```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 21
================================================================================
Generated: 2025-12-07T06:17:28.008981
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/synchronization/irrigation_synchronizer.py
================================================================================

"""Irrigation synchronization and chunk matrix validation for policy analysis."""

from farfan_pipeline.core.orchestrator.chunk_matrix_builder import (
    CHUNK_ID_PATTERN,
    DIMENSIONS,
    EXPECTED_CHUNK_COUNT,
    POLICY_AREAS,
    build_chunk_matrix,
)
from farfan_pipeline.core.types import ChunkData, PreprocessedDocument


class ChunkMatrix:
    """Validates and provides O(1) access to policy chunks organized by area Ã\227 dimension.

    Delegates validation to core.orchestrator.chunk_matrix_builder to ensure a
    single source of truth aligned with questionnaire_monolith.json.
    """

    POLICY_AREAS = POLICY_AREAS
    DIMENSIONS = DIMENSIONS
    EXPECTED_CHUNK_COUNT = EXPECTED_CHUNK_COUNT
    CHUNK_ID_PATTERN = CHUNK_ID_PATTERN

    def __init__(self, document: PreprocessedDocument) -> None:
        matrix, sorted_keys = build_chunk_matrix(document)
        self.chunks: dict[tuple[str, str], ChunkData] = matrix
        self._matrix = self.chunks  # Backward compatibility for existing callers
        self.sorted_keys = tuple(sorted_keys)

    def get_chunk(self, policy_area_id: str, dimension_id: str) -> ChunkData:
        """Retrieve chunk by policy area and dimension with O(1) lookup."""
        key = (policy_area_id, dimension_id)
        if key not in self.chunks:
            raise KeyError(f"Chunk not found for key: {policy_area_id}-{dimension_id}")
        return self.chunks[key]



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

"""Utility functions and helpers for SAAAAAA system."""
```

```
57:
58:
59:
60: ================================================================================
61: FILE: src/farfan_pipeline/utils/contract_io.py
62: ================================================================================
63:
64: """
65: Contract I/O Envelope - Universal Metadata Wrapper
66: ==================================================
67:
68: ContractEnvelope wraps every phase payload (input and output) with
69: universal metadata including schema_version, timestamp_utc, policy_unit_id,
70: content_digest, correlation_id, and deterministic event_id.
71:
72: This module provides the canonical wrapper for all phase I/O to ensure
73: consistent metadata handling across the pipeline.
74:
75: Author: Policy Analytics Research Unit
76: Version: 1.0.0
77: License: Proprietary
78: """
79:
80: from __future__ import annotations
81:
82: import hashlib
83: import json
84: from datetime import datetime, timezone
85: from typing import TYPE_CHECKING, Any
86:
87: from pydantic import BaseModel, Field, field_validator
88: from farfan_pipeline.core.calibration.decorators import calibrated_method
89:
90: if TYPE_CHECKING:
91:     from collections.abc import Mapping
92:
93: CANONICAL_SCHEMA_VERSION = "io-1.0"
94:
95:
96: def _canonical_json_bytes(obj: Any) -> bytes:
97:     """
98:     Convert object to canonical JSON bytes for deterministic hashing.
99:
100:     Args:
101:         obj: Object to serialize
102:
103:     Returns:
104:         Canonical JSON bytes (sorted keys, compact separators)
105:     """
106:     return json.dumps(
107:         obj,
108:         separators=(",", ":"),
109:         sort_keys=True,
110:         ensure_ascii=False
111:     ).encode("utf-8")
112:
```

```
113:
114: def sha256_hex(obj: Any) -> str:
115:     """
116:     Compute SHA-256 hex digest of object via canonical JSON.
117:
118:     Args:
119:         obj: Object to hash
120:
121:     Returns:
122:         64-character hexadecimal SHA-256 digest
123:
124:     Examples:
125:         >>> sha256_hex({"b": 2, "a": 1})
126:         'eed6d51ab37ca6df16a330c85094467efcab7b5746c0e02bc728a05069ede38b'
127:         >>> sha256_hex({"a": 1, "b": 2})  # Same despite key order
128:         'eed6d51ab37ca6df16a330c85094467efcab7b5746c0e02bc728a05069ede38b'
129:     """
130:     return hashlib.sha256(_canonical_json_bytes(obj)).hexdigest()
131:
132:
133: def utcnow_iso() -> str:
134:     """
135:     Get current UTC timestamp in ISO-8601 format with Z suffix.
136:
137:     Returns:
138:         ISO-8601 timestamp string with Z suffix
139:
140:     Examples:
141:         >>> ts = utcnow_iso()
142:         >>> ts.endswith('Z')
143:         True
144:         >>> 'T' in ts
145:         True
146:     """
147:     # Always Z-suffixed UTC; forbidden to use local time
148:     return datetime.now(timezone.utc).isoformat().replace("+00:00", "Z")
149:
150:
151: class ContractEnvelope(BaseModel):
152:     """
153:     Universal metadata wrapper for phase I/O payloads.
154:
155:     Wraps every phase input/output with consistent metadata for:
156:     - Schema versioning
157:     - Temporal tracking (UTC only)
158:     - Policy scope identification
159:     - Cryptographic verification
160:     - Request correlation
161:     - Deterministic event tracking
162:
163:     Fields:
164:       - schema_version: Logical envelope schema version
165:       - timestamp_utc: ISO-8601 Z-suffixed timestamp
166:       - policy_unit_id: Stable identifier selecting seeds & scope
167:       - correlation_id: Client-supplied request/run correlation
168:       - content_digest: SHA-256 over canonical JSON of `payload`
```

```
      - event_id: Deterministic SHA-256 over (policy_unit_id, content_digest)
      - payload: The phase's typed deliverable/expectation

    Examples:
        >>> payload = {"result": "success", "data": [1, 2, 3]}
        >>> env = ContractEnvelope.wrap(
        ...     payload,
        ...     policy_unit_id="PDM-001",
        ...     correlation_id="req-123"
        ... )
        >>> env.policy_unit_id
        'PDM-001'
        >>> len(env.content_digest)
        64
        >>> len(env.event_id)
        64
    """

    schema_version: str = Field(default=CANONICAL_SCHEMA_VERSION)
    timestamp_utc: str = Field(default_factory=utcnow_iso)
    policy_unit_id: str = Field(min_length=1)
    correlation_id: str | None = Field(default=None)
    content_digest: str = Field(min_length=64, max_length=64)
    event_id: str = Field(min_length=64, max_length=64)
    payload: Mapping[str, Any] | Any

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

    @field_validator("timestamp_utc")
    @classmethod
    def _validate_utc(cls, v: str) -> str:
        """Validate timestamp is Z-suffixed UTC ISO-8601."""
        # Quick sanity: must end with Z and parse
        if not v.endswith("Z"):
            raise ValueError("timestamp_utc must be Z-suffixed UTC (ISO-8601)")
        # Let it raise if invalid
        datetime.fromisoformat(v.replace("Z", "+00:00"))
        return v

    @classmethod
    def wrap(
        cls,
        payload: Any,
        *,
        policy_unit_id: str,
        correlation_id: str | None = None,
        schema_version: str = CANONICAL_SCHEMA_VERSION,
    ) -> ContractEnvelope:
        """
        Wrap a payload with universal metadata envelope.

        Args:
            payload: The phase deliverable/expectation to wrap
            policy_unit_id: Policy unit identifier
            correlation_id: Optional request correlation ID
            schema_version: Schema version (default: io-1.0)
```

```
225:
226:         Returns:
227:             ContractEnvelope with computed digests and event ID
228:
229:         Examples:
230:             >>> payload = {"status": "ok"}
231:             >>> env = ContractEnvelope.wrap(payload, policy_unit_id="PDM-001")
232:             >>> env.payload["status"]
233:             'ok'
234:             >>> env.policy_unit_id
235:             'PDM-001'
236:         """
237:         digest = sha256_hex(payload)
238:         event_id = sha256_hex({"policy_unit_id": policy_unit_id, "digest": digest})
239:         return cls(
240:             schema_version=schema_version,
241:             policy_unit_id=policy_unit_id,
242:             correlation_id=correlation_id,
243:             content_digest=digest,
244:             event_id=event_id,
245:             payload=payload,
246:         )
247:
248:
249: if __name__ == "__main__":
250:     import doctest
251:
252:     # Run doctests
253:     print("Running doctests...")
254:     doctest.testmod(verbose=True)
255:
256:     # Minimal deterministic check
257:     print("\n" + "="*60)
258:     print("ContractEnvelope Integration Tests")
259:     print("="*60)
260:
261:     print("\n1. Testing deterministic digest computation:")
262:     p = {"a": 1, "b": [2, 3]}
263:     e1 = ContractEnvelope.wrap(p, policy_unit_id="PU_123")
264:     e2 = ContractEnvelope.wrap({"b": [2, 3], "a": 1}, policy_unit_id="PU_123")
265:     assert e1.content_digest == e2.content_digest  # canonical JSON stable
266:     assert e1.event_id == e2.event_id
267:     print(f"   â\234\223 Digest is deterministic: {e1.content_digest[:16]}...")
268:     print(f"   â\234\223 Event ID is deterministic: {e1.event_id[:16]}...")
269:
270:     print("\n2. Testing envelope immutability:")
271:     try:
272:         e1.payload = {"modified": True}
273:         print("   â\234\227 FAILED: Envelope should be immutable")
274:     except Exception:
275:         print("   â\234\223 Envelope is immutable (frozen)")
276:
277:     print("\n3. Testing UTC timestamp validation:")
278:     assert e1.timestamp_utc.endswith('Z')
279:     print(f"   â\234\223 Timestamp is UTC: {e1.timestamp_utc}")
280:
```

```
    print("\n4. Testing correlation ID:")
    e3 = ContractEnvelope.wrap(p, policy_unit_id="PU_123", correlation_id="corr-456")
    assert e3.correlation_id == "corr-456"
    print(f"   â\234\223 Correlation ID: {e3.correlation_id}")

    print("\n" + "="*60)
    print("ContractEnvelope doctest OK - All tests passed!")
    print("="*60)


================================================================================
FILE: src/farfan_pipeline/utils/contracts.py
================================================================================

"""
CONTRACT DEFINITIONS - Frozen Data Shapes
==========================================

TypedDict and Protocol definitions for API contracts across modules.
All data shapes must be versioned and adapters maintained for one release cycle.

Purpose: Replace ad-hoc dicts with typed structures to prevent:
- unexpected keyword argument errors
- missing required positional arguments
- 'str' object has no attribute 'text' errors
- 'bool' object is not iterable
- unhashable type: 'dict' in sets

Version 2.0 Enhancement:
- Pydantic-based contracts with strict validation (enhanced_contracts.py)
- Backward compatibility with V1 TypedDict contracts maintained
- Domain-specific exceptions and structured logging
- Cryptographic content verification and deterministic execution
"""

from __future__ import annotations

from dataclasses import dataclass
from typing import (
    TYPE_CHECKING,
    Any,
    Literal,
    Protocol,
    TypedDict,
)

if TYPE_CHECKING:
    from collections.abc import Iterable, Mapping, Sequence
    from pathlib import Path

# ============================================================================
# V2 ENHANCED CONTRACTS - Pydantic-based with strict validation
# ============================================================================
# Import V2 contracts from enhanced_contracts module
# Use these for new code; V1 contracts maintained for backward compatibility
```

```
337: from farfan_pipeline.utils.enhanced_contracts import (
338:     # Pydantic Models
339:     AnalysisInputV2,
340:     AnalysisOutputV2,
341:     BaseContract,
342:     # Exceptions
343:     ContractValidationError,
344:     DataIntegrityError,
345:     DocumentMetadataV2,
346:     ExecutionContextV2,
347:     FlowCompatibilityError,
348:     ProcessedTextV2,
349:     # Utilities
350:     StructuredLogger,
351:     SystemConfigError,
352:     compute_content_digest,
353:     utc_now_iso,
354: )
355:
356: # ============================================================================
357: # DOCUMENT CONTRACTS - V1
358: # ============================================================================
359:
360: class DocumentMetadataV1(TypedDict, total=True):
361:     """Document metadata shape - all fields required."""
362:     file_path: str
363:     file_name: str
364:     num_pages: int
365:     file_size_bytes: int
366:     file_hash: str
367:
368: class DocumentMetadataV1Optional(TypedDict, total=False):
369:     """Optional document metadata fields."""
370:     pdf_metadata: dict[str, Any]
371:     author: str
372:     title: str
373:     creation_date: str
374:
375: class ProcessedTextV1(TypedDict, total=True):
376:     """Shape for processed text output."""
377:     raw_text: str
378:     normalized_text: str
379:     language: str
380:     encoding: str
381:
382: class ProcessedTextV1Optional(TypedDict, total=False):
383:     """Optional processed text fields."""
384:     sentences: list[str]
385:     sections: list[dict[str, Any]]
386:     tables: Mapping[str, Any]
387:
388: # ============================================================================
389: # ANALYSIS CONTRACTS - V1
390: # ============================================================================
391:
392: class AnalysisInputV1(TypedDict, total=True):
```

```
393:     """Required fields for analysis input - keyword-only."""
394:     text: str
395:     document_id: str
396:
397: class AnalysisInputV1Optional(TypedDict, total=False):
398:     """Optional fields for analysis input."""
399:     metadata: Mapping[str, Any]
400:     context: Mapping[str, Any]
401:     sentences: Sequence[str]
402:
403: class AnalysisOutputV1(TypedDict, total=True):
404:     """Shape for analysis output."""
405:     dimension: str
406:     category: str
407:     confidence: float
408:     matches: Sequence[str]
409:
410: class AnalysisOutputV1Optional(TypedDict, total=False):
411:     """Optional analysis output fields."""
412:     positions: Sequence[int]
413:     evidence: Sequence[str]
414:     warnings: Sequence[str]
415:
416: # ============================================================================
417: # EXECUTION CONTRACTS - V1
418: # ============================================================================
419:
420: class ExecutionContextV1(TypedDict, total=True):
421:     """Execution context for method invocation."""
422:     class_name: str
423:     method_name: str
424:     document_id: str
425:
426: class ExecutionContextV1Optional(TypedDict, total=False):
427:     """Optional execution context fields."""
