

@dataclass
class UncertaintyMetrics:
    mean: float
    std: float
    confidence_interval_95: tuple[float, float]
    epistemic_uncertainty: float = 0.0
    aleatoric_uncertainty: float = 0.0

    def to_dict(self):
        return {
            'mean': self.mean,
            'std': self.std,
            'confidence_interval_95': self.confidence_interval_95,
            'epistemic_uncertainty': self.epistemic_uncertainty,
            'aleatoric_uncertainty': self.aleatoric_uncertainty,
        }

class BootstrapAggregator:
    def __init__(self, n_samples=1000, random_seed=42):
        pass

def aggregate_with_uncertainty(scores, weights=None, n_bootstrap=1000, random_seed=42):
    mean = sum(scores) / len(scores) if scores else 0.0
    return mean, UncertaintyMetrics(
        mean=mean,
        std=0.0,
        confidence_interval_95=(mean, mean),
        epistemic_uncertainty=0.0,
        aleatoric_uncertainty=0.0
    )
```

src/farfan_pipeline/synchronization.py

```
from __future__ import annotations

import hashlib
import json
import logging
import re
from dataclasses import dataclass
from typing import Any, Mapping, Sequence

logger = logging.getLogger(__name__)

_PA_RE = re.compile(r"^PA(0[1-9]|10)$")
_DIM_RE = re.compile(r"^DIM0[1-6]$")
_CHUNK_ID_RE = re.compile(r"^PA(0[1-9]|10)-DIM0[1-6]$")


def _get_mapping_value(obj: Any, key: str) -> Any:  # noqa: ANN401
    if isinstance(obj, Mapping):
        return obj.get(key)
    return None


def _get_attr_or_key(obj: Any, name: str) -> Any:  # noqa: ANN401
    value = _get_mapping_value(obj, name)
    if value is not None:
        return value
    return getattr(obj, name, None)


def _coerce_id(value: Any) -> str | None:  # noqa: ANN401
    if isinstance(value, str):
        return value
    if hasattr(value, "value") and isinstance(value.value, str):
        return value.value
    return None


def _extract_policy_area_id(chunk: Any) -> str:
    pa_id = _coerce_id(_get_attr_or_key(chunk, "policy_area_id"))
    if pa_id is None:
        pa_id = _coerce_id(_get_attr_or_key(chunk, "policy_area"))
    if pa_id is None:
        chunk_id = _coerce_id(_get_attr_or_key(chunk, "chunk_id")) or _coerce_id(
            _get_attr_or_key(chunk, "id")
        )
        if chunk_id:
            normalized = chunk_id.replace("_", "-")
            if _CHUNK_ID_RE.match(normalized):
                return normalized.split("-", 1)[0]
    if pa_id is None:
        raise ValueError("Chunk missing policy_area_id")
    return pa_id
```

```
def _extract_dimension_id(chunk: Any) -> str:
    dim_id = _coerce_id(_get_attr_or_key(chunk, "dimension_id"))
    if dim_id is None:
        dim_id = _coerce_id(_get_attr_or_key(chunk, "dimension"))
    if dim_id is None:
        dim_id = _coerce_id(_get_attr_or_key(chunk, "dimension_causal"))
    if dim_id is None:
        chunk_id = _coerce_id(_get_attr_or_key(chunk, "chunk_id")) or _coerce_id(
            _get_attr_or_key(chunk, "id")
        )
        if chunk_id:
            normalized = chunk_id.replace("_", "-")
            if _CHUNK_ID_RE.match(normalized):
                return normalized.split("-", 1)[1]
    if dim_id is None:
        raise ValueError("Chunk missing dimension_id")
    return dim_id


def _extract_text(chunk: Any) -> str:
    text = _get_attr_or_key(chunk, "text")
    if not isinstance(text, str):
        raise ValueError(f"Chunk text must be str, got {type(text).__name__}")
    return text


def _extract_document_position(chunk: Any) -> tuple[int, int] | None:
    start = _get_attr_or_key(chunk, "start_offset")
    end = _get_attr_or_key(chunk, "end_offset")
    if isinstance(start, int) and isinstance(end, int):
        if start < 0 or end < start:
            raise ValueError(f"Invalid document position: ({start}, {end})")
        return (start, end)

    start = _get_attr_or_key(chunk, "start_pos")
    end = _get_attr_or_key(chunk, "end_pos")
    if isinstance(start, int) and isinstance(end, int):
        if start < 0 or end < start:
            raise ValueError(f"Invalid document position: ({start}, {end})")
        return (start, end)

    text_span = _get_attr_or_key(chunk, "text_span")
    if text_span is not None:
        start = getattr(text_span, "start", None)
        end = getattr(text_span, "end", None)
        if isinstance(start, int) and isinstance(end, int):
            if start < 0 or end < start:
                raise ValueError(f"Invalid document position: ({start}, {end})")
            return (start, end)

    return None
```

```
def _validate_ids(policy_area_id: str, dimension_id: str) -> None:
    if not _PA_RE.match(policy_area_id):
        raise ValueError(f"Invalid policy_area_id: {policy_area_id}")
    if not _DIM_RE.match(dimension_id):
        raise ValueError(f"Invalid dimension_id: {dimension_id}")


def _validate_chunk_identity(
    chunk: Any,
    *,
    expected_chunk_id: str,
) -> None:
    if not _CHUNK_ID_RE.match(expected_chunk_id):
        raise ValueError(f"Invalid chunk_id format: {expected_chunk_id}")

    declared = _coerce_id(_get_attr_or_key(chunk, "chunk_id"))
    if declared is not None and declared != expected_chunk_id:
        raise ValueError(
               f"Chunk identity mismatch: expected chunk_id={expected_chunk_id} but got
{declared}"
        )

    legacy_id = _coerce_id(_get_attr_or_key(chunk, "id"))
    if legacy_id is not None:
        normalized = legacy_id.replace("_", "-")
        if normalized != expected_chunk_id:
            raise ValueError(
                 f"Chunk identity mismatch: expected chunk_id={expected_chunk_id} but got
id={legacy_id}"
            )


@dataclass(frozen=True, slots=True)
class SmartPolicyChunk:
    chunk_id: str
    policy_area_id: str
    dimension_id: str
    text: str
    document_position: tuple[int, int] | None
    raw_chunk: Any | None = None

    @property
    def start_pos(self) -> int | None:
        if self.document_position is None:
            return None
        return self.document_position[0]

    @property
    def end_pos(self) -> int | None:
        if self.document_position is None:
            return None
        return self.document_position[1]
```

```
class ChunkMatrix:
    """60-slot PA×DIM chunk matrix with strict invariant validation."""

    EXPECTED_CHUNK_COUNT = 60

    def __init__(self, document: Any) -> None:  # noqa: ANN401
        self._preprocessed_document = document
        chunks = self._extract_chunks(document)
        self._chunk_matrix = self._build_matrix(chunks)
        self._matrix_keys_sorted = tuple(sorted(self._chunk_matrix.keys()))
        self._integrity_hash = self._compute_integrity_hash()

    @property
    def chunk_matrix(self) -> dict[tuple[str, str], SmartPolicyChunk]:
        return dict(self._chunk_matrix)

    @property
    def matrix_keys_sorted(self) -> tuple[tuple[str, str], ...]:
        return self._matrix_keys_sorted

    @property
    def integrity_hash(self) -> str:
        return self._integrity_hash

    def get_chunk(self, policy_area_id: str, dimension_id: str) -> SmartPolicyChunk:
        return self._chunk_matrix[(policy_area_id, dimension_id)]

    @staticmethod
    def _extract_chunks(document: Any) -> list[Any]:  # noqa: ANN401
        if document is None:
            raise ValueError("document is required")

        chunks = _get_attr_or_key(document, "chunks")
        if isinstance(chunks, Sequence) and not isinstance(chunks, (str, bytes)):
            return list(chunks)

        chunk_graph = _get_attr_or_key(document, "chunk_graph")
        if chunk_graph is not None:
            graph_chunks = _get_attr_or_key(chunk_graph, "chunks")
            if isinstance(graph_chunks, Mapping):
                return list(graph_chunks.values())

        if isinstance(document, Sequence) and not isinstance(document, (str, bytes)):
            return list(document)

        raise TypeError(
            "Unsupported document type for ChunkMatrix; expected .chunks sequence, "
            ".chunk_graph.chunks mapping, or a sequence of chunks"
        )

    @classmethod
          def   _build_matrix(cls,   chunks:   Sequence[Any])   ->   dict[tuple[str,   str],
SmartPolicyChunk]:
```

```
        chunk_matrix: dict[tuple[str, str], SmartPolicyChunk] = {}

        inserted_count = 0
        for chunk in chunks:
            inserted_count += 1
            policy_area_id = _extract_policy_area_id(chunk)
            dimension_id = _extract_dimension_id(chunk)
            _validate_ids(policy_area_id, dimension_id)

            expected_chunk_id = f"{policy_area_id}-{dimension_id}"
            _validate_chunk_identity(chunk, expected_chunk_id=expected_chunk_id)

            text = _extract_text(chunk)
            if not text.strip():
                raise ValueError(f"Chunk {expected_chunk_id} has empty text")

            document_position = _extract_document_position(chunk)

            key = (policy_area_id, dimension_id)
            if key in chunk_matrix:
                raise ValueError(
                    f"Duplicate chunk slot detected for key={key}. "
                    f"Inserted={inserted_count}, unique={len(chunk_matrix)}"
                )

            chunk_matrix[key] = SmartPolicyChunk(
                chunk_id=expected_chunk_id,
                policy_area_id=policy_area_id,
                dimension_id=dimension_id,
                text=text,
                document_position=document_position,
                raw_chunk=chunk,
            )

        if len(chunk_matrix) != cls.EXPECTED_CHUNK_COUNT:
            raise ValueError(
                "Chunk Matrix Invariant Violation: Expected 60 unique (PA, DIM) chunks "
                f"but found {len(chunk_matrix)}"
            )

        expected_keys = {
             (f"PA{pa:02d}", f"DIM{dim:02d}") for pa in range(1, 11) for dim in range(1,
7)
        }
        missing = expected_keys - set(chunk_matrix.keys())
        if missing:
            raise ValueError(f"Missing chunk combinations: {sorted(missing)}")

        chunks_per_policy_area = {
                    f"PA{pa:02d}": sum(1 for (pa_id, _) in chunk_matrix if pa_id ==
f"PA{pa:02d}")
            for pa in range(1, 11)
        }
        chunks_per_dimension = {
```

```
            f"DIM{dim:02d}": sum(
                1 for (_, dim_id) in chunk_matrix if dim_id == f"DIM{dim:02d}"
            )
            for dim in range(1, 7)
        }

        logger.info(
            "chunk_matrix_constructed",
            extra={
                "total_chunks": len(chunk_matrix),
                "inserted_count": inserted_count,
                "chunks_per_policy_area": chunks_per_policy_area,
                "chunks_per_dimension": chunks_per_dimension,
            },
        )

        return chunk_matrix

    def _compute_integrity_hash(self) -> str:
        payload = []
        for (pa_id, dim_id) in self._matrix_keys_sorted:
            chunk = self._chunk_matrix[(pa_id, dim_id)]
            text_hash = hashlib.sha256(chunk.text.encode("utf-8")).hexdigest()
            payload.append(
                {
                    "policy_area_id": pa_id,
                    "dimension_id": dim_id,
                    "chunk_id": chunk.chunk_id,
                    "text_sha256": text_hash,
                }
            )

                  json_bytes  =  json.dumps(payload,  sort_keys=True,  separators=(",",
":")).encode("utf-8")
        return hashlib.sha256(json_bytes).hexdigest()


__all__ = ["ChunkMatrix", "SmartPolicyChunk"]
```

src/farfan_pipeline/utils/cpp_adapter.py

```
"""CPP to Orchestrator Adapter.

This adapter converts Canon Policy Package (CPP) documents from the ingestion pipeline
into the orchestrator's PreprocessedDocument format.

Note: This is the canonical adapter implementation.

Design Principles:
- Preserves complete provenance information
- Orders chunks by text_span.start for deterministic ordering
- Computes provenance_completeness metric
- Provides prescriptive error messages on failure
- Supports micro, meso, and macro chunk resolutions
- Optional dependencies handled gracefully (pyarrow, structlog)
"""

from __future__ import annotations

import logging
from datetime import datetime, timezone
from types import MappingProxyType
from typing import Any

from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance

logger = logging.getLogger(__name__)

_EMPTY_MAPPING = MappingProxyType({})


class CPPAdapterError(Exception):
    """Raised when CPP to PreprocessedDocument conversion fails."""

    pass


class CPPAdapter:
    """
    Adapter to convert CanonPolicyPackage (CPP output) to PreprocessedDocument.

    This is the canonical adapter for the FARFAN pipeline, converting the rich
    CanonPolicyPackage data into the format expected by the orchestrator.
    """

    def __init__(self, enable_runtime_validation: bool = True) -> None:
        """Initialize the CPP adapter.

        Args:
                enable_runtime_validation: Enable WiringValidator for runtime contract
checking
        """
```

```
        self.logger = logging.getLogger(self.__class__.__name__)

        # Initialize WiringValidator for runtime contract validation
        self.enable_runtime_validation = enable_runtime_validation
        if enable_runtime_validation:
            try:
                from orchestration.wiring.validation import WiringValidator

                self.wiring_validator = WiringValidator()
                self.logger.info(
                    "WiringValidator enabled for runtime contract checking"
                )
            except ImportError:
                self.logger.warning(
                    "WiringValidator not available. Runtime validation disabled."
                )
                self.wiring_validator = None
        else:
            self.wiring_validator = None

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

        Args:
            canon_package: CanonPolicyPackage from ingestion
            document_id: Unique document identifier

        Returns:
            PreprocessedDocument ready for orchestrator

        Raises:
            CPPAdapterError: If conversion fails or data is invalid

        CanonPolicyPackage Expected Attributes:
            Required:
                - chunk_graph: ChunkGraph with .chunks dict
                - chunk_graph.chunks: dict of chunk objects with .text and .text_span

            Optional (handled with hasattr checks):
                - schema_version: str (default: 'CPP-2025.1')
                - quality_metrics: object with metrics like provenance_completeness,
                  structural_consistency, boundary_f1, kpi_linkage_rate,
                  budget_consistency_score, temporal_robustness, chunk_context_coverage
                - policy_manifest: object with axes, programs, projects, years,
territories
                - metadata: dict with optional 'spc_rich_data' key

            Chunk Optional Attributes (handled with hasattr checks):
                - entities: list of entity objects with .text attribute
                - time_facets: object with .years list
                - budget: object with amount, currency, year, use, source attributes
```

```
"""
self.logger.info(
    f"Converting CanonPolicyPackage to PreprocessedDocument: {document_id}"
)

# === COMPREHENSIVE VALIDATION PHASE (H1.5) ===
# 6-layer validation for robust phase-one output processing

# V1: Validate canon_package exists
if not canon_package:
    raise CPPAdapterError(
        "canon_package is None or empty. "
        "Ensure ingestion completed successfully."
    )

# V2: Validate document_id
if (
    not document_id
    or not isinstance(document_id, str)
    or not document_id.strip()
):
    raise CPPAdapterError(
        f"document_id must be a non-empty string. "
        f"Received: {repr(document_id)}"
    )

# V3: Validate chunk_graph exists
if not hasattr(canon_package, "chunk_graph") or not canon_package.chunk_graph:
    raise CPPAdapterError(
        "canon_package must have a valid chunk_graph. "
        "Check that SmartChunkConverter produced valid output."
    )

chunk_graph = canon_package.chunk_graph

# V4: Validate chunks dict is non-empty
if not chunk_graph.chunks:
    raise CPPAdapterError(
        "chunk_graph.chunks is empty - no chunks to process. "
        "Minimum 1 chunk required from phase-one."
    )

# V5: Validate individual chunks have required attributes
validation_failures = []
for chunk_id, chunk in chunk_graph.chunks.items():
    if not hasattr(chunk, "text"):
        validation_failures.append(
            f"Chunk {chunk_id}: missing 'text' attribute"
        )
    elif not chunk.text or not chunk.text.strip():
        validation_failures.append(
            f"Chunk {chunk_id}: text is empty or whitespace"
        )
```

```
            if not hasattr(chunk, "text_span"):
                validation_failures.append(
                    f"Chunk {chunk_id}: missing 'text_span' attribute"
                )
            elif not hasattr(chunk.text_span, "start") or not hasattr(
                chunk.text_span, "end"
            ):
                validation_failures.append(
                    f"Chunk {chunk_id}: invalid text_span (missing start/end)"
                )

        # V6: Report validation failures with context
        if validation_failures:
            failure_summary = "\n  - ".join(validation_failures)
            raise CPPAdapterError(
                    f"Chunk validation failed ({len(validation_failures)} errors):\n  -
{failure_summary}\n"
                f"Total chunks: {len(chunk_graph.chunks)}\n"
                f"This indicates SmartChunkConverter produced invalid output."
            )

        # Sort chunks by document position for deterministic ordering
        sorted_chunks = sorted(
            chunk_graph.chunks.values(),
            key=lambda c: (
                c.text_span.start if hasattr(c, "text_span") and c.text_span else 0
            ),
        )

        # === PHASE 2 HARDENING: STRICT CARDINALITY & METADATA ===
        # Enforce exactly 60 chunks for CPP canonical documents as per Jobfront 1
        processing_mode = "chunked"
        degradation_reason = None

        if len(sorted_chunks) != 60:
            raise CPPAdapterError(
                    f"Cardinality mismatch: Expected 60 chunks for 'chunked' processing
mode, "
                f"but found {len(sorted_chunks)}. This is a critical violation of the "
                f"CPP canonical format."
            )

        # Enforce metadata integrity
        for idx, chunk in enumerate(sorted_chunks):
            if not hasattr(chunk, "policy_area_id") or not chunk.policy_area_id:
                raise CPPAdapterError(f"Missing policy_area_id in chunk {chunk.id}")
            if not hasattr(chunk, "dimension_id") or not chunk.dimension_id:
                raise CPPAdapterError(f"Missing dimension_id in chunk {chunk.id}")
            if not hasattr(chunk, "chunk_type") or not chunk.chunk_type:
                raise CPPAdapterError(f"Missing chunk_type in chunk {chunk.id}")

        self.logger.info(f"Processing {len(sorted_chunks)} chunks")

        # Build full text by concatenating chunks
```

```
        full_text_parts: list[str] = []
        sentences: list[dict[str, Any]] = []
        sentence_metadata: list[dict[str, Any]] = []
        tables: list[dict[str, Any]] = []
        chunk_index: dict[str, int] = {}
        chunk_summaries: list[dict[str, Any]] = []
        chunks_data: list[ChunkData] = []

        # Track indices for building indexes
        term_index: dict[str, list[int]] = {}
        numeric_index: dict[str, list[int]] = {}
        temporal_index: dict[str, list[int]] = {}
        entity_index: dict[str, list[int]] = {}

        # Track running offset that matches how full_text is built
        current_offset = 0

        provenance_with_data = 0

        for idx, chunk in enumerate(sorted_chunks):
            chunk_text = chunk.text
            chunk_start = current_offset
            chunk_index[chunk.id] = idx

            # Add to full text
            full_text_parts.append(chunk_text)

                # Create sentence entry (each chunk is represented as a sentence for
orchestrator compatibility)
            sentences.append(
                {
                    "text": chunk_text,
                    "chunk_id": chunk.id,
                    "resolution": (
                        chunk.resolution.value.lower()
                        if hasattr(chunk, "resolution")
                        else None
                    ),
                }
            )

            # Create chunk metadata for per-sentence tracking
            chunk_end = chunk_start + len(chunk_text)

            # CRITICAL: Preserve PA×DIM metadata for Phase 2 question routing
            extra_metadata = {
                "chunk_id": chunk.id,
                "policy_area_id": (
                    chunk.policy_area_id if hasattr(chunk, "policy_area_id") else None
                ),
                "dimension_id": (
                    chunk.dimension_id if hasattr(chunk, "dimension_id") else None
                ),
                "resolution": (
```

```
        chunk.resolution.value.lower()
        if hasattr(chunk, "resolution")
        else None
    ),
}