428:     raw_text: str
429:     text: str
430:     metadata: Mapping[str, Any]
431:     tables: Mapping[str, Any]
432:     sentences: Sequence[str]
433:
434: # ============================================================================
435: # ERROR REPORTING CONTRACTS
436: # ============================================================================
437:
438: class ContractMismatchError(TypedDict, total=True):
439:     """Standard error shape for contract mismatches."""
440:     error_code: Literal["ERR_CONTRACT_MISMATCH"]
441:     stage: str
442:     function: str
443:     parameter: str
444:     expected_type: str
445:     got_type: str
446:     producer: str
447:     consumer: str
448:
```

```
# ============================================================================
# PROTOCOLS FOR PLUGGABLE BEHAVIOR
# ============================================================================

class TextProcessorProtocol(Protocol):
    """Protocol for text processing components."""

    @calibrated_method("farfan_core.utils.contracts.TextProcessorProtocol.normalize_unicode")
    def normalize_unicode(self, text: str) -> str:
        """Normalize unicode characters in text."""
        ...

    @calibrated_method("farfan_core.utils.contracts.TextProcessorProtocol.segment_into_sentences")
    def segment_into_sentences(self, text: str) -> Sequence[str]:
        """Segment text into sentences."""
        ...

class DocumentLoaderProtocol(Protocol):
    """Protocol for document loading components."""

    @calibrated_method("farfan_core.utils.contracts.DocumentLoaderProtocol.load_pdf")
    def load_pdf(self, *, pdf_path: Path) -> DocumentMetadataV1:
        """Load PDF and return metadata - keyword-only params."""
        ...

    @calibrated_method("farfan_core.utils.contracts.DocumentLoaderProtocol.validate_pdf")
    def validate_pdf(self, *, pdf_path: Path) -> bool:
        """Validate PDF file - keyword-only params."""
        ...

class AnalyzerProtocol(Protocol):
    """Protocol for analysis components."""

    def analyze(
        self,
        *,
        text: str,
        document_id: str,
        metadata: Mapping[str, Any] | None = None,
    ) -> AnalysisOutputV1:
        """Analyze text and return structured output - keyword-only params."""
        ...

# ============================================================================
# VALUE OBJECTS (prevent .text on strings)
# ============================================================================

@dataclass(frozen=True, slots=True)
class TextDocument:
    """Wrapper to prevent passing plain str where structured text is required."""
    text: str
    document_id: str
    metadata: Mapping[str, Any]

    def __post_init__(self) -> None:
        """Validate that text is non-empty."""
```

```
505:         if not isinstance(self.text, str):
506:             raise TypeError(
507:                 f"ERR_CONTRACT_MISMATCH: text must be str, got {type(self.text).__name__}"
508:             )
509:         if not self.text:
510:             raise ValueError("ERR_CONTRACT_MISMATCH: text cannot be empty")
511:
512: @dataclass(frozen=True, slots=True)
513: class SentenceCollection:
514:     """Type-safe collection of sentences (prevents iteration bugs)."""
515:     sentences: tuple[str, ...]  # Immutable and hashable
516:
517:     def __post_init__(self) -> None:
518:         """Validate sentences are strings."""
519:         if not all(isinstance(s, str) for s in self.sentences):
520:             raise TypeError(
521:                 "ERR_CONTRACT_MISMATCH: All sentences must be strings"
522:             )
523:
524:     def __iter__(self) -> Iterable[str]:
525:         """Make iterable."""
526:         return iter(self.sentences)
527:
528:     def __len__(self) -> int:
529:         """Return count."""
530:         return len(self.sentences)
531:
532: # ============================================================================
533: # SENTINEL VALUES (avoid None ambiguity)
534: # ============================================================================
535:
536: class _MissingSentinel:
537:     """Sentinel type for missing optional parameters."""
538:
539:     def __repr__(self) -> str:
540:         return "<MISSING>"
541:
542: MISSING: _MissingSentinel = _MissingSentinel()
543:
544: # ============================================================================
545: # RUNTIME VALIDATION HELPERS
546: # ============================================================================
547:
548: def validate_contract(
549:     value: Any,
550:     expected_type: type,
551:     *,
552:     parameter: str,
553:     producer: str,
554:     consumer: str,
555: ) -> None:
556:     """
557:     Validate value matches expected contract at runtime.
558:
559:     Raises TypeError with structured error message on mismatch.
560:     """
```

```
    if not isinstance(value, expected_type):
        error_msg = (
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected={expected_type.__name__}, "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )
        raise TypeError(error_msg)

def validate_mapping_keys(
    mapping: Mapping[str, Any],
    required_keys: Sequence[str],
    *,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate mapping contains required keys.

    Raises KeyError with structured message on missing keys.
    """
    missing = [key for key in required_keys if key not in mapping]
    if missing:
        error_msg = (
            f"ERR_CONTRACT_MISMATCH["
            f"missing_keys={missing}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )
        raise KeyError(error_msg)

def ensure_iterable_not_string(
    value: Any,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate value is iterable but NOT a string or bytes.

    Prevents "'bool' object is not iterable" and "iterate string as tokens" bugs.
    """
    if isinstance(value, (str, bytes)):
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected=Iterable (not str/bytes), "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
```

```
        )

    try:
        iter(value)
    except TypeError as e:
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected=Iterable, "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        ) from e

def ensure_hashable(
    value: Any,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate value is hashable (can be added to set or used as dict key).

    Prevents "unhashable type: 'dict'" errors.
    """
    try:
        hash(value)
    except TypeError as e:
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected=Hashable, "
            f"got={type(value).__name__} (unhashable), "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        ) from e


# ============================================================================
# MODULE EXPORTS
# ============================================================================

__all__ = [
    # V2 Enhanced Contracts (Pydantic-based) - RECOMMENDED FOR NEW CODE
    "AnalysisInputV2",
    "AnalysisOutputV2",
    "BaseContract",
    "DocumentMetadataV2",
    "ExecutionContextV2",
    "ProcessedTextV2",
    # V2 Exceptions
    "ContractValidationError",
    "DataIntegrityError",
```

```
    "FlowCompatibilityError",
    "SystemConfigError",
    # V2 Utilities
    "StructuredLogger",
    "compute_content_digest",
    "utc_now_iso",
    # V1 Contracts (TypedDict-based) - BACKWARD COMPATIBILITY
    "AnalysisInputV1",
    "AnalysisInputV1Optional",
    "AnalysisOutputV1",
    "AnalysisOutputV1Optional",
    "AnalyzerProtocol",
    "ContractMismatchError",
    "DocumentLoaderProtocol",
    "DocumentMetadataV1",
    "DocumentMetadataV1Optional",
    "ExecutionContextV1",
    "ExecutionContextV1Optional",
    "MISSING",
    "ProcessedTextV1",
    "ProcessedTextV1Optional",
    "SentenceCollection",
    "TextDocument",
    "TextProcessorProtocol",
    "ensure_hashable",
    "ensure_iterable_not_string",
    "validate_contract",
    "validate_mapping_keys",
]



================================================================================
FILE: src/farfan_pipeline/utils/contracts_runtime.py
================================================================================

"""
Runtime Contract Validation using Pydantic.

This module provides runtime validators for all TypedDict contracts defined
in core_contracts.py. These validators enforce:
- Value bounds and constraints
- Required vs optional fields with strict validation
- Schema versioning for backward compatibility
- Round-trip serialization guarantees

The validators mirror the TypedDict shapes exactly but add runtime enforcement.
Use these at public API boundaries and orchestrator edges.

Version: 1.0.0
Schema Version Format: sem-{major}.{minor}
"""

from typing import Any

from pydantic import BaseModel, ConfigDict, Field, field_validator
```

```
729: from farfan_pipeline.core.calibration.decorators import calibrated_method
730:
731: # ============================================================================
732: # CONFIGURATION
733: # ============================================================================
734:
735: class StrictModel(BaseModel):
736:     """Base model with strict configuration for all contract validators."""
737:
738:     model_config = ConfigDict(
739:         extra='forbid',  # Refuse unknown fields
740:         validate_assignment=True,
741:         str_strip_whitespace=True,
742:         validate_default=True,
743:         populate_by_name=True,  # Allow both field name and alias
744:     )
745:
746: # ============================================================================
747: # ANALYZER_ONE.PY CONTRACTS
748: # ============================================================================
749:
750: class SemanticAnalyzerInputModel(StrictModel):
751:     """Runtime validator for SemanticAnalyzerInputContract.
752:
753:     Validates:
754:     - text is non-empty
755:     - schema_version follows sem-X.Y pattern
756:     - segments is a list of strings
757:     - ontology_params is a valid dict
758:
759:     Example:
760:         >>> model = SemanticAnalyzerInputModel(
761:         ...     text="El plan de desarrollo municipal...",
762:         ...     schema_version="sem-1.0"
763:         ... )
764:     """
765:     text: str = Field(min_length=1, description="Document text to analyze")
766:     segments: list[str] = Field(
767:         default_factory=list,
768:         description="Pre-segmented text chunks"
769:     )
770:     ontology_params: dict[str, Any] = Field(
771:         default_factory=dict,
772:         description="Domain-specific ontology parameters"
773:     )
774:     schema_version: str = Field(
775:         default="sem-1.0",
776:         pattern=r"^sem-\d+\.\d+$",
777:         description="Contract schema version"
778:     )
779:
780:     @field_validator('text')
781:     @classmethod
782:     def text_not_empty(cls, v: str) -> str:
783:         """Ensure text is not just whitespace."""
784:         if not v or not v.strip():
```

```
785:             raise ValueError("text must contain non-whitespace characters")
786:         return v
787:
788: class SemanticAnalyzerOutputModel(StrictModel):
789:     """Runtime validator for SemanticAnalyzerOutputContract."""
790:     semantic_cube: dict[str, Any] = Field(description="Semantic analysis results")
791:     coherence_score: float = Field(ge=0.0, le=1.0, description="Coherence metric")
792:     complexity_score: float = Field(ge=0.0, description="Complexity metric")
793:     domain_classification: dict[str, float] = Field(
794:         description="Domain probability distribution"
795:     )
796:     schema_version: str = Field(
797:         default="sem-1.0",
798:         pattern=r"^sem-\d+\.\d+$"
799:     )
800:
801:     @field_validator('domain_classification')
802:     @classmethod
803:     def validate_probabilities(cls, v: dict[str, float]) -> dict[str, float]:
804:         """Ensure all domain probabilities are in [0, 1]."""
805:         for domain, prob in v.items():
806:             if not (0.0 <= prob <= 1.0):
807:                 raise ValueError(f"Probability for {domain} must be in [0, 1], got {prob}")
808:         return v
809:
810: # ============================================================================
811: # DERECK_BEACH.PY CONTRACTS
812: # ============================================================================
813:
814: class CDAFFrameworkInputModel(StrictModel):
815:     """Runtime validator for CDAFFrameworkInputContract."""
816:     document_text: str = Field(min_length=1, description="Document to analyze")
817:     plan_metadata: dict[str, Any] = Field(
818:         default_factory=dict,
819:         description="Metadata about the plan"
820:     )
821:     config: dict[str, Any] = Field(
822:         default_factory=dict,
823:         description="Framework configuration"
824:     )
825:     schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
826:
827: class CDAFFrameworkOutputModel(StrictModel):
828:     """Runtime validator for CDAFFrameworkOutputContract."""
829:     causal_mechanisms: list[dict[str, Any]] = Field(
830:         default_factory=list,
831:         description="Identified causal mechanisms"
832:     )
833:     evidential_tests: dict[str, Any] = Field(
834:         default_factory=dict,
835:         description="Statistical test results"
836:     )
837:     bayesian_inference: dict[str, Any] = Field(
838:         default_factory=dict,
839:         description="Bayesian analysis results"
840:     )
```

```
841:     audit_results: dict[str, Any] = Field(
842:         default_factory=dict,
843:         description="Audit findings and recommendations"
844:     )
845:     schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
846:
847: # ============================================================================
848: # FINANCIERO_VIABILIDAD_TABLAS.PY CONTRACTS
849: # ============================================================================
850:
851: class PDETAnalyzerInputModel(StrictModel):
852:     """Runtime validator for PDETAnalyzerInputContract."""
853:     document_content: str = Field(min_length=1, description="Document content")
854:     extract_tables: bool = Field(default=True, description="Whether to extract tables")
855:     config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
856:     schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
857:
858: class PDETAnalyzerOutputModel(StrictModel):
859:     """Runtime validator for PDETAnalyzerOutputContract."""
860:     extracted_tables: list[dict[str, Any]] = Field(
861:         default_factory=list,
862:         description="Extracted financial tables"
863:     )
864:     financial_indicators: dict[str, float] = Field(
865:         default_factory=dict,
866:         description="Calculated financial metrics"
867:     )
868:     viability_score: float = Field(
869:         ge=0.0, le=1.0,
870:         description="Overall viability score"
871:     )
872:     quality_scores: dict[str, float] = Field(
873:         default_factory=dict,
874:         description="Quality assessment scores"
875:     )
876:     schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
877:
878: # ============================================================================
879: # TEORIA_CAMBIO.PY CONTRACTS
880: # ============================================================================
881:
882: class TeoriaCambioInputModel(StrictModel):
883:     """Runtime validator for TeoriaCambioInputContract."""
884:     document_text: str = Field(min_length=1, description="Document to analyze")
885:     strategic_goals: list[str] = Field(
886:         default_factory=list,
887:         description="Identified strategic goals"
888:     )
889:     config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
890:     schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")
891:
892: class TeoriaCambioOutputModel(StrictModel):
893:     """Runtime validator for TeoriaCambioOutputContract."""
894:     causal_dag: dict[str, Any] = Field(
895:         default_factory=dict,
896:         description="Causal directed acyclic graph"
```

```
    )
    validation_results: dict[str, Any] = Field(
        default_factory=dict,
        description="Model validation results"
    )
    monte_carlo_results: dict[str, Any] | None = Field(
        default=None,
        description="Monte Carlo simulation results"
    )
    graph_visualizations: list[dict[str, Any]] | None = Field(
        default=None,
        description="Graph visualization data"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# CONTRADICTION_DETECCION.PY CONTRACTS
# ============================================================================

class ContradictionDetectorInputModel(StrictModel):
    """Runtime validator for ContradictionDetectorInputContract."""
    text: str = Field(min_length=1, description="Text to analyze for contradictions")
    plan_name: str = Field(min_length=1, description="Name of the plan")
    dimension: str | None = Field(
        default=None,
        description="PolicyDimension enum value"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class ContradictionDetectorOutputModel(StrictModel):
    """Runtime validator for ContradictionDetectorOutputContract."""
    contradictions: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Detected contradictions"
    )
    confidence_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Confidence in each detection"
    )
    temporal_conflicts: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Temporal inconsistencies"
    )
    severity_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Severity ratings"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# EMBEDDING_POLICY.PY CONTRACTS
# ============================================================================

class EmbeddingPolicyInputModel(StrictModel):
    """Runtime validator for EmbeddingPolicyInputContract."""