# Add facets if available
if hasattr(chunk, "policy_facets") and chunk.policy_facets:
    extra_metadata["policy_facets"] = {
        "axes": (
            chunk.policy_facets.axes
            if hasattr(chunk.policy_facets, "axes")
            else []
        ),
        "programs": (
            chunk.policy_facets.programs
            if hasattr(chunk.policy_facets, "programs")
            else []
        ),
        "projects": (
            chunk.policy_facets.projects
            if hasattr(chunk.policy_facets, "projects")
            else []
        ),
    }

if hasattr(chunk, "time_facets") and chunk.time_facets:
    extra_metadata["time_facets"] = {
        "years": (
            chunk.time_facets.years
            if hasattr(chunk.time_facets, "years")
            else []
        ),
        "periods": (
            chunk.time_facets.periods
            if hasattr(chunk.time_facets, "periods")
            else []
        ),
    }

if hasattr(chunk, "geo_facets") and chunk.geo_facets:
    extra_metadata["geo_facets"] = {
        "territories": (
            chunk.geo_facets.territories
            if hasattr(chunk.geo_facets, "territories")
            else []
        ),
        "regions": (
            chunk.geo_facets.regions
            if hasattr(chunk.geo_facets, "regions")
            else []
        ),
    }
```

```
sentence_metadata.append(
    {
        "index": idx,
        "page_number": None,
        "start_char": chunk_start,
        "end_char": chunk_end,
        "extra": dict(extra_metadata),
    }
)

chunk_summary = {
    "id": chunk.id,
    "resolution": (
        chunk.resolution.value.lower()
        if hasattr(chunk, "resolution")
        else None
    ),
    "text_span": {"start": chunk_start, "end": chunk_end},
    "policy_area_id": extra_metadata["policy_area_id"],
    "dimension_id": extra_metadata["dimension_id"],
    "has_kpi": hasattr(chunk, "kpi") and chunk.kpi is not None,
    "has_budget": hasattr(chunk, "budget") and chunk.budget is not None,
    "confidence": {
        "layout": (
            getattr(
                chunk.confidence,
                "layout",
                ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L256_66",
                    0.0,
                ),
            )
            if hasattr(chunk, "confidence")
            else ParameterLoaderV2.get(
                "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                "auto_param_L256_108",
                0.0,
            )
        ),
        "ocr": (
            getattr(
                chunk.confidence,
                "ocr",
                ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L257_60",
                    0.0,
                ),
            )
            if hasattr(chunk, "confidence")
            else ParameterLoaderV2.get(
                "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                "auto_param_L257_102",
```

```
                0.0,
            )
        ),
        "typing": (
            getattr(
                chunk.confidence,
                "typing",
                ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L258_66",
                    0.0,
                ),
            )
            if hasattr(chunk, "confidence")
            else ParameterLoaderV2.get(
                "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                "auto_param_L258_108",
                0.0,
            )
        ),
    },
}
chunk_summaries.append(chunk_summary)

if hasattr(chunk, "provenance") and chunk.provenance:
    provenance_with_data += 1
    if (
        not hasattr(chunk.provenance, "page_number")
        or chunk.provenance.page_number is None
    ):
        raise CPPAdapterError(
            f"Missing provenance.page_number in chunk {chunk.id}"
        )
    if (
        not hasattr(chunk.provenance, "section_header")
        or not chunk.provenance.section_header
    ):
        raise CPPAdapterError(
            f"Missing provenance.section_header in chunk {chunk.id}"
        )
else:
    raise CPPAdapterError(f"Missing provenance in chunk {chunk.id}")

# Advance offset by chunk length + 1 space separator
current_offset = chunk_end + 1

# Extract entities for entity_index
if hasattr(chunk, "entities") and chunk.entities:
    for entity in chunk.entities:
        entity_text = (
            entity.text if hasattr(entity, "text") else str(entity)
        )
        if entity_text not in entity_index:
            entity_index[entity_text] = []
```

```
                entity_index[entity_text].append(idx)

            # Extract temporal markers for temporal_index
            if hasattr(chunk, "time_facets") and chunk.time_facets:
                if hasattr(chunk.time_facets, "years") and chunk.time_facets.years:
                    for year in chunk.time_facets.years:
                        year_key = str(year)
                        if year_key not in temporal_index:
                            temporal_index[year_key] = []
                        temporal_index[year_key].append(idx)

            # Extract budget for tables
            if hasattr(chunk, "budget") and chunk.budget:
                budget = chunk.budget
                tables.append(
                    {
                        "table_id": f"budget_{idx}",
                          "label": f"Budget: {budget.source if hasattr(budget, 'source')
else 'Unknown'}",
                        "amount": getattr(budget, "amount", 0),
                        "currency": getattr(budget, "currency", "COP"),
                        "year": getattr(budget, "year", None),
                        "use": getattr(budget, "use", None),
                        "source": getattr(budget, "source", None),
                    }
                )

            # Create ChunkData object
            chunk_type_value = chunk.chunk_type
            if chunk_type_value not in [
                "diagnostic",
                "activity",
                "indicator",
                "resource",
                "temporal",
                "entity",
            ]:
                raise CPPAdapterError(
                    f"Invalid chunk_type '{chunk_type_value}' in chunk {chunk.id}"
                )

            chunks_data.append(
                ChunkData(
                    id=idx,
                    text=chunk_text,
                    chunk_type=chunk_type_value,
                    sentences=[idx],
                    tables=(
                        [len(tables) - 1]
                        if hasattr(chunk, "budget") and chunk.budget
                        else []
                    ),
                    start_pos=chunk_start,
                    end_pos=chunk_end,
```

```
            confidence=(
                getattr(chunk.confidence, "overall", 1.0)
                if hasattr(chunk, "confidence")
                else 1.0
            ),
            edges_out=[],  # Edges populated later if needed or from chunk_graph
            policy_area_id=extra_metadata["policy_area_id"],
            dimension_id=extra_metadata["dimension_id"],
            provenance=(
                Provenance(
                    page_number=chunk.provenance.page_number,
                    section_header=getattr(
                        chunk.provenance, "section_header", None
                    ),
                    bbox=getattr(chunk.provenance, "bbox", None),
                    span_in_page=getattr(
                        chunk.provenance, "span_in_page", None
                    ),
                    source_file=getattr(chunk.provenance, "source_file", None),
                )
                if hasattr(chunk, "provenance") and chunk.provenance
                else None
            ),
        )
    )

# Join full text
full_text = " ".join(full_text_parts)

if not full_text:
    raise CPPAdapterError("Generated full_text is empty")

# Build document indexes
indexes = {
    "term_index": {k: tuple(v) for k, v in term_index.items()},
    "numeric_index": {k: tuple(v) for k, v in numeric_index.items()},
    "temporal_index": {k: tuple(v) for k, v in temporal_index.items()},
    "entity_index": {k: tuple(v) for k, v in entity_index.items()},
}

# Build metadata from canon_package
metadata_dict = {
    "adapter_source": "CPPAdapter",
    "schema_version": (
        canon_package.schema_version
        if hasattr(canon_package, "schema_version")
        else "CPP-2025.1"
    ),
    "chunk_count": len(sorted_chunks),
    "processing_mode": "chunked",
    "chunks": chunk_summaries,
}

# Add quality metrics if available
```

```
    if hasattr(canon_package, "quality_metrics") and canon_package.quality_metrics:
        qm = canon_package.quality_metrics
        metadata_dict["quality_metrics"] = {
            "provenance_completeness": (
                qm.provenance_completeness
                if hasattr(qm, "provenance_completeness")
                else ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L328_117",
                    0.0,
                )
            ),
            "structural_consistency": (
                qm.structural_consistency
                if hasattr(qm, "structural_consistency")
                else ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L329_114",
                    0.0,
                )
            ),
            "boundary_f1": (
                qm.boundary_f1
                if hasattr(qm, "boundary_f1")
                else ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L330_81",
                    0.0,
                )
            ),
            "kpi_linkage_rate": (
                qm.kpi_linkage_rate
                if hasattr(qm, "kpi_linkage_rate")
                else ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L331_96",
                    0.0,
                )
            ),
            "budget_consistency_score": (
                qm.budget_consistency_score
                if hasattr(qm, "budget_consistency_score")
                else ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L332_120",
                    0.0,
                )
            ),
            "temporal_robustness": (
                qm.temporal_robustness
                if hasattr(qm, "temporal_robustness")
                else ParameterLoaderV2.get(
                    "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                    "auto_param_L333_105",
```

```
                0.0,
            )
        ),
        "chunk_context_coverage": (
            qm.chunk_context_coverage
            if hasattr(qm, "chunk_context_coverage")
            else ParameterLoaderV2.get(
                "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                "auto_param_L334_114",
                0.0,
            )
        ),
    }

# Add policy manifest if available
if hasattr(canon_package, "policy_manifest") and canon_package.policy_manifest:
    pm = canon_package.policy_manifest
    metadata_dict["policy_manifest"] = {
        "axes": pm.axes if hasattr(pm, "axes") else [],
        "programs": pm.programs if hasattr(pm, "programs") else [],
        "projects": pm.projects if hasattr(pm, "projects") else [],
        "years": pm.years if hasattr(pm, "years") else [],
        "territories": pm.territories if hasattr(pm, "territories") else [],
    }

# Add CPP rich data if available in metadata
if hasattr(canon_package, "metadata") and canon_package.metadata:
    if "spc_rich_data" in canon_package.metadata:
        metadata_dict["spc_rich_data"] = canon_package.metadata["spc_rich_data"]

if len(sorted_chunks) > 0:
    metadata_dict["provenance_completeness"] = provenance_with_data / len(
        sorted_chunks
    )

metadata = MappingProxyType(metadata_dict)

# Detect language (default to Spanish for Colombian policy documents)
language = "es"

# Create PreprocessedDocument (canonical orchestrator dataclass)
preprocessed_doc = PreprocessedDocument(
    document_id=document_id,
    raw_text=full_text,
    sentences=sentences,
    tables=tables,
    metadata=dict(metadata),
    sentence_metadata=sentence_metadata,
    indexes=indexes,
    structured_text={
        "full_text": full_text,
        "sections": (),
        "page_boundaries": (),
    },
```

```
            language=language,
            ingested_at=datetime.now(timezone.utc),
            full_text=full_text,
            chunks=chunks_data,
            chunk_index=chunk_index,
            chunk_graph={
                "chunks": {cid: chunk_index[cid] for cid in chunk_index},
                "edges": list(getattr(chunk_graph, "edges", [])),
            },
            processing_mode=processing_mode,
        )

        self.logger.info(
            f"Conversion complete: {len(sentences)} sentences, "
            f"{len(tables)} tables, {len(entity_index)} entities indexed"
        )

        # RUNTIME VALIDATION: Validate Adapter ? Orchestrator contract
        if self.wiring_validator is not None:
            self.logger.info("Validating Adapter ? Orchestrator contract (runtime)")
            try:
                # Convert PreprocessedDocument to dict for validation
                preprocessed_dict = {
                    "document_id": preprocessed_doc.document_id,
                    "sentence_metadata": preprocessed_doc.sentence_metadata,
                     "resolution_index": {},  # Placeholder, as it's not generated by the
adapter
                    "provenance_completeness": metadata_dict.get(
                        "provenance_completeness",
                        ParameterLoaderV2.get(
                            "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
                            "auto_param_L397_92",
                            0.0,
                        ),
                    ),
                }
                self.wiring_validator.validate_adapter_to_orchestrator(
                    preprocessed_dict
                )
                self.logger.info("? Adapter ? Orchestrator contract validation passed")
            except Exception as e:
                self.logger.error(
                    f"Adapter ? Orchestrator contract validation failed: {e}"
                )
                raise ValueError(
                        f"Runtime contract violation at Adapter ? Orchestrator boundary:
{e}"
                ) from e

        return preprocessed_doc


def adapt_cpp_to_orchestrator(
    canon_package: Any, document_id: str
```

```
) -> PreprocessedDocument:
    """
    Convenience function to adapt CPP to PreprocessedDocument.

    Args:
        canon_package: CanonPolicyPackage from ingestion
        document_id: Unique document identifier

    Returns:
        PreprocessedDocument for orchestrator

    Raises:
        CPPAdapterError: If conversion fails
    """
    adapter = CPPAdapter()
    return adapter.to_preprocessed_document(canon_package, document_id)


__all__ = [
    "CPPAdapter",
    "CPPAdapterError",
    "adapt_cpp_to_orchestrator",
]
```

```
src/farfan_pipeline/utils/metadata_loader.py

"""
Metadata Loader with Supply-Chain Security
Implements fail-fast validation with version pinning, checksum verification, and schema validation
"""

import hashlib
import json
import logging
from pathlib import Path
from collections.abc import Callable
from typing import Any, ParamSpec, TypeVar

import yaml

from farfan_pipeline.utils.paths import proj_root
from farfan_pipeline.core.parameters import ParameterLoaderV2

P = ParamSpec("P")
R = TypeVar("R")


def calibrated_method(_tag: str) -> Callable[[Callable[P, R]], Callable[P, R]]:
    """No-op legacy decorator (calibration system removed)."""

    def _decorator(fn: Callable[P, R]) -> Callable[P, R]:
        return fn

    return _decorator

try:
    import jsonschema
    JSONSCHEMA_AVAILABLE = True
except ImportError:
    JSONSCHEMA_AVAILABLE = False
    logging.warning("jsonschema not available - schema validation disabled")

logger = logging.getLogger(__name__)

class MetadataError(Exception):
    """Base exception for metadata errors"""
    pass

class MetadataVersionError(MetadataError):
    """Version mismatch error"""
    def __init__(self, expected: str, actual: str, file_path: str) -> None:
        self.expected = expected
        self.actual = actual
        self.file_path = file_path
        super().__init__(
            f"Version mismatch in {file_path}: expected {expected}, got {actual}"
        )
```

```
class MetadataIntegrityError(MetadataError):
    """Checksum/integrity violation error"""
        def __init__(self, file_path: str, expected_checksum: str | None = None,
actual_checksum: str | None = None) -> None:
        self.file_path = file_path
        self.expected_checksum = expected_checksum
        self.actual_checksum = actual_checksum
        msg = f"Integrity violation in {file_path}"
        if expected_checksum and actual_checksum:
            msg += f": expected checksum {expected_checksum}, got {actual_checksum}"
        super().__init__(msg)

class MetadataSchemaError(MetadataError):
    """Schema validation error"""
    def __init__(self, file_path: str, validation_errors: list) -> None:
        self.file_path = file_path
        self.validation_errors = validation_errors
        error_msgs = '\n'.join(f"  - {err}" for err in validation_errors)
        super().__init__(
            f"Schema validation failed for {file_path}:\n{error_msgs}"
        )

class MetadataMissingKeyError(MetadataError):
    """Required key missing in metadata"""
    def __init__(self, file_path: str, missing_key: str, context: str = "") -> None:
        self.file_path = file_path
        self.missing_key = missing_key
        self.context = context
        msg = f"Required key '{missing_key}' missing in {file_path}"
        if context:
            msg += f" ({context})"
        super().__init__(msg)

class MetadataLoader:
    """
    Unified metadata loader with strict validation

    Features:
    - Version pinning with semantic versioning
    - SHA-256 checksum verification
    - JSON Schema validation
    - Fail-fast on any violation
    - Structured logging of all errors
    """

    def __init__(self, workspace_root: Path | None = None) -> None:
        self.workspace_root = Path(workspace_root) if workspace_root else proj_root()
        self.schemas_dir = self.workspace_root / "schemas"

        # Loaded schemas cache
        self._schema_cache: dict[str, dict] = {}

    def load_and_validate_metadata(
```

```
        self,
        path: Path,
        schema_ref: str | None = None,
        required_version: str | None = None,
        expected_checksum: str | None = None,
        checksum_algorithm: str = "sha256"
    ) -> dict[str, Any]:
        """
        Load and validate metadata file with all safeguards

        Args:
            path: Path to metadata file (JSON or YAML)
            schema_ref: Schema file name (e.g., "rubric.schema.json")
                                    required_version:   Required   version   string   (e.g.,
"2.ParameterLoaderV2.get("farfan_core.utils.metadata_loader.MetadataLoader.__init__",
"auto_param_L105_64", 0.0)")
            expected_checksum: Expected SHA-256 checksum (hex)
            checksum_algorithm: Hash algorithm ("sha256", "md5")

        Returns:
            Validated metadata dictionary

        Raises:
            MetadataVersionError: Version mismatch
            MetadataIntegrityError: Checksum mismatch
            MetadataSchemaError: Schema validation failure
            MetadataMissingKeyError: Required key missing
        """

        # 1. Load file
        metadata = self._load_file(path)

        # 2. Version check
        if required_version:
            actual_version = metadata.get("version")
            if not actual_version:
                      raise  MetadataMissingKeyError(str(path),  "version",  "version  field
required")

            if actual_version != required_version:
                self._log_error(
                    rule_id="VERSION_MISMATCH",
                    file_path=str(path),
                    expected=required_version,
                    actual=actual_version
                )
                raise MetadataVersionError(required_version, actual_version, str(path))

            logger.info(f"? Version validated: {path.name} v{actual_version}")

        # 3. Checksum verification
        if expected_checksum:
            actual_checksum = self._calculate_checksum(metadata, checksum_algorithm)
```

```
            if actual_checksum != expected_checksum:
                self._log_error(
                    rule_id="CHECKSUM_MISMATCH",
                    file_path=str(path),
                    expected=expected_checksum,
                    actual=actual_checksum
                )
                                raise  MetadataIntegrityError(str(path),  expected_checksum,
actual_checksum)

            logger.info(f"? Checksum validated: {path.name} ({checksum_algorithm})")

        # 4. Schema validation
        if schema_ref and JSONSCHEMA_AVAILABLE:
            schema = self._load_schema(schema_ref)
            errors = self._validate_schema(metadata, schema)

            if errors:
                self._log_error(
                    rule_id="SCHEMA_VALIDATION_FAILED",
                    file_path=str(path),
                    errors=errors
                )
                raise MetadataSchemaError(str(path), errors)

            logger.info(f"? Schema validated: {path.name}")

        return metadata

    @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._load_file")
    def _load_file(self, path: Path) -> dict[str, Any]:
        """Load JSON or YAML file"""
        if not path.exists():
            raise FileNotFoundError(f"Metadata file not found: {path}")

        try:
            with open(path, encoding='utf-8') as f:
                content = f.read()

            if path.suffix in ['.json']:
                return json.loads(content)
            elif path.suffix in ['.yaml', '.yml']:
                return yaml.safe_load(content)
            else:
                raise ValueError(f"Unsupported file type: {path.suffix}")

        except (json.JSONDecodeError, yaml.YAMLError) as e:
            raise MetadataError(f"Failed to parse {path}: {e}")


@calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._calculate_checksum
")
     def _calculate_checksum(self, metadata: dict[str, Any], algorithm: str = "sha256")
-> str:
```

```
        """
        Calculate reproducible checksum of metadata

        Normalization:
        - JSON serialization with sorted keys
        - UTF-8 encoding
        - No whitespace variations
        """
        normalized = json.dumps(metadata, sort_keys=True, separators=(',', ':'))

        if algorithm == "sha256":
            return hashlib.sha256(normalized.encode('utf-8')).hexdigest()
        elif algorithm == "md5":
            return hashlib.md5(normalized.encode('utf-8')).hexdigest()
        else:
            raise ValueError(f"Unsupported algorithm: {algorithm}")

    @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._load_schema")
    def _load_schema(self, schema_ref: str) -> dict[str, Any]:
        """Load JSON Schema from schemas directory"""
        if schema_ref in self._schema_cache:
            return self._schema_cache[schema_ref]

        schema_path = self.schemas_dir / schema_ref

        if not schema_path.exists():
            raise FileNotFoundError(f"Schema not found: {schema_path}")

        with open(schema_path, encoding='utf-8') as f:
            schema = json.load(f)

        self._schema_cache[schema_ref] = schema
        return schema


@calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._validate_schema")
      def _validate_schema(self, metadata: dict[str, Any], schema: dict[str, Any]) ->
list:
        """Validate metadata against JSON Schema"""
        if not JSONSCHEMA_AVAILABLE:
            logger.warning("jsonschema not available - skipping schema validation")
            return []