```

```
    text: str = Field(min_length=1, description="Text to embed")
    dimensions: list[str] = Field(
        default_factory=list,
        description="Policy dimensions to analyze"
    )
    embedding_model_config: dict[str, Any] = Field(
        default_factory=dict,
        description="Embedding model configuration",
        alias="model_config"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class EmbeddingPolicyOutputModel(StrictModel):
    """Runtime validator for EmbeddingPolicyOutputContract."""
    embeddings: list[list[float]] = Field(
        default_factory=list,
        description="Generated embeddings"
    )
    similarity_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Similarity metrics"
    )
    bayesian_evaluation: dict[str, Any] = Field(
        default_factory=dict,
        description="Bayesian evaluation results"
    )
    policy_metrics: dict[str, float] = Field(
        default_factory=dict,
        description="Policy-specific metrics"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# SEMANTIC_CHUNKING_POLICY.PY CONTRACTS
# ============================================================================

class SemanticChunkingInputModel(StrictModel):
    """Runtime validator for SemanticChunkingInputContract."""
    text: str = Field(min_length=1, description="Text to chunk")
    preserve_structure: bool = Field(
        default=True,
        description="Whether to preserve document structure"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class SemanticChunkingOutputModel(StrictModel):
    """Runtime validator for SemanticChunkingOutputContract."""
    chunks: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Semantic chunks"
    )
    causal_dimensions: dict[str, dict[str, Any]] = Field(
        default_factory=dict,
        description="Causal dimension analysis"
    )
```

```
    key_excerpts: dict[str, list[str]] = Field(
        default_factory=dict,
        description="Key excerpts by category"
    )
    summary: dict[str, Any] = Field(
        default_factory=dict,
        description="Summary statistics"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# POLICY_PROCESSOR.PY CONTRACTS
# ============================================================================

class PolicyProcessorInputModel(StrictModel):
    """Runtime validator for PolicyProcessorInputContract."""
    data: Any = Field(description="Raw data to process")
    text: str = Field(min_length=1, description="Text content")
    sentences: list[str] = Field(
        default_factory=list,
        description="Pre-segmented sentences"
    )
    tables: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Extracted tables"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class PolicyProcessorOutputModel(StrictModel):
    """Runtime validator for PolicyProcessorOutputContract."""
    processed_data: dict[str, Any] = Field(
        default_factory=dict,
        description="Processed results"
    )
    evidence_bundles: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Evidence bundles"
    )
    bayesian_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Bayesian scores"
    )
    matched_patterns: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Matched patterns"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# EXPORTS
# ============================================================================

__all__ = [
    # Base
    'StrictModel',
```

```
1065:
1066:     # Analyzer_one
1067:     'SemanticAnalyzerInputModel',
1068:     'SemanticAnalyzerOutputModel',
1069:
1070:     # derek_beach
1071:     'CDAFFrameworkInputModel',
1072:     'CDAFFrameworkOutputModel',
1073:
1074:     # financiero_viabilidad_tablas
1075:     'PDETAnalyzerInputModel',
1076:     'PDETAnalyzerOutputModel',
1077:
1078:     # teoria_cambio
1079:     'TeoriaCambioInputModel',
1080:     'TeoriaCambioOutputModel',
1081:
1082:     # contradiction_deteccion
1083:     'ContradictionDetectorInputModel',
1084:     'ContradictionDetectorOutputModel',
1085:
1086:     # embedding_policy
1087:     'EmbeddingPolicyInputModel',
1088:     'EmbeddingPolicyOutputModel',
1089:
1090:     # semantic_chunking_policy
1091:     'SemanticChunkingInputModel',
1092:     'SemanticChunkingOutputModel',
1093:
1094:     # policy_processor
1095:     'PolicyProcessorInputModel',
1096:     'PolicyProcessorOutputModel',
1097: ]
1098:
1099:
1100:
1101: ================================================================================
1102: FILE: src/farfan_pipeline/utils/core_contracts.py
1103: ================================================================================
1104:
1105: """
1106: Core Module Contracts - Type-safe API boundaries for pure library modules.
1107:
1108: This module defines InputContract and OutputContract TypedDicts for each
1109: core module to establish clear API boundaries and enable dependency injection.
1110:
1111: Architectural Principles:
1112: - Core modules receive all data via InputContract parameters
1113: - Core modules return data via OutputContract structures
1114: - No I/O operations within core modules
1115: - All I/O happens in orchestrator/factory.py
1116: - Type-safe contracts with strict typing
1117:
1118: Version: 1.1.0
1119: Schema Version: sem-1.0 (initial stable release)
1120: Status: Active - Runtime validation available in contracts_runtime.py
```

```
"""

from typing import Any, TypedDict
from farfan_pipeline.core.calibration.decorators import calibrated_method

try:
    from typing import NotRequired  # Python 3.11+
except ImportError:
    from typing_extensions import NotRequired  # Python 3.9-3.10

# ============================================================================
# ANALYZER_ONE.PY CONTRACTS
# ============================================================================

class SemanticAnalyzerInputContract(TypedDict):
    """Input contract for SemanticAnalyzer methods.

    Example:
        {
            "text": "El plan de desarrollo municipal...",
            "segments": ["Segment 1", "Segment 2"],
            "ontology_params": {"domain": "municipal"}
        }
    """
    text: str
    segments: NotRequired[list[str]]
    ontology_params: NotRequired[dict[str, Any]]

class SemanticAnalyzerOutputContract(TypedDict):
    """Output contract for SemanticAnalyzer methods."""
    semantic_cube: dict[str, Any]
    coherence_score: float
    complexity_score: float
    domain_classification: dict[str, float]

# ============================================================================
# DERECK_BEACH.PY CONTRACTS
# ============================================================================

class CDAFFrameworkInputContract(TypedDict):
    """Input contract for CDAFFramework (Causal Deconstruction Audit Framework)."""
    document_text: str
    plan_metadata: dict[str, Any]
    config: NotRequired[dict[str, Any]]

class CDAFFrameworkOutputContract(TypedDict):
    """Output contract for CDAFFramework."""
    causal_mechanisms: list[dict[str, Any]]
    evidential_tests: dict[str, Any]
    bayesian_inference: dict[str, Any]
    audit_results: dict[str, Any]

# ============================================================================
# FINANCIERO_VIABILIDAD_TABLAS.PY CONTRACTS
# ============================================================================

```

```
class PDETAnalyzerInputContract(TypedDict):
    """Input contract for PDET (Programas de Desarrollo con Enfoque Territorial) Analyzer."""
    document_content: str
    extract_tables: NotRequired[bool]
    config: NotRequired[dict[str, Any]]

class PDETAnalyzerOutputContract(TypedDict):
    """Output contract for PDET Analyzer."""
    extracted_tables: list[dict[str, Any]]
    financial_indicators: dict[str, float]
    viability_score: float
    quality_scores: dict[str, float]

# ============================================================================
# TEORIA_CAMBIO.PY CONTRACTS
# ============================================================================

class TeoriaCambioInputContract(TypedDict):
    """Input contract for Theory of Change analysis."""
    document_text: str
    strategic_goals: NotRequired[list[str]]
    config: NotRequired[dict[str, Any]]

class TeoriaCambioOutputContract(TypedDict):
    """Output contract for Theory of Change analysis."""
    causal_dag: dict[str, Any]
    validation_results: dict[str, Any]
    monte_carlo_results: NotRequired[dict[str, Any]]
    graph_visualizations: NotRequired[list[dict[str, Any]]]

# ============================================================================
# CONTRADICTION_DETECCION.PY CONTRACTS
# ============================================================================

class ContradictionDetectorInputContract(TypedDict):
    """Input contract for PolicyContradictionDetector."""
    text: str
    plan_name: str
    dimension: NotRequired[str]  # PolicyDimension enum value
    config: NotRequired[dict[str, Any]]

class ContradictionDetectorOutputContract(TypedDict):
    """Output contract for PolicyContradictionDetector."""
    contradictions: list[dict[str, Any]]
    confidence_scores: dict[str, float]
    temporal_conflicts: list[dict[str, Any]]
    severity_scores: dict[str, float]

# ============================================================================
# EMBEDDING_POLICY.PY CONTRACTS
# ============================================================================

class EmbeddingPolicyInputContract(TypedDict):
    """Input contract for embedding-based policy analysis."""
    text: str
    dimensions: NotRequired[list[str]]
```

```
    model_config: NotRequired[dict[str, Any]]

class EmbeddingPolicyOutputContract(TypedDict):
    """Output contract for embedding policy analysis."""
    embeddings: list[list[float]]
    similarity_scores: dict[str, float]
    bayesian_evaluation: dict[str, Any]
    policy_metrics: dict[str, float]

# ============================================================================
# SEMANTIC_CHUNKING_POLICY.PY CONTRACTS
# ============================================================================

class SemanticChunkingInputContract(TypedDict):
    """Input contract for semantic chunking and policy document analysis."""
    text: str
    preserve_structure: NotRequired[bool]
    config: NotRequired[dict[str, Any]]

class SemanticChunkingOutputContract(TypedDict):
    """Output contract for semantic chunking."""
    chunks: list[dict[str, Any]]
    causal_dimensions: dict[str, dict[str, Any]]
    key_excerpts: dict[str, list[str]]
    summary: dict[str, Any]

# ============================================================================
# POLICY_PROCESSOR.PY CONTRACTS
# ============================================================================

class PolicyProcessorInputContract(TypedDict):
    """Input contract for IndustrialPolicyProcessor."""
    data: Any
    text: str
    sentences: NotRequired[list[str]]
    tables: NotRequired[list[dict[str, Any]]]
    config: NotRequired[dict[str, Any]]

class PolicyProcessorOutputContract(TypedDict):
    """Output contract for IndustrialPolicyProcessor."""
    processed_data: dict[str, Any]
    evidence_bundles: list[dict[str, Any]]
    bayesian_scores: dict[str, float]
    matched_patterns: list[dict[str, Any]]

# ============================================================================
# SHARED DATA STRUCTURES
# ============================================================================

class DocumentData(TypedDict):
    """Standard document data structure from farfan_core.core.orchestrator.

    This is what the orchestrator/factory provides to core modules.