        # Import check for type checker
        import jsonschema as js
        validator = js.Draft7Validator(schema)
        errors = []

        for error in validator.iter_errors(metadata):
            error_path = '.'.join(str(p) for p in error.path) if error.path else 'root'
            errors.append(f"{error_path}: {error.message}")

        return errors
```

```
    @calibrated_method("farfan_core.utils.metadata_loader.MetadataLoader._log_error")
    def _log_error(self, rule_id: str, file_path: str, **kwargs) -> None:
        """Structured error logging"""
        from datetime import datetime, timezone

        log_entry = {
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "level": "ERROR",
            "rule_id": rule_id,
            "file_path": file_path,
            **kwargs
        }

        logger.error(json.dumps(log_entry, indent=2))

# REMOVED: load_cuestionario() - LEGACY FUNCTION
# Questionnaire monolith must ONLY be loaded via factory.load_questionnaire_monolith()
# This enforces architectural requirement: Single I/O boundary in factory.py
# See: src/farfan_core/core/orchestrator/factory.py::load_questionnaire_monolith()

def load_execution_mapping(
    path: Path | None = None,
    required_version: str = "2.0"
) -> dict[str, Any]:
    """
    Load and validate execution_mapping.yaml

    Args:
        path: Path to execution mapping (default: execution_mapping.yaml)
        required_version: Required version

    Returns:
        Validated execution mapping
    """
    if path is None:
        path = proj_root() / "execution_mapping.yaml"

    loader = MetadataLoader()
    return loader.load_and_validate_metadata(
        path=path,
        schema_ref="execution_mapping.schema.json",
        required_version=required_version
    )

def load_rubric_scoring(
    path: Path | None = None,
    required_version: str = "2.0"
) -> dict[str, Any]:
    """
    Load and validate rubric_scoring.json

    Args:
        path: Path to rubric scoring (default: rubric_scoring.json)
        required_version: Required version
```

```
Returns:
    Validated rubric scoring configuration
"""
if path is None:
    path = proj_root() / "rubric_scoring.json"

loader = MetadataLoader()
return loader.load_and_validate_metadata(
    path=path,
    schema_ref="rubric.schema.json",
    required_version=required_version
)
```

```
src/farfan_pipeline/utils/method_config_loader.py

"""
Method Configuration Loader for Canonical JSON Specification.

Provides unified access to method parameters from the canonical
parameterization specification.
"""
import ast
import json
from pathlib import Path
from collections.abc import Callable
from typing import Any, ParamSpec, TypeVar
from farfan_pipeline.core.parameters import ParameterLoaderV2

P = ParamSpec("P")
R = TypeVar("R")


def calibrated_method(_tag: str) -> Callable[[Callable[P, R]], Callable[P, R]]:
    """No-op legacy decorator (calibration system removed)."""

    def _decorator(fn: Callable[P, R]) -> Callable[P, R]:
        return fn

    return _decorator


class MethodConfigLoader:
    """
    Loads and provides access to method parameters from canonical JSON.

    Usage:
        loader = MethodConfigLoader("CANONICAL_METHOD_PARAMETERIZATION_SPEC.json")
        threshold = loader.get_method_parameter(
            "CAUSAL.BMI.infer_mech_v1",
            "kl_divergence_threshold"
        )

    Note:
        The loader expects the JSON spec to follow the canonical schema with
        keys: specification_metadata, methods, and epistemic_validation_summary.
    """

    def __init__(self, spec_path: str | Path) -> None:
        self.spec_path = Path(spec_path)
        with open(self.spec_path) as f:
            self.spec = json.load(f)

        # Validate schema before use
        self.validate_spec_schema()

        # Build index for fast lookup
        self._method_index = {
```

```
        method["canonical_id"]: method
        for method in self.spec["methods"]
    }

@calibrated_method("legacy.noop.method_config_loader.validate_spec_schema")
def validate_spec_schema(self) -> None:
    """
    Validate JSON spec matches expected schema.

    Raises:
        ValueError: If spec is missing required keys
    """
    required_keys = {"specification_metadata", "methods"}
    # Note: epistemic_validation_summary is optional in some versions
    if not required_keys.issubset(self.spec.keys()):
        missing = required_keys - set(self.spec.keys())
        raise ValueError(f"Spec missing required keys: {missing}")

def get_method_parameter(
    self,
    canonical_id: str,
    param_name: str,
    override: Any = None
) -> Any:
    """
    Get parameter value for a method.

    Args:
        canonical_id: Canonical method ID (e.g., "CAUSAL.BMI.infer_mech_v1")
        param_name: Parameter name
        override: Optional override value (takes precedence over default)

    Returns:
        Parameter value (default or override)

    Raises:
        KeyError: If method or parameter not found
    """
    if canonical_id not in self._method_index:
        raise KeyError(f"Method {canonical_id} not found in canonical spec")

    method = self._method_index[canonical_id]

    for param in method["parameters"]:
        if param["name"] == param_name:
            return override if override is not None else param["default"]

    raise KeyError(f"Parameter {param_name} not found for method {canonical_id}")

@calibrated_method("legacy.noop.method_config_loader.get_method_description")
def get_method_description(self, canonical_id: str) -> str:
    """Get method description."""
    return self._method_index[canonical_id]["description"]
```

```
    @calibrated_method("legacy.noop.method_config_loader.get_parameter_spec")
    def get_parameter_spec(self, canonical_id: str, param_name: str) -> dict:
        """Get full parameter specification including allowed values."""
        method = self._method_index[canonical_id]
        for param in method["parameters"]:
            if param["name"] == param_name:
                return param
        raise KeyError(f"Parameter {param_name} not found")

    def validate_parameter_value(
        self,
        canonical_id: str,
        param_name: str,
        value: Any
    ) -> bool:
        """
        Validate parameter value against allowed_values specification.

        Returns:
            True if valid, raises ValueError if invalid
        """
        param_spec = self.get_parameter_spec(canonical_id, param_name)
        allowed = param_spec["allowed_values"]

        if allowed["kind"] == "range":
            spec = allowed["spec"]
            min_val, max_val = self._parse_range(spec)
            if not (min_val <= value <= max_val):
                raise ValueError(f"{param_name}={value} out of range {spec}")

        elif allowed["kind"] == "set":
            spec = allowed["spec"]
            valid_values = self._parse_set(spec)
            if value not in valid_values:
                raise ValueError(f"{param_name}={value} not in allowed set {spec}")

        return True


@calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_ran
ge")
    def _parse_range(self, spec: str) -> tuple[float, float]:
        """
                                            Parse      range      specification      like
'[ParameterLoaderV2.get("farfan_core.utils.method_config_loader.MethodConfigLoader._pars
e_range",                              "auto_param_L135_41",                              0.0),
ParameterLoaderV2.get("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_
range", "auto_param_L135_46", 1.0)], inclusive' or '[100, 10000], integer'.

        Args:
            spec: Range specification string with format "[min, max], modifiers"
                  Modifiers can include: inclusive, exclusive, integer

        Returns:
```

```
            Tuple of (min_val, max_val) as floats

        Raises:
            ValueError: If spec format is invalid

        Note:
            The inclusive/exclusive and integer modifiers are parsed but not
            currently enforced in validation. This maintains compatibility with
            the current spec while allowing future enhancement.
        """
        try:
            # Extract bracketed part before any modifiers
            bracket_part = spec.split("]")[0] + "]"
            parts = bracket_part.replace("[", "").replace("]", "").split(",")
            min_val = float(parts[0].strip())
            max_val = float(parts[1].strip())
            return min_val, max_val
        except (IndexError, ValueError) as e:
            raise ValueError(f"Invalid range spec: {spec}") from e


@calibrated_method("farfan_core.utils.method_config_loader.MethodConfigLoader._parse_set
")
    def _parse_set(self, spec: str | list) -> set:
        """
        Parse set specification safely.

        Args:
            spec: Either a list or a string representation of a Python literal

        Returns:
            Set of allowed values

        Raises:
            ValueError: If spec cannot be parsed safely

        Note:
            Uses ast.literal_eval() for safe parsing of string specs.
            Only Python literals (strings, numbers, tuples, lists, dicts,
            booleans, None) are supported - no arbitrary code execution.
        """
        if isinstance(spec, list):
            return set(spec)
        try:
            # Use ast.literal_eval for safer parsing
            return set(ast.literal_eval(spec))
        except (ValueError, SyntaxError) as e:
            raise ValueError(f"Invalid set spec: {spec}") from e
```

src/farfan_pipeline/utils/paths.py

```
"""Path helpers (single source of truth).

Re-exported from `farfan_pipeline.phases.Phase_zero.paths`.
"""

from __future__ import annotations

from farfan_pipeline.phases.Phase_zero.paths import (  # noqa: F401
    PROJECT_ROOT,
    SRC_DIR,
    DATA_DIR,
    TESTS_DIR,
    CONFIG_DIR,
    QUESTIONNAIRE_FILE,
    PathError,
    PathTraversalError,
    PathNotFoundError,
    PathOutsideWorkspaceError,
    UnnormalizedPathError,
    proj_root,
    src_dir,
    data_dir,
    tmp_dir,
    build_dir,
    cache_dir,
    reports_dir,
    is_within,
    safe_join,
    normalize_unicode,
    normalize_case,
    resources,
    validate_read_path,
    validate_write_path,
    get_env_path,
    get_workdir,
    get_tmpdir,
    get_reports_dir,
)
```

src/farfan_pipeline/utils/validation/schema_validator.py

```
"""
Schema validation for monolith initialization.