    """
    raw_text: str
    sentences: list[str]
```

```
    tables: list[dict[str, Any]]
    metadata: dict[str, Any]

__all__ = [
    # Analyzer_one
    'SemanticAnalyzerInputContract',
    'SemanticAnalyzerOutputContract',

    # derek_beach
    'CDAFFrameworkInputContract',
    'CDAFFrameworkOutputContract',

    # financiero_viabilidad_tablas
    'PDETAnalyzerInputContract',
    'PDETAnalyzerOutputContract',

    # teoria_cambio
    'TeoriaCambioInputContract',
    'TeoriaCambioOutputContract',

    # contradiction_deteccion
    'ContradictionDetectorInputContract',
    'ContradictionDetectorOutputContract',

    # embedding_policy
    'EmbeddingPolicyInputContract',
    'EmbeddingPolicyOutputContract',

    # semantic_chunking_policy
    'SemanticChunkingInputContract',
    'SemanticChunkingOutputContract',

    # policy_processor
    'PolicyProcessorInputContract',
    'PolicyProcessorOutputContract',

    # Shared
    'DocumentData',
]



================================================================================
FILE: src/farfan_pipeline/utils/coverage_gate.py
================================================================================

#!/usr/bin/env python3
"""
Coverage Enforcement Gate
=========================
Enforces hard-fail at <555 methods threshold + audit.json emission

Requirements:
- Count all public methods across Producer classes
- Generate audit.json with method counts and validation results
- Hard-fail if total methods < 555
```

```
- Include schema validation results
"""

import ast
import json
import sys
from datetime import datetime
from pathlib import Path
from farfan_pipeline.core.calibration.decorators import calibrated_method


def count_methods_in_class(filepath: Path, class_name: str) -> tuple[list[str], dict[str, int]]:
    """Count public and private methods in a class and return method names"""
    if not filepath.exists():
        return [], {"public": 0, "private": 0, "total": 0}

    with open(filepath, encoding='utf-8') as f:
        tree = ast.parse(f.read())

    method_names = []
    method_counts = {
        "public": 0,
        "private": 0,
        "total": 0
    }

    for node in ast.walk(tree):
        if isinstance(node, ast.ClassDef) and node.name == class_name:
            for item in node.body:
                if isinstance(item, ast.FunctionDef):
                    method_names.append(item.name)
                    if not item.name.startswith('_'):
                        method_counts["public"] += 1
                    else:
                        method_counts["private"] += 1
                    method_counts["total"] += 1

    return method_names, method_counts

def validate_schema_exists(module_dir: Path) -> tuple[bool, list[str]]:
    """Validate that schema files exist for a module"""
    if not module_dir.exists():
        return False, []

    schema_files = list(module_dir.glob("*.schema.json"))
    return len(schema_files) > 0, [f.name for f in schema_files]

def count_file_methods(filepath: Path) -> tuple[int, int]:
    """Count all public and total methods in a file"""
    if not filepath.exists():
        return 0, 0

    with open(filepath, encoding='utf-8') as f:
        try:
            tree = ast.parse(f.read())
            public_methods = 0
```

```
            all_methods = 0

            for node in ast.walk(tree):
                if isinstance(node, ast.FunctionDef):
                    all_methods += 1
                    if not node.name.startswith('_'):
                        public_methods += 1

            return public_methods, all_methods
        except Exception as e:
            print(f"Error parsing {filepath}: {e}")
            return 0, 0

def count_all_methods() -> dict[str, any]:
    """Count all methods across all modules and producers"""

    # All files to analyze
    files_to_analyze = [
        "financiero_viabilidad_tablas.py",
        "Analyzer_one.py",
        "contradiction_deteccion.py",
        "embedding_policy.py",
        "teoria_cambio.py",
        "derek_beach.py",
        "policy_processor.py",
        "report_assembly.py",
        "semantic_chunking_policy.py"
    ]

    # Producer classes to check
    producers = {
        "SemanticChunkingProducer": "semantic_chunking_policy.py",
        "EmbeddingPolicyProducer": "embedding_policy.py",
        "DerekBeachProducer": "derek_beach.py",
        "ReportAssemblyProducer": "report_assembly.py"
    }

    results = {
        "timestamp": datetime.now().isoformat(),
        "files": {},
        "producers": {},
        "totals": {
            "file_public_methods": 0,
            "file_total_methods": 0,
            "producer_methods": 0,
            "threshold": 555,
            "meets_threshold": False
        },
        "schema_validation": {},
        "audit_status": "PENDING"
    }

    # Count file methods
    print("=" * 80)
    print("FILE METHOD COUNTS")
    print("=" * 80)
```

```
1457:
1458:     for filepath_str in files_to_analyze:
1459:         filepath = Path(filepath_str)
1460:         public_methods, total_methods = count_file_methods(filepath)
1461:         results["files"][filepath_str] = {
1462:             "public_methods": public_methods,
1463:             "total_methods": total_methods
1464:         }
1465:         results["totals"]["file_public_methods"] += public_methods
1466:         results["totals"]["file_total_methods"] += total_methods
1467:         print(f"{filepath_str:45} | {public_methods:4} public | {total_methods:4} total")
1468:
1469:     # Count Producer methods
1470:     print("\n" + "=" * 80)
1471:     print("PRODUCER METHOD COUNTS")
1472:     print("=" * 80)
1473:
1474:     for class_name, filepath in producers.items():
1475:         methods, counts = count_methods_in_class(Path(filepath), class_name)
1476:         results["producers"][class_name] = {
1477:             "file": filepath,
1478:             "methods": methods,
1479:             "counts": counts,
1480:             "public_methods": counts["public"]
1481:         }
1482:         results["totals"]["producer_methods"] += counts["public"]
1483:         print(f"{class_name:45} | {counts['public']:3} public | {counts['private']:3} private | {counts['total']:3} total")
1484:
1485:     # Update meets_threshold
1486:     results["totals"]["meets_threshold"] = (
1487:         results["totals"]["file_total_methods"] >= 555
1488:     )
1489:
1490:     # Validate schemas
1491:     print("\n" + "=" * 80)
1492:     print("SCHEMA VALIDATION")
1493:     print("=" * 80)
1494:
1495:     schema_modules = [
1496:         "semantic_chunking_policy",
1497:         "embedding_policy",
1498:         "derek_beach",
1499:         "report_assembly"
1500:     ]
1501:
1502:     for module in schema_modules:
1503:         module_dir = Path("schemas") / module
1504:         has_schemas, schema_files = validate_schema_exists(module_dir)
1505:         results["schema_validation"][module] = {
1506:             "has_schemas": has_schemas,
1507:             "schema_files": schema_files,
1508:             "schema_count": len(schema_files)
1509:         }
1510:         status = "â\234\223" if has_schemas else "â\234\227"
1511:         print(f"{module:35} | {status} | {len(schema_files)} schemas")
1512:
```

```
    # Determine audit status
    all_have_schemas = all(
        v["has_schemas"] for v in results["schema_validation"].values()
    )

    if results["totals"]["meets_threshold"] and all_have_schemas:
        results["audit_status"] = "PASS"
    else:
        results["audit_status"] = "FAIL"

    return results

def main() -> int:
    """Main entry point"""
    print("\n" + "=" * 80)
    print("COVERAGE ENFORCEMENT GATE")
    print("=" * 80 + "\n")

    # Count all methods
    results = count_all_methods()

    # Print summary
    print("\n" + "=" * 80)
    print("SUMMARY")
    print("=" * 80)
    print(f"Total file methods:      {results['totals']['file_total_methods']:4}")
    print(f"Total public methods:    {results['totals']['file_public_methods']:4}")
    print(f"Producer methods:        {results['totals']['producer_methods']:4}")
    print(f"Threshold:               {results['totals']['threshold']:4}")
    print(f"Meets threshold:         {results['totals']['meets_threshold']}")
    print(f"All schemas present:     {all(v['has_schemas'] for v in results['schema_validation'].values())}")
    print(f"Audit status:            {results['audit_status']}")

    # Save audit.json
    audit_path = Path("audit.json")
    with open(audit_path, 'w', encoding='utf-8') as f:
        json.dump(results, f, indent=2)

    print(f"\nâ\234\223 Audit results saved to {audit_path}")

    # Enforce hard-fail
    if not results['totals']['meets_threshold']:
        print("\n" + "=" * 80)
        print("â\235\214 COVERAGE GATE FAILED")
        print("=" * 80)
        print(f"Required: {results['totals']['threshold']} methods")
        print(f"Found:    {results['totals']['file_total_methods']} methods")
        print(f"Gap:      {results['totals']['threshold'] - results['totals']['file_total_methods']} methods")
        print("=" * 80 + "\n")
        return 1

    # Check schema validation
    if not all(v['has_schemas'] for v in results['schema_validation'].values()):
        print("\n" + "=" * 80)
        print("â\235\214 SCHEMA VALIDATION FAILED")
        print("=" * 80)
```

```
1569:         for module, validation in results['schema_validation'].items():
1570:             if not validation['has_schemas']:
1571:                 print(f"Missing schemas for: {module}")
1572:         print("=" * 80 + "\n")
1573:         return 1
1574:
1575:     print("\n" + "=" * 80)
1576:     print("â\234\223 COVERAGE GATE PASSED")
1577:     print("=" * 80)
1578:     print(f"All {results['totals']['file_total_methods']} methods accounted for")
1579:     print(f"{results['totals']['file_public_methods']} public methods available")
1580:     print(f"{results['totals']['producer_methods']} producer methods exposed")
1581:     print("All schema contracts validated")
1582:     print("=" * 80 + "\n")
1583:
1584:     return 0
1585:
1586: if __name__ == "__main__":
1587:     sys.exit(main())
1588:
1589:
1590:
1591: ================================================================================
1592: FILE: src/farfan_pipeline/utils/cpp_adapter.py
1593: ================================================================================
1594:
1595: """CPP to Orchestrator Adapter.
1596:
1597: This adapter converts Canon Policy Package (CPP) documents from the ingestion pipeline
1598: into the orchestrator's PreprocessedDocument format.
1599:
1600: Note: This is the canonical adapter implementation. SPC (Smart Policy Chunks) is the
1601: precursor to CPP.
1602:
1603: Design Principles:
1604: - Preserves complete provenance information
1605: - Orders chunks by text_span.start for deterministic ordering
1606: - Computes provenance_completeness metric
1607: - Provides prescriptive error messages on failure
1608: - Supports micro, meso, and macro chunk resolutions
1609: - Optional dependencies handled gracefully (pyarrow, structlog)
1610: """
1611:
1612: from __future__ import annotations
1613:
1614: import logging
1615: from datetime import datetime, timezone
1616: from types import MappingProxyType
1617: from typing import Any
1618:
1619: from farfan_pipeline.core.parameters import ParameterLoaderV2
1620: from farfan_pipeline.core.types import ChunkData, PreprocessedDocument, Provenance
1621:
1622: logger = logging.getLogger(__name__)
1623:
1624: _EMPTY_MAPPING = MappingProxyType({})
```

```
1625:
1626:
1627: class CPPAdapterError(Exception):
1628:     """Raised when CPP to PreprocessedDocument conversion fails."""
1629:
1630:     pass
1631:
1632:
1633: class CPPAdapter:
1634:     """
1635:     Adapter to convert CanonPolicyPackage (CPP output) to PreprocessedDocument.
1636:
1637:     This is the canonical adapter for the FARFAN pipeline, converting the rich
1638:     CanonPolicyPackage data into the format expected by the orchestrator.
1639:     """
1640:
1641:     def __init__(self, enable_runtime_validation: bool = True) -> None:
1642:         """Initialize the CPP adapter.
1643:
1644:         Args:
1645:             enable_runtime_validation: Enable WiringValidator for runtime contract checking
1646:         """
1647:         self.logger = logging.getLogger(self.__class__.__name__)
1648:
1649:         # Initialize WiringValidator for runtime contract validation
1650:         self.enable_runtime_validation = enable_runtime_validation
1651:         if enable_runtime_validation:
1652:             try:
1653:                 from farfan_pipeline.core.wiring.validation import WiringValidator
1654:
1655:                 self.wiring_validator = WiringValidator()
1656:                 self.logger.info(
1657:                     "WiringValidator enabled for runtime contract checking"
1658:                 )
1659:             except ImportError:
1660:                 self.logger.warning(
1661:                     "WiringValidator not available. Runtime validation disabled."
1662:                 )
1663:                 self.wiring_validator = None
1664:         else:
1665:             self.wiring_validator = None
1666:
1667:     def to_preprocessed_document(
1668:         self, canon_package: Any, document_id: str
1669:     ) -> PreprocessedDocument:
1670:         """
1671:         Convert CanonPolicyPackage to PreprocessedDocument.
1672:
1673:         Args:
1674:             canon_package: CanonPolicyPackage from ingestion
1675:             document_id: Unique document identifier
1676:
1677:         Returns:
1678:             PreprocessedDocument ready for orchestrator
1679:
1680:         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: 'SPC-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 SPC/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"SPC canonical format."
            )
```

```
1793:
1794:         # Enforce metadata integrity
1795:         for idx, chunk in enumerate(sorted_chunks):
1796:             if not hasattr(chunk, "policy_area_id") or not chunk.policy_area_id:
1797:                 raise CPPAdapterError(f"Missing policy_area_id in chunk {chunk.id}")
1798:             if not hasattr(chunk, "dimension_id") or not chunk.dimension_id:
1799:                 raise CPPAdapterError(f"Missing dimension_id in chunk {chunk.id}")
1800:             if not hasattr(chunk, "chunk_type") or not chunk.chunk_type:
1801:                 raise CPPAdapterError(f"Missing chunk_type in chunk {chunk.id}")
1802:
1803:         self.logger.info(f"Processing {len(sorted_chunks)} chunks")
1804:
1805:         # Build full text by concatenating chunks
1806:         full_text_parts: list[str] = []
1807:         sentences: list[dict[str, Any]] = []
1808:         sentence_metadata: list[dict[str, Any]] = []
1809:         tables: list[dict[str, Any]] = []
1810:         chunk_index: dict[str, int] = {}
1811:         chunk_summaries: list[dict[str, Any]] = []
1812:         chunks_data: list[ChunkData] = []
1813:
1814:         # Track indices for building indexes
1815:         term_index: dict[str, list[int]] = {}
1816:         numeric_index: dict[str, list[int]] = {}
1817:         temporal_index: dict[str, list[int]] = {}
1818:         entity_index: dict[str, list[int]] = {}
1819:
1820:         # Track running offset that matches how full_text is built
1821:         current_offset = 0
1822:
1823:         provenance_with_data = 0
1824:
1825:         for idx, chunk in enumerate(sorted_chunks):
1826:             chunk_text = chunk.text
1827:             chunk_start = current_offset
1828:             chunk_index[chunk.id] = idx
1829:
1830:             # Add to full text
1831:             full_text_parts.append(chunk_text)
1832:
1833:             # Create sentence entry (each chunk is represented as a sentence for orchestrator compatibility)
1834:             sentences.append(
1835:                 {
1836:                     "text": chunk_text,
1837:                     "chunk_id": chunk.id,
1838:                     "resolution": (
1839:                         chunk.resolution.value.lower()
1840:                         if hasattr(chunk, "resolution")
1841:                         else None
1842:                     ),
1843:                 }
1844:             )
1845:
1846:             # Create chunk metadata for per-sentence tracking
1847:             chunk_end = chunk_start + len(chunk_text)
1848:
```

```
            # CRITICAL: Preserve PAÃ\227DIM 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 "SPC-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:
```

```
2129:             qm = canon_package.quality_metrics
2130:             metadata_dict["quality_metrics"] = {
2131:                 "provenance_completeness": (
2132:                     qm.provenance_completeness
2133:                     if hasattr(qm, "provenance_completeness")
2134:                     else ParameterLoaderV2.get(
2135:                         "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
2136:                         "auto_param_L328_117",
2137:                         0.0,
2138:                     )
2139:                 ),
2140:                 "structural_consistency": (
2141:                     qm.structural_consistency
2142:                     if hasattr(qm, "structural_consistency")
2143:                     else ParameterLoaderV2.get(
2144:                         "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
2145:                         "auto_param_L329_114",
2146:                         0.0,
2147:                     )
2148:                 ),
2149:                 "boundary_f1": (
2150:                     qm.boundary_f1
2151:                     if hasattr(qm, "boundary_f1")
2152:                     else ParameterLoaderV2.get(
2153:                         "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
2154:                         "auto_param_L330_81",
2155:                         0.0,
2156:                     )
2157:                 ),
2158:                 "kpi_linkage_rate": (
2159:                     qm.kpi_linkage_rate
2160:                     if hasattr(qm, "kpi_linkage_rate")
2161:                     else ParameterLoaderV2.get(
2162:                         "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
2163:                         "auto_param_L331_96",
2164:                         0.0,
2165:                     )
2166:                 ),
2167:                 "budget_consistency_score": (
2168:                     qm.budget_consistency_score
2169:                     if hasattr(qm, "budget_consistency_score")
2170:                     else ParameterLoaderV2.get(
2171:                         "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
2172:                         "auto_param_L332_120",
2173:                         0.0,
2174:                     )
2175:                 ),
2176:                 "temporal_robustness": (
2177:                     qm.temporal_robustness
2178:                     if hasattr(qm, "temporal_robustness")
2179:                     else ParameterLoaderV2.get(
2180:                         "farfan_core.utils.cpp_adapter.CPPAdapter.__init__",
2181:                         "auto_param_L333_105",
2182:                         0.0,
2183:                     )
2184:                 ),
```

```
                "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 SPC 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 â\206\222 Orchestrator contract
        if self.wiring_validator is not None:
            self.logger.info("Validating Adapter â\206\222 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("â\234\223 Adapter â\206\222 Orchestrator contract validation passed")
            except Exception as e:
                self.logger.error(
                    f"Adapter â\206\222 Orchestrator contract validation failed: {e}"
                )
                raise ValueError(
                    f"Runtime contract violation at Adapter â\206\222 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:
```

```
2297:         PreprocessedDocument for orchestrator
2298:
2299:     Raises:
2300:         CPPAdapterError: If conversion fails
2301:     """
2302:     adapter = CPPAdapter()
2303:     return adapter.to_preprocessed_document(canon_package, document_id)
2304:
2305:
2306: __all__ = [
2307:     "CPPAdapter",
2308:     "CPPAdapterError",
2309:     "adapt_cpp_to_orchestrator",
2310: ]
2311:
2312:
2313:
2314: ================================================================================
2315: FILE: src/farfan_pipeline/utils/determinism/__init__.py
2316: ================================================================================
2317:
2318: """Determinism utilities for reproducible runs."""