This module implements the Monolith Initialization Validator (MIV) that scans
and verifies the integrity of the global schema before runtime execution.
"""

import hashlib
import json
from datetime import datetime, timezone
from pathlib import Path
from collections.abc import Callable
from typing import Any, ParamSpec, TypeVar

import jsonschema
from pydantic import BaseModel, ConfigDict, Field

P = ParamSpec("P")
R = TypeVar("R")


def calibrated_method(_tag: str) -> Callable[[Callable[P, R]], Callable[P, R]]:
    """No-op legacy decorator (calibration system removed)."""

    def _decorator(fn: Callable[P, R]) -> Callable[P, R]:
        return fn

    return _decorator


class SchemaInitializationError(Exception):
    """Raised when schema initialization validation fails."""
    pass

class MonolithIntegrityReport(BaseModel):
    """Report of monolith integrity validation."""

    model_config = ConfigDict(extra='allow')

                            timestamp:      str      =      Field(default_factory=lambda:
datetime.now(timezone.utc).isoformat())
    schema_version: str
    validation_passed: bool
    errors: list[str] = Field(default_factory=list)
    warnings: list[str] = Field(default_factory=list)
    schema_hash: str
    question_counts: dict[str, int]
    referential_integrity: dict[str, bool]

class MonolithSchemaValidator:
    """
    Monolith Initialization Validator (MIV).
```

```
    Bootstrapping process that scans and verifies the integrity of the
    global schema before runtime execution.
    """

    EXPECTED_SCHEMA_VERSION = "2.0.0"
    EXPECTED_MICRO_QUESTIONS = 300
    EXPECTED_MESO_QUESTIONS = 4
    EXPECTED_MACRO_QUESTIONS = 1
    EXPECTED_POLICY_AREAS = 10
    EXPECTED_DIMENSIONS = 6
    EXPECTED_CLUSTERS = 4

    def __init__(self, schema_path: str | None = None) -> None:
        """
        Initialize validator.

        Args:
            schema_path: Path to JSON schema file (optional)
        """
        self.schema_path = schema_path
        self.schema: dict[str, Any] | None = None
        self.errors: list[str] = []
        self.warnings: list[str] = []

        if schema_path:
            self._load_schema()

    @calibrated_method("legacy.noop.schema_validator._load_schema")
    def _load_schema(self, **kwargs: Any) -> None:
        """
        Load JSON schema from file.

        Handles file existence checks and JSON decoding errors with specific warnings.
        """
        if not self.schema_path:
            return

        schema_file = Path(self.schema_path)
        if not schema_file.exists():
            self.warnings.append(f"Schema file not found: {self.schema_path}")
            return

        try:
            with open(schema_file, mode='r', encoding='utf-8') as f:
                self.schema = json.load(f)
        except json.JSONDecodeError as e:
                 self.warnings.append(f"Invalid JSON in schema file {self.schema_path}:
{e}")
        except Exception as e:
              self.warnings.append(f"Unexpected error loading schema {self.schema_path}:
{e}")

    def validate_monolith(
```

```
    self,
    monolith: dict[str, Any],
    strict: bool = True
) -> MonolithIntegrityReport:
    """
    Validate monolith structure and integrity.

    Args:
        monolith: Monolith configuration dictionary
        strict: If True, raises exception on validation failure

    Returns:
        MonolithIntegrityReport with validation results

    Raises:
        SchemaInitializationError: If validation fails and strict=True
    """
    self.errors = []
    self.warnings = []

    # 1. Validate structure
    self._validate_structure(monolith).value

    # 2. Validate schema version
    schema_version = self._validate_schema_version(monolith).value

    # 3. Validate question counts
    question_counts = self._validate_question_counts(monolith).value

    # 4. Validate referential integrity
    referential_integrity = self._validate_referential_integrity(monolith)

    # 5. Validate against JSON schema if available
    if self.schema:
        self._validate_against_schema(monolith).value

    # 6. Validate field coverage
    field_coverage = self._validate_field_coverage(monolith).value

    # 7. Validate semantic consistency
    semantic_consistency = self._validate_semantic_consistency(monolith).value

    # 8. Calculate schema hash
    schema_hash = self._calculate_schema_hash(monolith).value

    # Build report
    validation_passed = len(self.errors) == 0

    report = MonolithIntegrityReport(
        schema_version=schema_version,
        validation_passed=validation_passed,
        errors=self.errors,
        warnings=self.warnings,
        schema_hash=schema_hash,
```

```
            question_counts=question_counts,
            referential_integrity=referential_integrity
        )

        # Raise error if strict mode and validation failed
        if strict and not validation_passed:
            error_msg = "Schema initialization failed:\n" + "\n".join(
                f"  - {e}" for e in self.errors
            )
            raise SchemaInitializationError(error_msg)

        return report


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._validate_structure")
    def _validate_structure(self, monolith: dict[str, Any], **kwargs: Any) -> None:
        """Validate top-level structure."""
        required_keys = ['schema_version', 'version', 'blocks', 'integrity']

        for key in required_keys:
            if key not in monolith:
                self.errors.append(f"Missing required top-level key: {key}")

        if 'blocks' in monolith:
            blocks = monolith['blocks']
            required_blocks = [
                'niveles_abstraccion',
                'micro_questions',
                'meso_questions',
                'macro_question',
                'scoring'
            ]

            for block in required_blocks:
                if block not in blocks:
                    self.errors.append(f"Missing required block: {block}")


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._validate_schema_version")
    def _validate_schema_version(self, monolith: dict[str, Any], **kwargs: Any) -> str:
        """Validate schema version."""
        schema_version = monolith.get('schema_version', '')

        if not schema_version:
            self.errors.append("Missing schema_version")
            return ''

        # Allow any version but warn if not expected
        if schema_version != self.EXPECTED_SCHEMA_VERSION:
            self.warnings.append(
                f"Schema version {schema_version} differs from expected "
                f"{self.EXPECTED_SCHEMA_VERSION}"
```

```
            )

        return schema_version


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._validate_question_counts")
      def _validate_question_counts(self, monolith: dict[str, Any], **kwargs: Any) ->
dict[str, int]:
        """Validate question counts."""
        blocks = monolith.get('blocks', {})

        micro_count = len(blocks.get('micro_questions', []))
        meso_count = len(blocks.get('meso_questions', []))
        macro_exists = 1 if blocks.get('macro_question') else 0
        total_count = micro_count + meso_count + macro_exists

        # Validate counts
        if micro_count != self.EXPECTED_MICRO_QUESTIONS:
            self.errors.append(
                f"Expected {self.EXPECTED_MICRO_QUESTIONS} micro questions, "
                f"got {micro_count}"
            )

        if meso_count != self.EXPECTED_MESO_QUESTIONS:
            self.errors.append(
                f"Expected {self.EXPECTED_MESO_QUESTIONS} meso questions, "
                f"got {meso_count}"
            )

        if not macro_exists:
            self.errors.append("Missing macro question")

        expected_total = (
            self.EXPECTED_MICRO_QUESTIONS +
            self.EXPECTED_MESO_QUESTIONS +
            self.EXPECTED_MACRO_QUESTIONS
        )

        if total_count != expected_total:
            self.errors.append(
                f"Expected {expected_total} total questions, got {total_count}"
            )

        return {
            'micro': micro_count,
            'meso': meso_count,
            'macro': macro_exists,
            'total': total_count
        }

    def _validate_referential_integrity(
        self,
        monolith: dict[str, Any]
```

```
) -> dict[str, bool]:
    """
    Validate referential integrity.

    Ensures no dangling foreign keys or invalid cross-references.
    """
    results = {
        'policy_areas': True,
        'dimensions': True,
        'clusters': True,
        'micro_questions': True
    }

    blocks = monolith.get('blocks', {})
    niveles = blocks.get('niveles_abstraccion', {})

    # Get all valid IDs
    valid_policy_areas = {
        pa['policy_area_id']
        for pa in niveles.get('policy_areas', [])
    }

    valid_dimensions = {
        dim['dimension_id']
        for dim in niveles.get('dimensions', [])
    }

    valid_clusters = {
        cl['cluster_id']
        for cl in niveles.get('clusters', [])
    }

    # Validate cluster references to policy areas
    for cluster in niveles.get('clusters', []):
        cluster_id = cluster.get('cluster_id', 'UNKNOWN')
        for pa_id in cluster.get('policy_area_ids', []):
            if pa_id not in valid_policy_areas:
                self.errors.append(
                    f"Cluster {cluster_id} references invalid policy area: {pa_id}"
                )
                results['clusters'] = False

    # Validate micro questions reference valid areas/dimensions
    for question in blocks.get('micro_questions', []):
        q_id = question.get('question_id', 'UNKNOWN')
        pa_id = question.get('policy_area_id')
        dim_id = question.get('dimension_id')

        if pa_id and pa_id not in valid_policy_areas:
            self.errors.append(
                f"Question {q_id} references invalid policy area: {pa_id}"
            )
            results['micro_questions'] = False
```

```
            if dim_id and dim_id not in valid_dimensions:
                self.errors.append(
                    f"Question {q_id} references invalid dimension: {dim_id}"
                )
                results['micro_questions'] = False

        # Validate meso questions reference valid clusters
        for question in blocks.get('meso_questions', []):
            q_id = question.get('question_id', 'UNKNOWN')
            cl_id = question.get('cluster_id')

            if cl_id and cl_id not in valid_clusters:
                self.errors.append(
                    f"Meso question {q_id} references invalid cluster: {cl_id}"
                )

        return results


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._validate_against_schema")
    def _validate_against_schema(self, monolith: dict[str, Any], **kwargs: Any) -> None:
        """Validate monolith against JSON schema."""
        if not self.schema:
            return

        try:
            jsonschema.validate(instance=monolith, schema=self.schema)
        except jsonschema.ValidationError as e:
            self.errors.append(f"Schema validation error: {e.message}")
        except Exception as e:
            self.warnings.append(f"Schema validation failed: {e}")


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._validate_field_coverage")
      def _validate_field_coverage(self, monolith: dict[str, Any], **kwargs: Any) ->
dict[str, float]:
        """
        Validate field coverage for micro-questions.

        Enforces >= 99% coverage for critical fields.
        """
        blocks = monolith.get('blocks', {})
        micro_questions = blocks.get('micro_questions', [])
        total_micro = len(micro_questions)

        if total_micro == 0:
            return {}

        critical_fields = [
            "question_id",
            "question_global",
            "base_slot",
```

```
            "dimension_id",
            "policy_area_id",
            "cluster_id",
            "scoring_modality",
            "scoring_definition_ref",
            "expected_elements",
            "method_sets",
            "failure_contract",
            "validations"
        ]

        coverage_stats = {}

        for field in critical_fields:
            present_count = 0
            for q in micro_questions:
                val = q.get(field)
                # Check for presence and non-emptiness (for lists/strings)
                if val is not None:
                    if isinstance(val, (list, dict, str)) and len(val) == 0:
                            pass # Empty container/string counts as missing for critical
fields
                    else:
                        present_count += 1

            coverage = present_count / total_micro
            coverage_stats[field] = coverage

            if coverage < 0.99:
                self.errors.append(
                      f"Field coverage violation: '{field}' has {coverage:.2%} coverage,
required >= 99%"
                )
            elif coverage < 1.0:
                self.warnings.append(
                        f"Field coverage warning: '{field}' has {coverage:.2%} coverage
(should be 100%)"
                )

        return coverage_stats


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._validate_semantic_consistency")
    def _validate_semantic_consistency(self, monolith: dict[str, Any], **kwargs: Any) ->
bool:
        """
        Validate semantic consistency of micro-questions.

        Checks:
        - question_global uniqueness and range (1-300)
        - base_slot format (D[1-6]-Q[1-5])
        - dimension_id format (DIM0[1-6])
        - policy_area_id format (PA0[1-9]|PA10)
```

```
        - cluster_id format (CL0[1-4])
        """
        blocks = monolith.get('blocks', {})
        micro_questions = blocks.get('micro_questions', [])

        seen_globals = set()
        all_valid = True

        import re
        base_slot_pattern = re.compile(r"^D[1-6]-Q[1-5]$")
        dim_pattern = re.compile(r"^DIM0[1-6]$")
        pa_pattern = re.compile(r"^PA(0[1-9]|10)$")
        cluster_pattern = re.compile(r"^CL0[1-4]$")

        for q in micro_questions:
            q_id = q.get("question_id", "UNKNOWN")
            q_global = q.get("question_global")

            # Check question_global
            if not isinstance(q_global, int) or not (1 <= q_global <= 300):
                 self.errors.append(f"Question {q_id}: Invalid question_global {q_global}
(must be 1-300)")
                all_valid = False
            elif q_global in seen_globals:
                        self.errors.append(f"Question {q_id}: Duplicate question_global
{q_global}")
                all_valid = False
            else:
                seen_globals.add(q_global)

            # Check patterns
            base_slot = q.get("base_slot")
            if not base_slot or not base_slot_pattern.match(base_slot):
                self.errors.append(f"Question {q_id}: Invalid base_slot '{base_slot}'")
                all_valid = False

            dim_id = q.get("dimension_id")
            if not dim_id or not dim_pattern.match(dim_id):
                self.errors.append(f"Question {q_id}: Invalid dimension_id '{dim_id}'")
                all_valid = False

            pa_id = q.get("policy_area_id")
            if not pa_id or not pa_pattern.match(pa_id):
                self.errors.append(f"Question {q_id}: Invalid policy_area_id '{pa_id}'")
                all_valid = False

            cluster_id = q.get("cluster_id")
            if not cluster_id or not cluster_pattern.match(cluster_id):
                            self.errors.append(f"Question {q_id}: Invalid cluster_id
'{cluster_id}'")
                all_valid = False

        # Check for gaps in question_global
        if len(seen_globals) == 300:
```

```
            expected_set = set(range(1, 301))
            if seen_globals != expected_set:
                missing = expected_set - seen_globals
                self.errors.append(f"Missing question_global values: {missing}")
                all_valid = False

        return all_valid


@calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidato
r._calculate_schema_hash")
    def _calculate_schema_hash(self, monolith: dict[str, Any], **kwargs: Any) -> str:
        """Calculate deterministic hash of monolith schema."""
        # Create canonical JSON representation
        canonical = json.dumps(monolith, sort_keys=True, ensure_ascii=True)

        # Calculate SHA-256 hash
        hash_obj = hashlib.sha256(canonical.encode('utf-8'))
        return hash_obj.hexdigest()

    def generate_validation_report(
        self,
        report: MonolithIntegrityReport,
        output_path: str
    ) -> None:
        """
        Generate and save validation report artifact.

        Args:
            report: Validation report
            output_path: Path to save report JSON
        """
        output_file = Path(output_path)
        output_file.parent.mkdir(parents=True, exist_ok=True)

        with open(output_file, 'w', encoding='utf-8') as f:
            json.dump(
                report.model_dump(),
                f,
                indent=2,
                ensure_ascii=False
            )

def validate_monolith_schema(
    monolith: dict[str, Any],
    schema_path: str | None = None,
    strict: bool = True
) -> MonolithIntegrityReport:
    """
    Convenience function to validate monolith schema.

    Args:
        monolith: Monolith configuration
        schema_path: Optional path to JSON schema
```

```
    strict: If True, raises exception on failure

Returns:
    MonolithIntegrityReport

Raises:
    SchemaInitializationError: If validation fails and strict=True
"""
validator = MonolithSchemaValidator(schema_path=schema_path)
return validator.validate_monolith(monolith, strict=strict)
```

src/methods_dispensary/__init__.py

```
"""Compatibility shim for legacy imports.

The methods dispensary implementation moved to `farfan_pipeline.methods`.
This module preserves historical imports like:

- `from methods_dispensary.derek_beach import DerekBeachProducer`

New code should prefer:
- `from farfan_pipeline.methods.derek_beach import DerekBeachProducer`
"""

from __future__ import annotations

from pathlib import Path


# Redirect package submodule resolution to the new location.
__path__ = [
               str((Path(__file__).resolve().parent.parent    /    "farfan_pipeline"    /
"methods").resolve())
]

# Preserve the legacy convenience exports by delegating.
from farfan_pipeline.methods import *  # noqa: F403,E402
```

src/ontology/canonical_value_chain.py

```
"""
Canonical value chain specification and validator.
"""
from __future__ import annotations

import hashlib
import json
from dataclasses import dataclass
from pathlib import Path
from typing import Any, Dict, List, Tuple


VALUE_CHAIN_DIMENSIONS: Dict[str, Dict[str, Any]] = {
    "DIM01": {
        "code": "DIM01",
        "name": "INSUMOS",
        "label": "Diagnóstico y Recursos",
        "base_slots": ["D1-Q1", "D1-Q2", "D1-Q3", "D1-Q4", "D1-Q5"],
        "analytical_variables": {
            "linea_base_diagnostico": {
                "slot": "D1-Q1",
                "expected_elements": [
                    "cobertura_territorial_especificada",
                    "fuentes_oficiales",
                    "indicadores_cuantitativos",
                    "series_temporales_años",
                ],
            },
            "dimensionamiento_brecha": {
                "slot": "D1-Q2",
                "expected_elements": [
                    "brecha_cuantificada",
                    "limitaciones_datos",
                    "subregistro",
                ],
            },
            "asignacion_recursos": {
                "slot": "D1-Q3",
                "expected_elements": [
                    "asignacion_explicita",
                    "suficiencia_justificada",
                    "trazabilidad_ppi_bpin",
                ],
            },
            "capacidad_institucional": {
                "slot": "D1-Q4",
                "expected_elements": [
                    "cuellos_botella",
                    "datos_sistemas",
                    "gobernanza",
                    "procesos",
                    "talento_humano",
```

```
            ],
        },
        "marco_restricciones": {
            "slot": "D1-Q5",
            "expected_elements": [
                "coherencia_demostrada",
                "restricciones_legales",
                "restricciones_presupuestales",
                "restricciones_temporales",
            ],
        },
    },
    "keywords_general": [
        "línea base",
        "año base",
        "situación inicial",
        "diagnóstico",
        "DANE",
        "Medicina Legal",
        "Fiscalía",
        "Policía Nacional",
        "SIVIGILA",
        "SISPRO",
        "brecha",
        "déficit",
        "rezago",
        "subregistro",
        "cifra negra",
        "recursos",
        "presupuesto",
        "PPI",
        "BPIN",
        "asignación",
        "millones",
        "capacidad instalada",
        "talento humano",
        "personal idóneo",
        "cuello de botella",
        "limitación institucional",
        "barrera",
        "marco legal",
        "Ley",
        "Decreto",
        "competencias",
        "restricción",
    ],
},
"DIM02": {
    "code": "DIM02",
    "name": "ACTIVIDADES",
    "label": "Diseño de Intervención",
    "base_slots": ["D2-Q1", "D2-Q2", "D2-Q3", "D2-Q4", "D2-Q5"],
    "analytical_variables": {
        "estructura_operativa": {
```

```
            "slot": "D2-Q1",
            "expected_elements": [
                "columna_costo",
                "columna_cronograma",
                "columna_producto",
                "columna_responsable",
                "formato_tabular",
            ],
        },
        "diseño_intervencion": {
            "slot": "D2-Q2",
            "expected_elements": [
                "instrumento_especificado",
                "logica_causal_explicita",
                "poblacion_objetivo_definida",
            ],
        },
        "pertinencia_causal": {
            "slot": "D2-Q3",
            "expected_elements": [
                "aborda_causa_raiz",
                "vinculo_diagnostico_actividad",
            ],
        },
        "gestion_riesgos": {
            "slot": "D2-Q4",
            "expected_elements": [
                "mitigacion_propuesta",
                "riesgos_identificados",
            ],
        },
        "articulacion_actividades": {
            "slot": "D2-Q5",
            "expected_elements": [
                "complementariedad_explicita",
                "secuenciacion_logica",
            ],
        },
    },
    "keywords_general": [
        "matriz operativa",
        "plan de acción",
        "cronograma",
        "responsable",
        "actividad",
        "intervención",
        "programa",
        "proyecto",
        "estrategia",
        "población objetivo",
        "beneficiarios",
        "focalización",
        "causa raíz",
        "árbol de problemas",
```

```
        "teoría de cambio",
        "riesgo",
        "mitigación",
        "contingencia",
        "articulación",
        "complementariedad",
        "secuencia",
        "etapa",
        "fase",
    ],
},
"DIM03": {
    "code": "DIM03",
    "name": "PRODUCTOS",
    "label": "Productos y Outputs",
    "base_slots": ["D3-Q1", "D3-Q2", "D3-Q3", "D3-Q4", "D3-Q5"],
    "analytical_variables": {
        "indicadores_producto": {
            "slot": "D3-Q1",
            "expected_elements": [
                "fuente_verificacion",
                "linea_base_producto",
                "meta_cuantitativa",
            ],
        },
        "dosificacion_metas": {
            "slot": "D3-Q2",
            "expected_elements": [
                "dosificacion_definida",
                "proporcionalidad_meta_brecha",
            ],
        },
        "trazabilidad": {
            "slot": "D3-Q3",
            "expected_elements": [
                "trazabilidad_organizacional",
                "trazabilidad_presupuestal",
            ],
        },
        "factibilidad": {
            "slot": "D3-Q4",
            "expected_elements": [
                "coherencia_recursos",
                "factibilidad_tecnica",
                "realismo_plazos",
            ],
        },
        "conexion_resultados": {
            "slot": "D3-Q5",
            "expected_elements": [
                "conexion_producto_resultado",
                "mecanismo_causal_explicito",
            ],
        },
```