2319:
2320: from farfan_pipeline.utils.determinism.seeds import DeterministicContext, SeedFactory
2321:
2322: __all__ = [
2323:     "DeterministicContext",
2324:     "SeedFactory",
2325: ]
2326:
2327:
2328:
2329: ================================================================================
2330: FILE: src/farfan_pipeline/utils/determinism/seeds.py
2331: ================================================================================
2332:
2333: """Deterministic seed management for reproducible execution."""
2334:
2335: from __future__ import annotations
2336:
2337: import hashlib
2338: import os
2339: import random
2340: from dataclasses import dataclass
2341: from typing import TYPE_CHECKING
2342: from farfan_pipeline.core.calibration.decorators import calibrated_method
2343:
2344: if TYPE_CHECKING:
2345:     from collections.abc import Iterable
2346:
2347: try:
2348:     import numpy as np
2349:     NUMPY_AVAILABLE = True
2350: except ImportError:  # pragma: no cover - optional dependency
2351:     np = None  # type: ignore
2352:     NUMPY_AVAILABLE = False
```

```
2353:
2354: class SeedFactory:
2355:     """Factory that derives stable seeds from canonical metadata."""
2356:
2357:     DEFAULT_SALT = b"PDM_DETERMINISM_SALT_2025"
2358:
2359:     def __init__(self, salt: bytes | None = None) -> None:
2360:         self._salt = salt or self.DEFAULT_SALT
2361:
2362:     @calibrated_method("farfan_core.utils.determinism.seeds.SeedFactory.derive_seed")
2363:     def derive_seed(self, components: Iterable[str]) -> int:
2364:         """Derive a deterministic 32-bit seed from ordered components."""
2365:
2366:         material = "|".join(str(component) for component in components)
2367:         digest = hashlib.sha256(self._salt + material.encode("utf-8")).digest()
2368:         return int.from_bytes(digest[:4], byteorder="big")
2369:
2370:     @calibrated_method("farfan_core.utils.determinism.seeds.SeedFactory.derive_run_seed")
2371:     def derive_run_seed(self, questionnaire_hash: str, run_id: str) -> int:
2372:         """Derive run-wide seed based on questionnaire hash and run identifier."""
2373:
2374:         return self.derive_seed([questionnaire_hash, run_id])
2375:
2376:     @calibrated_method("farfan_core.utils.determinism.seeds.SeedFactory.configure_environment")
2377:     def configure_environment(self, seed: int) -> None:
2378:         """Configure deterministic state for Python and NumPy."""
2379:
2380:         os.environ["PYTHONHASHSEED"] = str(seed)
2381:         random.seed(seed)
2382:         if NUMPY_AVAILABLE and np is not None:
2383:             np.random.seed(seed)
2384:
2385: @dataclass
2386: class DeterministicContext:
2387:     """Deterministic execution context shared with all producers."""
2388:
2389:     questionnaire_hash: str
2390:     run_id: str
2391:     seed: int
2392:     numpy_rng: np.random.Generator | None = None
2393:
2394:     @calibrated_method("farfan_core.utils.determinism.seeds.DeterministicContext.apply")
2395:     def apply(self) -> None:
2396:         """Apply deterministic seeding across the runtime environment."""
2397:
2398:         os.environ["PYTHONHASHSEED"] = str(self.seed)
2399:         random.seed(self.seed)
2400:         if NUMPY_AVAILABLE and np is not None:
2401:             self.numpy_rng = np.random.default_rng(self.seed)
2402:
2403:     @classmethod
2404:     def from_factory(
2405:         cls,
2406:         factory: SeedFactory,
2407:         questionnaire_hash: str,
2408:         run_id: str
```

```
    ) -> DeterministicContext:
        seed = factory.derive_run_seed(questionnaire_hash, run_id)
        context = cls(questionnaire_hash=questionnaire_hash, run_id=run_id, seed=seed)
        context.apply()
        return context


================================================================================
FILE: src/farfan_pipeline/utils/determinism_helpers.py
================================================================================

"""
Determinism Helpers - Centralized Seeding and State Management
==============================================================

Provides centralized determinism enforcement for the entire pipeline:
- Stable seed derivation from policy_unit_id and correlation_id
- Context manager for scoped deterministic execution
- Controls random, numpy.random, and other stochastic libraries

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""

from __future__ import annotations

import json
import os
import random
from contextlib import contextmanager
from dataclasses import dataclass
from hashlib import sha256
from typing import TYPE_CHECKING, Any

import numpy as np
from farfan_pipeline.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    from collections.abc import Iterator


def _seed_from(*parts: Any) -> int:
    """
    Derive a 32-bit seed from arbitrary parts via SHA-256.

    Args:
        *parts: Components to hash (will be JSON-serialized)

    Returns:
        32-bit integer seed suitable for random/numpy

    Examples:
        >>> s1 = _seed_from("PU_123", "corr-1")
        >>> s2 = _seed_from("PU_123", "corr-1")
```

```
        >>> s1 == s2
        True
        >>> s3 = _seed_from("PU_123", "corr-2")
        >>> s1 != s3
        True
    """
    raw = json.dumps(parts, sort_keys=True, separators=(",", ":"), ensure_ascii=False)
    # 32-bit seed for numpy/py random
    return int(sha256(raw.encode("utf-8")).hexdigest()[:8], 16)


@dataclass(frozen=True)
class Seeds:
    """Container for seeds used in deterministic execution."""
    py: int
    np: int


@contextmanager
def deterministic(
    policy_unit_id: str | None = None,
    correlation_id: str | None = None
) -> Iterator[Seeds]:
    """
    Context manager for deterministic execution.

    Sets seeds for Python's random and NumPy's random based on
    policy_unit_id and correlation_id. Seeds are derived deterministically
    via SHA-256 hashing.

    Args:
        policy_unit_id: Policy unit identifier (default: env var or "default")
        correlation_id: Correlation identifier (default: env var or "run")

    Yields:
        Seeds object with py and np seed values

    Examples:
        >>> with deterministic("PU_123", "corr-1") as seeds:
        ...     v1 = random.random()
        ...     a1 = np.random.rand(3)
        >>> with deterministic("PU_123", "corr-1") as seeds:
        ...     v2 = random.random()
        ...     a2 = np.random.rand(3)
        >>> v1 == v2  # Deterministic
        True
        >>> np.array_equal(a1, a2)  # Deterministic
        True
    """
    base = policy_unit_id or os.getenv("POLICY_UNIT_ID", "default")
    salt = correlation_id or os.getenv("CORRELATION_ID", "run")
    s = _seed_from("fixed", base, salt)

    # Set seeds for both random modules
    random.seed(s)
    np.random.seed(s)
```

```
2521:
2522:     try:
2523:         yield Seeds(py=s, np=s)
2524:     finally:
2525:         # Keep deterministic state; caller may reseed per-phase if needed
2526:         pass
2527:
2528:
2529: def create_deterministic_rng(seed: int) -> np.random.Generator:
2530:     """
2531:     Create a deterministic NumPy random number generator.
2532:
2533:     Use this for local RNG that doesn't affect global state.
2534:
2535:     Args:
2536:         seed: Integer seed
2537:
2538:     Returns:
2539:         NumPy Generator instance
2540:
2541:     Examples:
2542:         >>> rng = create_deterministic_rng(42)
2543:         >>> v1 = rng.random()
2544:         >>> rng = create_deterministic_rng(42)
2545:         >>> v2 = rng.random()
2546:         >>> v1 == v2
2547:         True
2548:     """
2549:     return np.random.default_rng(seed)
2550:
2551:
2552: if __name__ == "__main__":
2553:     import doctest
2554:
2555:     # Run doctests
2556:     print("Running doctests...")
2557:     doctest.testmod(verbose=True)
2558:
2559:     # Integration tests
2560:     print("\n" + "="*60)
2561:     print("Determinism Integration Tests")
2562:     print("="*60)
2563:
2564:     print("\n1. Testing seed derivation:")
2565:     s1 = _seed_from("PU_123", "corr-1")
2566:     s2 = _seed_from("PU_123", "corr-1")
2567:     s3 = _seed_from("PU_123", "corr-2")
2568:     assert s1 == s2
2569:     assert s1 != s3
2570:     print(f"   â\234\223 Same inputs â\206\222 same seed: {s1}")
2571:     print(f"   â\234\223 Different inputs â\206\222 different seed: {s3}")
2572:
2573:     print("\n2. Testing deterministic context with random:")
2574:     with deterministic("PU_123", "corr-1") as seeds1:
2575:         a = random.random()
2576:         b = random.randint(0, 100)
```

```
    with deterministic("PU_123", "corr-1") as seeds2:
        c = random.random()
        d = random.randint(0, 100)
    assert a == c
    assert b == d
    print(f"   â\234\223 Python random is deterministic: {a:.6f}")
    print(f"   â\234\223 Python randint is deterministic: {b}")

    print("\n3. Testing deterministic context with numpy:")
    with deterministic("PU_123", "corr-1") as seeds:
        arr1 = np.random.rand(3).tolist()
    with deterministic("PU_123", "corr-1") as seeds:
        arr2 = np.random.rand(3).tolist()
    assert arr1 == arr2
    print(f"   â\234\223 NumPy random is deterministic: {arr1}")

    print("\n4. Testing local RNG generator:")
    rng1 = create_deterministic_rng(42)
    v1 = rng1.random()
    rng2 = create_deterministic_rng(42)
    v2 = rng2.random()
    assert v1 == v2
    print(f"   â\234\223 Local RNG is deterministic: {v1:.6f}")

    print("\n5. Testing different correlation IDs produce different results:")
    with deterministic("PU_123", "corr-A"):
        val_a = random.random()
    with deterministic("PU_123", "corr-B"):
        val_b = random.random()
    assert val_a != val_b
    print("   â\234\223 Different correlation â\206\222 different values")
    print(f"      corr-A: {val_a:.6f}")
    print(f"      corr-B: {val_b:.6f}")

    print("\n" + "="*60)
    print("Determinism doctest OK - All tests passed!")
    print("="*60)



================================================================================
FILE: src/farfan_pipeline/utils/deterministic_execution.py
================================================================================

"""
Deterministic Execution Utilities - Production Grade
=====================================================

Utilities for ensuring deterministic, reproducible execution across
the policy analysis pipeline.

Features:
- Deterministic random seed management
- UTC-only timestamp handling
- Structured execution logging
- Side-effect isolation
```

```
- Reproducible event ID generation

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""

import hashlib
import logging
import random
import time
import uuid
from collections.abc import Callable, Iterator
from contextlib import contextmanager
from datetime import datetime, timezone
from typing import Any

import numpy as np

from farfan_pipeline.utils.enhanced_contracts import StructuredLogger, utc_now_iso
from farfan_pipeline.core.calibration.decorators import calibrated_method

# ============================================================================
# DETERMINISTIC SEED MANAGEMENT
# ============================================================================

class DeterministicSeedManager:
    """
    Manages random seeds for deterministic execution.

    All stochastic operations must use seeds managed by this class to ensure
    reproducibility across runs.

    Examples:
        >>> manager = DeterministicSeedManager(base_seed=42)
        >>> with manager.scoped_seed("operation1"):
        ...     value = random.random()
        >>> # Seed is automatically restored after context
    """

    def __init__(self, base_seed: int = 42) -> None:
        """
        Initialize seed manager with base seed.

        Args:
            base_seed: Master seed for all derived seeds
        """
        self.base_seed = base_seed
        self._seed_counter = 0
        self._initialize_seeds(base_seed)

    @calibrated_method("farfan_core.utils.deterministic_execution.DeterministicSeedManager._initialize_seeds")
    def _initialize_seeds(self, seed: int) -> None:
        """Initialize all random number generators with deterministic seeds."""
        random.seed(seed)
        np.random.seed(seed)
```

```
        # For reproducibility, also set hash seed
        # Note: PYTHONHASHSEED should be set in environment for full determinism

    @calibrated_method("farfan_core.utils.deterministic_execution.DeterministicSeedManager.get_derived_seed")
    def get_derived_seed(self, operation_name: str) -> int:
        """
        Generate a deterministic seed for a specific operation.

        Args:
            operation_name: Unique name for the operation

        Returns:
            Deterministic integer seed derived from operation name and base seed

        Examples:
            >>> manager = DeterministicSeedManager(42)
            >>> seed1 = manager.get_derived_seed("test")
            >>> seed2 = manager.get_derived_seed("test")
            >>> seed1 == seed2  # Deterministic
            True
        """
        # Use cryptographic hash for stable seed derivation
        hash_input = f"{self.base_seed}:{operation_name}".encode()
        hash_digest = hashlib.sha256(hash_input).digest()
        # Convert first 4 bytes to int
        return int.from_bytes(hash_digest[:4], byteorder='big')

    @contextmanager
    @calibrated_method("farfan_core.utils.deterministic_execution.DeterministicSeedManager.scoped_seed")
    def scoped_seed(self, operation_name: str) -> Iterator[int]:
        """
        Context manager for scoped seed usage.

        Sets seeds for the operation, then restores original state.

        Args:
            operation_name: Unique name for the operation

        Yields:
            Derived seed for this operation

        Examples:
            >>> manager = DeterministicSeedManager(42)
            >>> with manager.scoped_seed("my_operation") as seed:
            ...     result = random.randint(0, 100)
        """
        # Save current state
        random_state = random.getstate()
        np_state = np.random.get_state()

        # Set new seed
        derived_seed = self.get_derived_seed(operation_name)
        self._initialize_seeds(derived_seed)

        try:
            yield derived_seed
```

```
2745:         finally:
2746:             # Restore state
2747:             random.setstate(random_state)
2748:             np.random.set_state(np_state)
2749:
2750:     @calibrated_method("farfan_core.utils.deterministic_execution.DeterministicSeedManager.get_event_id")
2751:     def get_event_id(self, operation_name: str, timestamp_utc: str | None = None) -> str:
2752:         """
2753:         Generate a reproducible event ID for an operation.
2754:
2755:         Args:
2756:             operation_name: Operation name
2757:             timestamp_utc: Optional UTC timestamp (ISO-8601); if None, uses current time
2758:
2759:         Returns:
2760:             Deterministic event ID based on operation and timestamp
2761:
2762:         Examples:
2763:             >>> manager = DeterministicSeedManager(42)
2764:             >>> event_id = manager.get_event_id("test", "2024-01-01T00:00:00Z")
2765:             >>> len(event_id)
2766:             64
2767:         """
2768:         ts = timestamp_utc or utc_now_iso()
2769:         hash_input = f"{self.base_seed}:{operation_name}:{ts}".encode()
2770:         return hashlib.sha256(hash_input).hexdigest()
2771:
2772:
2773: # ============================================================================
2774: # DETERMINISTIC EXECUTION WRAPPER
2775: # ============================================================================
2776:
2777: class DeterministicExecutor:
2778:     """
2779:     Wraps functions to ensure deterministic execution with observability.
2780:
2781:     Features:
2782:     - Automatic seed management
2783:     - Structured logging of execution
2784:     - Latency tracking
2785:     - Error handling with event IDs
2786:
2787:     Examples:
2788:         >>> executor = DeterministicExecutor(base_seed=42, logger_name="test")
2789:         >>> @executor.deterministic(operation_name="my_func")
2790:         ... def my_function(x: int) -> int:
2791:         ...     return x + random.randint(0, 10)
2792:     """
2793:
2794:     def __init__(
2795:         self,
2796:         base_seed: int = 42,
2797:         logger_name: str = "deterministic_executor",
2798:         enable_logging: bool = True
2799:     ) -> None:
2800:         """
```

```
        Initialize deterministic executor.

        Args:
            base_seed: Master seed for all operations
            logger_name: Logger name for structured logging
            enable_logging: Whether to enable structured logging
        """
        self.seed_manager = DeterministicSeedManager(base_seed)
        self.logger = StructuredLogger(logger_name) if enable_logging else None
        self.enable_logging = enable_logging

    def deterministic(
        self,
        operation_name: str,
        log_inputs: bool = False,
        log_outputs: bool = False
    ) -> Callable:
        """
        Decorator to make a function deterministic with logging.

        Args:
            operation_name: Unique name for this operation
            log_inputs: Whether to log input parameters
            log_outputs: Whether to log output values

        Returns:
            Decorated function with deterministic execution
        """
        def decorator(func: Callable) -> Callable:
            def wrapper(*args: Any, **kwargs: Any) -> Any:
                # Generate correlation and event IDs
                correlation_id = str(uuid.uuid4())
                event_id = self.seed_manager.get_event_id(operation_name)

                # Start timing
                start_time = time.perf_counter()

                # Execute with scoped seed
                try:
                    with self.seed_manager.scoped_seed(operation_name) as seed:
                        result = func(*args, **kwargs)

                        # Calculate latency
                        latency_ms = (time.perf_counter() - start_time) * 1000

                        # Log success
                        if self.enable_logging and self.logger:
                            log_data = {
                                "event_id": event_id,
                                "seed": seed,
                                "latency_ms": latency_ms,
                            }
                            if log_inputs:
                                log_data["inputs"] = str(args)[:100]  # Truncate for safety
                            if log_outputs:
                                log_data["outputs"] = str(result)[:100]
```

```
                                self.logger.log_execution(
                                    operation=operation_name,
                                    correlation_id=correlation_id,
                                    success=True,
                                    latency_ms=latency_ms,
                                    **log_data
                                )

                            return result

                    except Exception as e:
                        # Calculate latency even on error
                        latency_ms = (time.perf_counter() - start_time) * 1000

                        # Log error
                        if self.enable_logging and self.logger:
                            self.logger.log_execution(
                                operation=operation_name,
                                correlation_id=correlation_id,
                                success=False,
                                latency_ms=latency_ms,
                                event_id=event_id,
                                error=str(e)[:200]  # Truncate for safety
                            )

                        # Re-raise with event ID
                        raise RuntimeError(f"[{event_id}] {operation_name} failed: {e}") from e

                return wrapper
            return decorator


# ============================================================================
# UTC TIMESTAMP UTILITIES
# ============================================================================

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

    Returns:
        Current datetime in UTC timezone

    Examples:
        >>> dt = enforce_utc_now()
        >>> dt.tzinfo is not None
        True
    """
    return datetime.now(timezone.utc)


def parse_utc_timestamp(timestamp_str: str) -> datetime:
    """
    Parse ISO-8601 timestamp and enforce UTC.

```

```
    Args:
        timestamp_str: ISO-8601 timestamp string

    Returns:
        Parsed datetime in UTC

    Raises:
        ValueError: If timestamp is not UTC or invalid format

    Examples:
        >>> dt = parse_utc_timestamp("2024-01-01T00:00:00Z")
        >>> dt.year
        2024
    """
    dt = datetime.fromisoformat(timestamp_str.replace('Z', '+00:00'))

    # Enforce UTC
    if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
        raise ValueError(f"Timestamp must be UTC: {timestamp_str}")

    return dt


# ============================================================================
# SIDE-EFFECT ISOLATION
# ============================================================================

@contextmanager
def isolated_execution() -> Iterator[None]:
    """
    Context manager to isolate side effects during execution.

    Current isolation:
    - Prevents print statements (captured and logged as warning)
    - Future: file I/O restrictions, network restrictions

    Yields:
        None

    Examples:
        >>> with isolated_execution():
        ...     # Code here has controlled side effects
        ...     pass
    """
    # For now, minimal isolation - can be extended with more restrictions
    import io
    import sys

    # Capture stdout/stderr to detect violations
    old_stdout = sys.stdout
    old_stderr = sys.stderr
    stdout_capture = io.StringIO()
    stderr_capture = io.StringIO()

    try:
        sys.stdout = stdout_capture
```

```
2969:         sys.stderr = stderr_capture
2970:         yield
2971:     finally:
2972:         sys.stdout = old_stdout
2973:         sys.stderr = old_stderr
2974:
2975:         # Log any captured output as warning (side effect violation)
2976:         if stdout_capture.getvalue():
2977:             logging.warning(
2978:                 "Side effect detected: stdout captured during isolated execution: %s",
2979:                 stdout_capture.getvalue()[:200]
2980:             )
2981:         if stderr_capture.getvalue():
2982:             logging.warning(
2983:                 "Side effect detected: stderr captured during isolated execution: %s",
2984:                 stderr_capture.getvalue()[:200]
2985:             )
2986:
2987:
2988: # ==============================================================================
2989: # IN-SCRIPT TESTS
2990: # ==============================================================================
2991:
2992: if __name__ == "__main__":
2993:     import doctest
2994:
2995:     # Run doctests
2996:     print("Running doctests...")
2997:     doctest.testmod(verbose=True)
2998:
2999:     # Additional tests
3000:     print("\n" + "="*60)
3001:     print("Deterministic Execution Tests")
3002:     print("="*60)
3003:
3004:     # Test 1: Seed manager determinism
3005:     print("\n1. Testing seed manager determinism:")
3006:     manager1 = DeterministicSeedManager(42)
3007:     manager2 = DeterministicSeedManager(42)
3008:
3009:     seed1_a = manager1.get_derived_seed("test_op")
3010:     seed1_b = manager1.get_derived_seed("test_op")
3011:     seed2_a = manager2.get_derived_seed("test_op")
3012:
3013:     assert seed1_a == seed1_b == seed2_a, "Seeds must be deterministic"
3014:     print(f"   â\234\223 Deterministic seeds: {seed1_a} == {seed1_b} == {seed2_a}")
3015:
3016:     # Test 2: Scoped seed restoration
3017:     print("\n2. Testing scoped seed restoration:")
3018:     manager = DeterministicSeedManager(42)
3019:
3020:     initial_value = random.random()
3021:     with manager.scoped_seed("temp_operation"):
3022:         _ = random.random()  # Different value inside scope
3023:     restored_value = random.random()
3024:
```

```
    # Reset and check if we can reproduce
    manager._initialize_seeds(42)
    reproduced_value = random.random()

    print(f"   â\234\223 Initial value: {initial_value:.6f}")
    print(f"   â\234\223 Reproduced value: {reproduced_value:.6f}")
    assert abs(initial_value - reproduced_value) < 1e-10, "Seed restoration failed"
    print("   â\234\223 Seed restoration successful")

    # Test 3: Deterministic executor
    print("\n3. Testing deterministic executor:")
    executor = DeterministicExecutor(base_seed=42, enable_logging=False)

    @executor.deterministic(operation_name="test_function")
    def sample_function(n: int) -> float:
        return sum(random.random() for _ in range(n))

    result1 = sample_function(5)

    # Reset and run again
    executor.seed_manager._initialize_seeds(42)
    result2 = sample_function(5)

    print(f"   â\234\223 Result 1: {result1:.6f}")
    print(f"   â\234\223 Result 2: {result2:.6f}")
    assert abs(result1 - result2) < 1e-10, "Deterministic execution failed"
    print("   â\234\223 Deterministic execution verified")

    # Test 4: UTC enforcement
    print("\n4. Testing UTC enforcement:")
    utc_now = enforce_utc_now()
    print(f"   â\234\223 UTC now: {utc_now.isoformat()}")
    assert utc_now.tzinfo is not None, "Must have timezone"

    # Test 5: Event ID reproducibility
    print("\n5. Testing event ID reproducibility:")
    manager = DeterministicSeedManager(42)
    event_id1 = manager.get_event_id("operation", "2024-01-01T00:00:00Z")
    event_id2 = manager.get_event_id("operation", "2024-01-01T00:00:00Z")
    assert event_id1 == event_id2, "Event IDs must be reproducible"
    print(f"   â\234\223 Event ID: {event_id1[:16]}...")
    print("   â\234\223 Event ID reproducibility verified")

    print("\n" + "="*60)
    print("All tests passed!")
    print("="*60)


================================================================================
FILE: src/farfan_pipeline/utils/domain_errors.py
================================================================================

from farfan_pipeline.core.calibration.decorators import calibrated_method
"""
Domain-Specific Exceptions - Contract Violation Errors
```

```
3081: ==========================================================
3082:
3083: Provides domain-specific exception hierarchy for contract violations.
3084:
3085: Exception Hierarchy:
3086:     ContractViolationError (base)
3087:     â\224\234â\224\200â\224\200 DataContractError (data/payload violations)
3088:     â\224\224â\224\200â\224\200 SystemContractError (system/configuration violations)
3089:
3090: Author: Policy Analytics Research Unit
3091: Version: 1.0.0
3092: License: Proprietary
3093: """
3094:
3095:
3096: class ContractViolationError(Exception):
3097:     """
3098:     Base exception for all contract violations.
3099:
3100:     Use this as the base class for specific contract violation types.
3101:
3102:     Examples:
3103:         >>> try:
3104:         ...     raise ContractViolationError("Contract violated")
3105:         ... except ContractViolationError as e:
3106:         ...     print(f"Caught: {e}")
3107:         Caught: Contract violated
3108:     """
3109:     pass
3110:
3111:
3112: class DataContractError(ContractViolationError):
3113:     """
3114:     Exception for data/payload contract violations.
3115:
3116:     Raised when:
3117:     - Payload schema is invalid
3118:     - Required fields are missing
3119:     - Field values are out of range
3120:     - Data integrity checks fail (e.g., digest mismatch)
3121:
3122:     Examples:
3123:         >>> try:
3124:         ...     raise DataContractError("Invalid payload schema")
3125:         ... except DataContractError as e:
3126:         ...     print(f"Data error: {e}")
3127:         Data error: Invalid payload schema
3128:     """
3129:     pass
3130:
3131:
3132: class SystemContractError(ContractViolationError):
3133:     """
3134:     Exception for system/configuration contract violations.
3135:
3136:     Raised when:
```

```
    - System configuration is invalid
    - Required resources are unavailable
    - Environment preconditions are not met
    - Infrastructure failures occur

    Examples:
        >>> try:
        ...     raise SystemContractError("Configuration missing")
        ... except SystemContractError as e:
        ...     print(f"System error: {e}")
        System error: Configuration missing
    """
    pass


if __name__ == "__main__":
    import doctest

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Integration tests
    print("\n" + "="*60)
    print("Domain Exceptions Integration Tests")
    print("="*60)

    print("\n1. Testing exception hierarchy:")
    assert issubclass(DataContractError, ContractViolationError)
    assert issubclass(SystemContractError, ContractViolationError)
    print("   â\234\223 DataContractError inherits from ContractViolationError")
    print("   â\234\223 SystemContractError inherits from ContractViolationError")

    print("\n2. Testing exception catching:")
    try:
        raise DataContractError("Test data error")
    except ContractViolationError as e:
        assert isinstance(e, DataContractError)
        print("   â\234\223 DataContractError caught as ContractViolationError")

    try:
        raise SystemContractError("Test system error")
    except ContractViolationError as e:
        assert isinstance(e, SystemContractError)
        print("   â\234\223 SystemContractError caught as ContractViolationError")

    print("\n3. Testing specific exception catching:")
    try:
        raise DataContractError("Payload validation failed")
    except DataContractError as e:
        assert str(e) == "Payload validation failed"
        print("   â\234\223 DataContractError caught specifically")

    try:
        raise SystemContractError("Config file not found")
    except SystemContractError as e:
```

```
        assert str(e) == "Config file not found"
        print("   â\234\223 SystemContractError caught specifically")

    print("\n4. Testing error differentiation:")
    errors = []

    try:
        raise DataContractError("Data issue")
    except ContractViolationError as e:
        errors.append(("data", type(e).__name__))

    try:
        raise SystemContractError("System issue")
    except ContractViolationError as e:
        errors.append(("system", type(e).__name__))

    assert errors[0] == ("data", "DataContractError")
    assert errors[1] == ("system", "SystemContractError")
    print("   â\234\223 Data and system errors are distinguishable")

    print("\n" + "="*60)
    print("Domain exceptions doctest OK - All tests passed!")
    print("="*60)



================================================================================
FILE: src/farfan_pipeline/utils/enhanced_contracts.py
================================================================================

"""
Enhanced Contract System with Pydantic - Production Grade
==========================================================

Strict contract definitions with cryptographic verification, deterministic
execution guarantees, and comprehensive validation.