```
    },
    "keywords_general": [
        "producto",
        "output",
        "entregable",
        "bien",
        "servicio",
        "indicador de producto",
        "meta de producto",
        "MP-",
        "línea base",
        "meta cuatrienio",
        "fuente de verificación",
        "dosificación",
        "programación anual",
        "avance",
        "responsable",
        "secretaría",
        "dependencia",
        "entidad",
        "código BPIN",
        "proyecto de inversión",
        "PPI",
        "factible",
        "viable",
        "realista",
        "coherente",
        "genera",
        "produce",
        "contribuye a",
    ],
},
"DIM04": {
    "code": "DIM04",
    "name": "RESULTADOS",
    "label": "Resultados y Outcomes",
    "base_slots": ["D4-Q1", "D4-Q2", "D4-Q3", "D4-Q4", "D4-Q5"],
    "analytical_variables": {
        "indicadores_resultado": {
            "slot": "D4-Q1",
            "expected_elements": [
                "horizonte_temporal",
                "linea_base_resultado",
                "meta_resultado",
                "metrica_outcome",
            ],
        },
        "cadena_causal": {
            "slot": "D4-Q2",
            "expected_elements": [
                "cadena_causal_explicita",
                "condiciones_habilitantes",
                "supuestos_identificados",
            ],
```

```
        },
        "alcanzabilidad": {
            "slot": "D4-Q3",
            "expected_elements": [
                "evidencia_comparada",
                "justificacion_capacidad",
                "justificacion_recursos",
            ],
        },
        "coherencia_problematica": {
            "slot": "D4-Q4",
            "expected_elements": [
                "criterios_exito_definidos",
                "vinculo_resultado_problema",
            ],
        },
        "alineacion_estrategica": {
            "slot": "D4-Q5",
            "expected_elements": [
                "alineacion_ods",
                "alineacion_pnd",
            ],
        },
    },
    "keywords_general": [
        "resultado",
        "outcome",
        "efecto",
        "cambio",
        "transformación",
        "indicador de resultado",
        "meta de resultado",
        "MR-",
        "reducción",
        "incremento",
        "mejora",
        "disminución",
        "tasa",
        "porcentaje",
        "índice",
        "cobertura",
        "supuesto",
        "condición",
        "factor externo",
        "evidencia",
        "buena práctica",
        "caso exitoso",
        "ODS",
        "objetivo de desarrollo sostenible",
        "PND",
        "plan nacional de desarrollo",
        "política nacional",
    ],
},
```

```
"DIM05": {
    "code": "DIM05",
    "name": "IMPACTOS",
    "label": "Impactos de Largo Plazo",
    "base_slots": ["D5-Q1", "D5-Q2", "D5-Q3", "D5-Q4", "D5-Q5"],
    "analytical_variables": {
        "impacto_esperado": {
            "slot": "D5-Q1",
            "expected_elements": [
                "impacto_definido",
                "rezago_temporal",
                "ruta_transmision",
            ],
        },
        "medicion_impacto": {
            "slot": "D5-Q2",
            "expected_elements": [
                "justifica_validez",
                "usa_indices_compuestos",
                "usa_proxies",
            ],
        },
        "limitaciones_medicion": {
            "slot": "D5-Q3",
            "expected_elements": [
                "documenta_validez",
                "proxy_para_intangibles",
                "reconoce_limitaciones",
            ],
        },
        "riesgos_sistemicos": {
            "slot": "D5-Q4",
            "expected_elements": [
                "alineacion_marcos",
                "riesgos_sistemicos",
            ],
        },
        "realismo_impacto": {
            "slot": "D5-Q5",
            "expected_elements": [
                "analisis_realismo",
                "efectos_no_deseados",
                "hipotesis_limite",
            ],
        },
    },
    "keywords_general": [
        "impacto",
        "efecto de largo plazo",
        "transformación estructural",
        "indicador de impacto",
        "meta de impacto",
        "MI-",
        "bienestar",
```

```
        "calidad de vida",
        "desarrollo humano",
        "índice",
        "índice de desarrollo",
        "IDH",
        "IPM",
        "pobreza",
        "desigualdad",
        "Gini",
        "NBI",
        "sostenibilidad",
        "perdurabilidad",
        "irreversibilidad",
        "proxy",
        "aproximación",
        "medición indirecta",
        "riesgo sistémico",
        "efecto no deseado",
        "externalidad",
    ],
},
"DIM06": {
    "code": "DIM06",
    "name": "CAUSALIDAD",
    "label": "Teoría de Cambio",
    "base_slots": ["D6-Q1", "D6-Q2", "D6-Q3", "D6-Q4", "D6-Q5"],
    "analytical_variables": {
        "teoria_cambio": {
            "slot": "D6-Q1",
            "expected_elements": [
                "diagrama_causal",
                "supuestos_verificables",
                "teoria_cambio_explicita",
            ],
        },
        "coherencia_logica": {
            "slot": "D6-Q2",
            "expected_elements": [
                "evita_saltos_logicos",
                "proporcionalidad_eslabones",
            ],
        },
        "testabilidad": {
            "slot": "D6-Q3",
            "expected_elements": [
                "propone_pilotos_o_pruebas",
                "reconoce_inconsistencias",
            ],
        },
        "adaptabilidad": {
            "slot": "D6-Q4",
            "expected_elements": [
                "ciclos_aprendizaje",
                "mecanismos_correccion",
```

```
                    "sistema_monitoreo",
                ],
            },
            "contextualidad": {
                "slot": "D6-Q5",
                "expected_elements": [
                    "analisis_contextual",
                    "enfoque_diferencial",
                ],
            },
        },
        "keywords_general": [
            "teoría de cambio",
            "marco lógico",
            "cadena de valor",
            "si-entonces",
            "porque",
            "genera",
            "produce",
            "causa",
            "mecanismo causal",
            "vínculo causal",
            "conexión lógica",
            "supuesto",
            "hipótesis",
            "condición",
            "premisa",
            "eslabón",
            "nivel",
            "secuencia causal",
            "piloto",
            "prueba",
            "validación",
            "verificación",
            "monitoreo",
            "seguimiento",
            "evaluación",
            "ajuste",
            "contexto",
            "territorio",
            "enfoque diferencial",
            "particularidad",
        ],
    },
}


@dataclass
class ValidationResult:
    passed: bool
    errors: List[Dict[str, Any]]
    input_hashes: Dict[str, str]
```

```
def _sha256_file(path: Path) -> str:
    return hashlib.sha256(path.read_bytes()).hexdigest()


def validate_value_chain_spec(
    monolith_path: Path,
    output_path: Path,
) -> ValidationResult:
    monolith = json.loads(monolith_path.read_text())
    micro_questions = monolith.get("blocks", {}).get("micro_questions", [])
    universe_expected = set()
    for mq in micro_questions:
        for elem in mq.get("expected_elements", []) or []:
            if isinstance(elem, str):
                universe_expected.add(elem)
    errors: List[Dict[str, Any]] = []
    for dim_id, dim_cfg in VALUE_CHAIN_DIMENSIONS.items():
        base_slots = set(dim_cfg.get("base_slots", []))
        for var_name, var_cfg in dim_cfg.get("analytical_variables", {}).items():
            slot = var_cfg.get("slot")
            if slot not in base_slots:
                errors.append(
                    {
                        "dimension": dim_id,
                        "variable": var_name,
                        "error": "slot_not_in_base_slots",
                        "slot": slot,
                    }
                )
            for element in var_cfg.get("expected_elements", []):
                if element not in universe_expected:
                    errors.append(
                        {
                            "dimension": dim_id,
                            "variable": var_name,
                            "missing_expected_element": element,
                        }
                    )
    passed = len(errors) == 0
    input_hashes = {str(monolith_path): _sha256_file(monolith_path)}
    output = {
        "passed": passed,
        "errors": errors,
        "input_hashes": input_hashes,
        "record_counts": {"micro_questions": len(micro_questions)},
    }
    output_path.parent.mkdir(parents=True, exist_ok=True)
    output_path.write_text(json.dumps(output, ensure_ascii=False, indent=2))
    return ValidationResult(passed=passed, errors=errors, input_hashes=input_hashes)


__all__ = ["VALUE_CHAIN_DIMENSIONS", "validate_value_chain_spec", "ValidationResult"]
```

src/orchestration/__init__.py

```
"""Compatibility shim for legacy imports.

The orchestration implementation moved to `farfan_pipeline.orchestration`.
This module preserves historical imports like:

- `from orchestration.orchestrator import Orchestrator`

New code should prefer:
- `from farfan_pipeline.orchestration.orchestrator import Orchestrator`
"""

from __future__ import annotations

from pathlib import Path

# Redirect package submodule resolution to the new location.
__path__ = [
                str((Path(__file__).resolve().parent.parent    /    "farfan_pipeline"    /
"orchestration").resolve())
]

# Preserve the legacy convenience exports.
from farfan_pipeline.orchestration.meta_layer import (  # noqa: E402
    MetaLayerConfig,
    MetaLayerEvaluator,
    create_default_config as create_default_meta_config,
)
from farfan_pipeline.orchestration.congruence_layer import (  # noqa: E402
    CongruenceLayerConfig,
    CongruenceLayerEvaluator,
    create_default_congruence_config,
)
from farfan_pipeline.orchestration.chain_layer import (  # noqa: E402
    ChainLayerConfig,
    ChainLayerEvaluator,
    create_default_chain_config,
)

__all__ = [
    "MetaLayerConfig",
    "MetaLayerEvaluator",
    "create_default_meta_config",
    "CongruenceLayerConfig",
    "CongruenceLayerEvaluator",
    "create_default_congruence_config",
    "ChainLayerConfig",
    "ChainLayerEvaluator",
    "create_default_chain_config",
]
```