Features:
- Static typing with Pydantic BaseModel
- Schema versioning for backward compatibility
- Cryptographic content digests (SHA-256)
- UTC timestamps (ISO-8601)
- Domain-specific exceptions
- Structured JSON logging
- Flow compatibility validation

Author: Policy Analytics Research Unit
Version: 2.0.0
License: Proprietary
"""

from __future__ import annotations

import hashlib
import json
import logging
```

```
import uuid
from datetime import datetime, timezone
from typing import Any

from pydantic import BaseModel, ConfigDict, Field, field_validator
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

# ==============================================================================
# DOMAIN-SPECIFIC EXCEPTIONS
# ==============================================================================

class ContractValidationError(Exception):
    """Raised when contract validation fails."""

    def __init__(self, message: str, field: str | None = None, event_id: str | None = None) -> None:
        self.field = field
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


class DataIntegrityError(Exception):
    """Raised when data integrity checks fail (e.g., hash mismatch)."""

    def __init__(self, message: str, expected: str | None = None, got: str | None = None, event_id: str | None = None) -> None:
        self.expected = expected
        self.got = got
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


class SystemConfigError(Exception):
    """Raised when system configuration is invalid."""

    def __init__(self, message: str, config_key: str | None = None, event_id: str | None = None) -> None:
        self.config_key = config_key
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


class FlowCompatibilityError(Exception):
    """Raised when data flow between components is incompatible."""

    def __init__(self, message: str, producer: str | None = None, consumer: str | None = None, event_id: str | None = None) -> None:
        self.producer = producer
        self.consumer = consumer
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


# ==============================================================================
# UTILITY FUNCTIONS FOR DETERMINISM AND VALIDATION
# ==============================================================================

def compute_content_digest(content: str | bytes | dict[str, Any]) -> str:
    """
```

```
    Compute SHA-256 digest of content in a deterministic way.

    Args:
        content: String, bytes, or dict to hash

    Returns:
        Hexadecimal SHA-256 digest

    Examples:
        >>> digest = compute_content_digest("test")
        >>> len(digest)
        64
        >>> digest == compute_content_digest("test")  # Deterministic
        True
    """
    if isinstance(content, dict):
        # Sort keys for deterministic JSON
        content_str = json.dumps(content, sort_keys=True, ensure_ascii=True)
        content_bytes = content_str.encode('utf-8')
    elif isinstance(content, str):
        content_bytes = content.encode('utf-8')
    elif isinstance(content, bytes):
        content_bytes = content
    else:
        raise ContractValidationError(
            f"Cannot compute digest for type {type(content).__name__}",
            field="content"
        )

    return hashlib.sha256(content_bytes).hexdigest()


def utc_now_iso() -> str:
    """
    Get current UTC timestamp in ISO-8601 format.

    Returns:
        ISO-8601 timestamp string (UTC timezone)

    Examples:
        >>> ts = utc_now_iso()
        >>> 'T' in ts and 'Z' in ts
        True
    """
    return datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')


# ============================================================================
# BASE CONTRACT MODEL
# ============================================================================

class BaseContract(BaseModel):
    """
    Base contract model with common fields for all contracts.

    All contracts must include:
```

```
    - schema_version: Semantic version for contract evolution
    - timestamp_utc: ISO-8601 UTC timestamp
    - correlation_id: UUID for request tracing
    """

    model_config = ConfigDict(
        frozen=True,  # Immutable for safety
        extra='forbid',  # Reject unknown fields
        validate_assignment=True,
        str_strip_whitespace=True,
    )

    schema_version: str = Field(
        default="2.0.0",
        description="Contract schema version (semantic versioning)",
        pattern=r"^\d+\.\d+\.\d+$"
    )

    timestamp_utc: str = Field(
        default_factory=utc_now_iso,
        description="UTC timestamp in ISO-8601 format"
    )

    correlation_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="UUID for request correlation and tracing"
    )

    @field_validator('timestamp_utc')
    @classmethod
    def validate_timestamp(cls, v: str) -> str:
        """Validate timestamp is ISO-8601 format and UTC."""
        try:
            dt = datetime.fromisoformat(v.replace('Z', '+00:00'))
            # Ensure UTC
            if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
                raise ValueError("Timestamp must be UTC")
            return v
        except (ValueError, AttributeError) as e:
            raise ContractValidationError(
                f"Invalid ISO-8601 timestamp: {v}",
                field="timestamp_utc"
            ) from e


# ============================================================================
# DOCUMENT CONTRACTS - V2
# ============================================================================

class DocumentMetadataV2(BaseContract):
    """
    Enhanced document metadata with cryptographic verification.

    Attributes:
        file_path: Absolute path to document
        file_name: Document filename
```

```
3417:         num_pages: Number of pages
3418:         file_size_bytes: File size in bytes
3419:         content_digest: SHA-256 hash of file content
3420:         policy_unit_id: Unique identifier for policy unit
3421:         encoding: Character encoding (default: utf-8)
3422:     """
3423:
3424:     file_path: str = Field(..., description="Absolute path to document")
3425:     file_name: str = Field(..., description="Document filename", min_length=1)
3426:     num_pages: int = Field(..., description="Number of pages", ge=1)
3427:     file_size_bytes: int = Field(..., description="File size in bytes", ge=0)
3428:     content_digest: str = Field(..., description="SHA-256 hash of content", pattern=r"^[a-f0-9]{64}$")
3429:     policy_unit_id: str = Field(..., description="Unique policy unit identifier")
3430:     encoding: str = Field(default="utf-8", description="Character encoding")
3431:
3432:     # Optional metadata
3433:     pdf_metadata: dict[str, Any] | None = Field(default=None, description="PDF metadata dictionary")
3434:     author: str | None = Field(default=None, description="Document author")
3435:     title: str | None = Field(default=None, description="Document title")
3436:     creation_date: str | None = Field(default=None, description="Document creation date")
3437:
3438:
3439: class ProcessedTextV2(BaseContract):
3440:     """
3441:     Enhanced processed text with input/output validation.
3442:
3443:     Attributes:
3444:         raw_text: Original unprocessed text
3445:         normalized_text: Normalized/cleaned text
3446:         language: Detected language code
3447:         input_digest: SHA-256 of raw_text input
3448:         output_digest: SHA-256 of normalized_text output
3449:         policy_unit_id: Policy unit identifier
3450:         processing_latency_ms: Processing time in milliseconds
3451:     """
3452:
3453:     raw_text: str = Field(..., description="Original unprocessed text", min_length=1)
3454:     normalized_text: str = Field(..., description="Normalized/cleaned text", min_length=1)
3455:     language: str = Field(..., description="ISO 639-1 language code", pattern=r"^[a-z]{2}$")
3456:     input_digest: str = Field(..., description="SHA-256 of raw_text", pattern=r"^[a-f0-9]{64}$")
3457:     output_digest: str = Field(..., description="SHA-256 of normalized_text", pattern=r"^[a-f0-9]{64}$")
3458:     policy_unit_id: str = Field(..., description="Policy unit identifier")
3459:     processing_latency_ms: float = Field(..., description="Processing latency in ms", ge=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCom
patibilityError.__init__", "auto_param_L236_89", 0.0))
3460:
3461:     # Optional fields
3462:     sentences: list[str] | None = Field(default=None, description="Sentence segmentation")
3463:     sections: list[dict[str, Any]] | None = Field(default=None, description="Document sections")
3464:     payload_size_bytes: int | None = Field(default=None, description="Payload size", ge=0)
3465:
3466:     @field_validator('input_digest')
3467:     @classmethod
3468:     def validate_input_digest(cls, v: str, info) -> str:
3469:         """Verify input digest matches raw_text if available."""
3470:         # This is validated post-construction
3471:         return v
```

```
3472:
3473:
3474: # ========================================================================
3475: # ANALYSIS CONTRACTS - V2
3476: # ========================================================================
3477:
3478: class AnalysisInputV2(BaseContract):
3479:     """
3480:     Enhanced analysis input with cryptographic verification.
3481:
3482:     Attributes:
3483:         text: Input text to analyze
3484:         document_id: Unique document identifier
3485:         policy_unit_id: Policy unit identifier
3486:         input_digest: SHA-256 of input text
3487:         payload_size_bytes: Size of input payload
3488:     """
3489:
3490:     text: str = Field(..., description="Input text to analyze", min_length=1)
3491:     document_id: str = Field(..., description="Unique document identifier")
3492:     policy_unit_id: str = Field(..., description="Policy unit identifier")
3493:     input_digest: str = Field(
3494:         ...,
3495:         description="SHA-256 hash of input text",
3496:         pattern=r"^[a-f0-9]{64}$"
3497:     )
3498:     payload_size_bytes: int = Field(..., description="Payload size in bytes", ge=0)
3499:
3500:     # Optional context
3501:     metadata: dict[str, Any] | None = Field(default=None, description="Additional metadata")
3502:     context: dict[str, Any] | None = Field(default=None, description="Execution context")
3503:     sentences: list[str] | None = Field(default=None, description="Pre-segmented sentences")
3504:
3505:     @classmethod
3506:     def create_from_text(
3507:         cls,
3508:         text: str,
3509:         document_id: str,
3510:         policy_unit_id: str,
3511:         **kwargs: Any
3512:     ) -> AnalysisInputV2:
3513:         """
3514:         Factory method to create AnalysisInputV2 with auto-computed digest.
3515:
3516:         Args:
3517:             text: Input text
3518:             document_id: Document ID
3519:             policy_unit_id: Policy unit ID
3520:             **kwargs: Additional optional fields
3521:
3522:         Returns:
3523:             Validated AnalysisInputV2 instance
3524:         """
3525:         input_digest = compute_content_digest(text)
3526:         payload_size_bytes = len(text.encode('utf-8'))
3527:
```

```
3528:         return cls(
3529:             text=text,
3530:             document_id=document_id,
3531:             policy_unit_id=policy_unit_id,
3532:             input_digest=input_digest,
3533:             payload_size_bytes=payload_size_bytes,
3534:             **kwargs
3535:         )
3536:
3537:
3538: class AnalysisOutputV2(BaseContract):
3539:     """
3540:     Enhanced analysis output with confidence bounds and validation.
3541:
3542:     Attributes:
3543:         dimension: Analysis dimension
3544:         category: Result category
3545:         confidence: Confidence score [ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__init__", "auto_param_L322_38", 0.
0), ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__init__", "auto_param_L322_43", 1.0)]
3546:         matches: Evidence matches
3547:         output_digest: SHA-256 of output content
3548:         policy_unit_id: Policy unit identifier
3549:         processing_latency_ms: Processing time in milliseconds
3550:     """
3551:
3552:     dimension: str = Field(..., description="Analysis dimension", min_length=1)
3553:     category: str = Field(..., description="Result category", min_length=1)
3554:     confidence: float = Field(..., description="Confidence score", ge=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__i
nit__", "auto_param_L331_70", 0.0), le=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__init__", "auto_param_L331_78", 1.0))
3555:     matches: list[str] = Field(..., description="Evidence matches")
3556:     output_digest: str = Field(..., description="SHA-256 of output", pattern=r"^[a-f0-9]{64}$")
3557:     policy_unit_id: str = Field(..., description="Policy unit identifier")
3558:     processing_latency_ms: float = Field(..., description="Processing latency in ms", ge=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCom
patibilityError.__init__", "auto_param_L335_89", 0.0))
3559:
3560:     # Optional fields
3561:     positions: list[int] | None = Field(default=None, description="Match positions")
3562:     evidence: list[str] | None = Field(default=None, description="Supporting evidence")
3563:     warnings: list[str] | None = Field(default=None, description="Validation warnings")
3564:     payload_size_bytes: int | None = Field(default=None, description="Output payload size", ge=0)
3565:
3566:     @field_validator('confidence')
3567:     @classmethod
3568:     def validate_confidence_numerical_stability(cls, v: float) -> float:
3569:         """Ensure confidence is numerically stable and within bounds."""
3570:         if not (0.0 <= v <= 1.0):
3571:             raise ContractValidationError(
3572:                 f"Confidence must be in [0.0, 1.0], got {v}",
3573:                 field="confidence"
3574:             )
3575:         # Round to avoid floating point precision issues
3576:         return round(v, 6)
3577:
3578:
3579: # ============================================================================
3580: # EXECUTION CONTRACTS - V2
```

```
# ============================================================================

class ExecutionContextV2(BaseContract):
    """
    Enhanced execution context with full observability.

    Attributes:
        class_name: Executor class name
        method_name: Method being executed
        document_id: Document identifier
        policy_unit_id: Policy unit identifier
        execution_id: Unique execution identifier
        parent_correlation_id: Parent request correlation ID
    """

    class_name: str = Field(..., description="Executor class name", min_length=1)
    method_name: str = Field(..., description="Method being executed", min_length=1)
    document_id: str = Field(..., description="Document identifier")
    policy_unit_id: str = Field(..., description="Policy unit identifier")
    execution_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="Unique execution identifier"
    )
    parent_correlation_id: str | None = Field(
        default=None,
        description="Parent correlation ID for nested calls"
    )

    # Optional context
    raw_text: str | None = Field(default=None, description="Raw input text")
    text: str | None = Field(default=None, description="Processed text")
    metadata: dict[str, Any] | None = Field(default=None, description="Metadata")
    tables: dict[str, Any] | None = Field(default=None, description="Extracted tables")
    sentences: list[str] | None = Field(default=None, description="Sentences")


# ============================================================================
# STRUCTURED LOGGING HELPER
# ============================================================================

class StructuredLogger:
    """
    Structured JSON logger for observability.

    Logs include:
    - correlation_id for tracing
    - latencies per operation
    - payload sizes
    - cryptographic fingerprints
    - NO PII
    """

    def __init__(self, name: str) -> None:
        """Initialize logger with name."""
        self.logger = logging.getLogger(name)
        self.logger.setLevel(logging.INFO)
```

```
3637:
3638:     def log_contract_validation(
3639:         self,
3640:         contract_type: str,
3641:         correlation_id: str,
3642:         success: bool,
3643:         latency_ms: float,
3644:         payload_size_bytes: int = 0,
3645:         content_digest: str | None = None,
3646:         error: str | None = None
3647:     ) -> None:
3648:         """Log contract validation event."""
3649:         log_entry = {
3650:             "event": "contract_validation",
3651:             "contract_type": contract_type,
3652:             "correlation_id": correlation_id,
3653:             "success": success,
3654:             "latency_ms": round(latency_ms, 3),
3655:             "payload_size_bytes": payload_size_bytes,
3656:             "timestamp_utc": utc_now_iso(),
3657:         }
3658:
3659:         if content_digest:
3660:             log_entry["content_digest"] = content_digest
3661:
3662:         if error:
3663:             log_entry["error"] = error
3664:
3665:         self.logger.info(json.dumps(log_entry, sort_keys=True))
3666:
3667:     def log_execution(
3668:         self,
3669:         operation: str,
3670:         correlation_id: str,
3671:         success: bool,
3672:         latency_ms: float,
3673:         **kwargs: Any
3674:     ) -> None:
3675:         """Log execution event with additional context."""
3676:         log_entry = {
3677:             "event": "execution",
3678:             "operation": operation,
3679:             "correlation_id": correlation_id,
3680:             "success": success,
3681:             "latency_ms": round(latency_ms, 3),
3682:             "timestamp_utc": utc_now_iso(),
3683:         }
3684:         log_entry.update(kwargs)
3685:
3686:         self.logger.info(json.dumps(log_entry, sort_keys=True))
3687:
3688:
3689: # ==============================================================================
3690: # IN-SCRIPT TESTS
3691: # ==============================================================================
3692:
```

```
3693: if __name__ == "__main__":
3694:     import doctest
3695:
3696:     # Run doctests
3697:     print("Running doctests...")
3698:     doctest.testmod(verbose=True)
3699:
3700:     # Contract validation examples
3701:     print("\n" + "="*60)
3702:     print("Contract Validation Examples")
3703:     print("="*60)
3704:
3705:     # Example 1: Document metadata
3706:     print("\n1. DocumentMetadataV2 validation:")
3707:     doc_meta = DocumentMetadataV2(
3708:         file_path="/path/to/document.pdf",
3709:         file_name="document.pdf",
3710:         num_pages=10,
3711:         file_size_bytes=1024000,
3712:         content_digest="a" * 64,  # Valid SHA-256 hex
3713:         policy_unit_id="PDM-001"
3714:     )
3715:     print(f"   â\234\223 Valid: correlation_id={doc_meta.correlation_id[:8]}...")
3716:
3717:     # Example 2: Analysis input with auto-digest
3718:     print("\n2. AnalysisInputV2 with auto-computed digest:")
3719:     analysis_input = AnalysisInputV2.create_from_text(
3720:         text="Sample policy text for analysis",
3721:         document_id="DOC-123",
3722:         policy_unit_id="PDM-001"
3723:     )
3724:     print(f"   â\234\223 Valid: input_digest={analysis_input.input_digest[:16]}...")
3725:     print(f"   â\234\223 Payload size: {analysis_input.payload_size_bytes} bytes")
3726:
3727:     # Example 3: Analysis output with confidence validation
3728:     print("\n3. AnalysisOutputV2 with confidence bounds:")
3729:     analysis_output = AnalysisOutputV2(
3730:         dimension="Dimension1",
3731:         category="CategoryA",
3732:         confidence=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.StructuredLogger.__init__", "auto_param_L509_19", 0.85),  # Must be in [Param
eterLoaderV2.get("farfan_core.utils.enhanced_contracts.StructuredLogger.__init__", "auto_param_L509_40", 0.0), ParameterLoaderV2.get("farfan_core.utils.enhanced_co
ntracts.StructuredLogger.__init__", "auto_param_L509_45", 1.0)]
3733:         matches=["evidence1", "evidence2"],
3734:         output_digest="b" * 64,
3735:         policy_unit_id="PDM-001",
3736:         processing_latency_ms=123.456
3737:     )
3738:     print(f"   â\234\223 Valid: confidence={analysis_output.confidence}")
3739:
3740:     # Example 4: Structured logging
3741:     print("\n4. Structured logging example:")
3742:     logger = StructuredLogger("test_logger")
3743:     logger.log_contract_validation(
3744:         contract_type="AnalysisInputV2",
3745:         correlation_id=analysis_input.correlation_id,
3746:         success=True,
```

```
3747:         latency_ms=5.2,
3748:         payload_size_bytes=analysis_input.payload_size_bytes,
3749:         content_digest=analysis_input.input_digest
3750:     )
3751:     print("   â\234\223 JSON log emitted to logger")
3752:
3753:     # Example 5: Exception handling
3754:     print("\n5. Domain-specific exceptions:")
3755:     try:
3756:         raise ContractValidationError("Invalid field", field="test_field")
3757:     except ContractValidationError as e:
3758:         print(f"   â\234\223 ContractValidationError: {e}")
3759:         print(f"   â\234\223 Event ID: {e.event_id}")
3760:
3761:     print("\n" + "="*60)
3762:     print("All validation examples passed!")
3763:     print("="*60)
3764:
3765:
3766:
3767: ================================================================================
3768: FILE: src/farfan_pipeline/utils/evidence_registry.py
3769: ================================================================================
3770:
3771: """Append-only evidence registry with cryptographic hashing.
3772:
3773: This module implements a small ledger that stores evidence entries produced by
3774: analysis components. Each entry links to the previous one through a SHA-256
3775: hash, producing an immutable chain that can be verified for tampering.
3776: """
3777: from __future__ import annotations
3778:
3779: import hashlib
3780: import json
3781: from dataclasses import asdict, dataclass
3782: from datetime import datetime
3783: from pathlib import Path
3784: from typing import TYPE_CHECKING, Any
3785: from farfan_pipeline.core.calibration.decorators import calibrated_method
3786:
3787: if TYPE_CHECKING:
3788:     from collections.abc import Iterable
3789:
3790: def _canonical_json(payload: dict[str, Any]) -> str:
3791:     """Return a canonical JSON representation with sorted keys."""
3792:     return json.dumps(payload, ensure_ascii=False, sort_keys=True, separators=(",", ":"))
3793:
3794: @dataclass(frozen=True)
3795: class EvidenceRecord:
3796:     """Single append-only evidence entry."""
3797:
3798:     index: int
3799:     timestamp: str
3800:     method_name: str
3801:     evidence: list[str]
3802:     metadata: dict[str, Any]
```

```
    previous_hash: str
    entry_hash: str

    @staticmethod
    def create(
        index: int,
        method_name: str,
        evidence: Iterable[str],
        metadata: dict[str, Any] | None,
        previous_hash: str,
        timestamp: datetime | None = None,
    ) -> EvidenceRecord:
        """Build a new evidence record and compute its hash."""
        ts = (timestamp or datetime.utcnow()).isoformat() + "Z"
        metadata_dict = dict(metadata or {})
        evidence_list = list(evidence)

        payload = {
            "index": index,
            "timestamp": ts,
            "method_name": method_name,
            "evidence": evidence_list,
            "metadata": metadata_dict,
            "previous_hash": previous_hash,
        }
        digest = hashlib.sha256(_canonical_json(payload).encode("utf-8")).hexdigest()
        return EvidenceRecord(
            index=index,
            timestamp=ts,
            method_name=method_name,
            evidence=evidence_list,
            metadata=metadata_dict,
            previous_hash=previous_hash,
            entry_hash=digest,
        )

class EvidenceRegistry:
    """Append-only registry that persists evidence records to disk."""

    def __init__(self, storage_path: Path | None = None, auto_load: bool = True) -> None:
        self.storage_path = storage_path or Path(".evidence_registry.json")
        self._records: list[EvidenceRecord] = []
        if auto_load and self.storage_path.exists():
            self._records = self._load_records(self.storage_path)

    @property
    @calibrated_method("farfan_core.utils.evidence_registry.EvidenceRegistry.records")
    def records(self) -> tuple[EvidenceRecord, ...]:
        """Expose records as an immutable tuple."""
        return tuple(self._records)

    def append(
        self,
        method_name: str,
        evidence: Iterable[str],
        metadata: dict[str, Any] | None = None,
```

```
3859:         monolith_hash: str | None = None,
3860:     ) -> EvidenceRecord:
3861:         """Append a new evidence record to the registry.
3862:
3863:         Args:
3864:             method_name: Name of the method producing evidence
3865:             evidence: Evidence strings
3866:             metadata: Additional metadata dictionary
3867:             monolith_hash: SHA-256 hash of questionnaire_monolith.json (recommended)
3868:
3869:         ARCHITECTURAL NOTE: Including monolith_hash ensures evidence is
3870:         traceable to the specific questionnaire version that generated it.
3871:         Use factory.compute_monolith_hash() to generate this value.
3872:         """
3873:         previous_hash = self._records[-1].entry_hash if self._records else "GENESIS"
3874:
3875:         # Merge monolith_hash into metadata if provided
3876:         enriched_metadata = dict(metadata or {})
3877:         if monolith_hash is not None:
3878:             enriched_metadata['monolith_hash'] = monolith_hash
3879:
3880:         record = EvidenceRecord.create(
3881:             index=len(self._records),
3882:             method_name=method_name,
3883:             evidence=evidence,
3884:             metadata=enriched_metadata,
3885:             previous_hash=previous_hash,
3886:         )
3887:         self._records.append(record)
3888:         return record
3889:
3890:     # ------------------------------------------------------------------
3891:     # Persistence
3892:     # ------------------------------------------------------------------
3893:     @calibrated_method("farfan_core.utils.evidence_registry.EvidenceRegistry.save")
3894:     def save(self) -> None:
3895:         """Persist the registry to disk."""
3896:         payload = [_serialize_record(record) for record in self._records]
3897:         self.storage_path.write_text(
3898:             json.dumps(payload, indent=2, ensure_ascii=False),
3899:             encoding="utf-8",
3900:         )
3901:
3902:     @calibrated_method("farfan_core.utils.evidence_registry.EvidenceRegistry._load_records")
3903:     def _load_records(self, path: Path) -> list[EvidenceRecord]:
3904:         try:
3905:             data = json.loads(path.read_text(encoding="utf-8"))
3906:         except json.JSONDecodeError as exc:
3907:             raise ValueError(f"Evidence registry at {path} is not valid JSON: {exc}") from exc
3908:
3909:         if not isinstance(data, list):
3910:             raise ValueError("Evidence registry payload must be a list")
3911:
3912:         records: list[EvidenceRecord] = []
3913:         for index, raw in enumerate(data):
3914:             if not isinstance(raw, dict):
```

```
3915:                 raise ValueError("Evidence record must be a JSON object")
3916:             expected_index = raw.get("index")
3917:             if expected_index != index:
3918:                 raise ValueError(
3919:                     f"Evidence record index mismatch at position {index}: found {expected_index}"
3920:                 )
3921:             record = EvidenceRecord(
3922:                 index=index,
3923:                 timestamp=str(raw.get("timestamp")),
3924:                 method_name=str(raw.get("method_name")),
3925:                 evidence=list(raw.get("evidence", [])),
3926:                 metadata=dict(raw.get("metadata", {})),
3927:                 previous_hash=str(raw.get("previous_hash")),
3928:                 entry_hash=str(raw.get("entry_hash")),
3929:             )
3930:             records.append(record)
3931:
3932:         self._assert_chain(records)
3933:         return records
3934:
3935:     # ------------------------------------------------------------------
3936:     # Verification utilities
3937:     # ------------------------------------------------------------------
3938:     @calibrated_method("farfan_core.utils.evidence_registry.EvidenceRegistry.verify")
3939:     def verify(self) -> bool:
3940:         """Verify registry integrity by recomputing all hashes."""
3941:         self._assert_chain(self._records)
3942:         return True
3943:
3944:     @staticmethod
3945:     def _assert_chain(records: list[EvidenceRecord]) -> None:
3946:         previous_hash = "GENESIS"
3947:         for expected_index, record in enumerate(records):
3948:             if record.index != expected_index:
3949:                 raise ValueError(
3950:                     f"Evidence record out of order: expected index {expected_index}, got {record.index}"
3951:                 )
3952:             payload = {
3953:                 "index": record.index,
3954:                 "timestamp": record.timestamp,
3955:                 "method_name": record.method_name,
3956:                 "evidence": record.evidence,
3957:                 "metadata": record.metadata,
3958:                 "previous_hash": previous_hash,
3959:             }
3960:             computed = hashlib.sha256(_canonical_json(payload).encode("utf-8")).hexdigest()
3961:             if computed != record.entry_hash:
3962:                 raise ValueError(
3963:                     "Evidence record hash mismatch at index "
3964:                     f"{record.index}: expected {computed}, found {record.entry_hash}"
3965:                 )
3966:             previous_hash = record.entry_hash
3967:
3968: def _serialize_record(record: EvidenceRecord) -> dict[str, Any]:
3969:     payload = asdict(record)
3970:     payload["evidence"] = list(record.evidence)
```

```
    payload["metadata"] = dict(record.metadata)
    return payload

__all__ = [
    "EvidenceRecord",
    "EvidenceRegistry",
]


================================================================================
FILE: src/farfan_pipeline/utils/hash_utils.py
================================================================================

"""
Hash utilities for deterministic content hashing.

This module provides cryptographic hashing functions used across the pipeline
for content integrity verification and change detection.

Author: Integration Team
Version: 1.0.0
Python: 3.10+
"""

import hashlib
import json
from typing import Any


def compute_hash(data: dict[str, Any]) -> str:
    """
    Compute deterministic SHA-256 hash of dictionary data.

    This function creates a canonical JSON representation with sorted keys
    and stable separators to ensure identical dictionaries always produce
    the same hash, regardless of key insertion order.

    Args:
        data: Dictionary to hash

    Returns:
        Hexadecimal SHA-256 digest (64 characters)

    Example:
        >>> data = {"b": 2, "a": 1}
        >>> hash1 = compute_hash(data)
        >>> hash2 = compute_hash({"a": 1, "b": 2})
        >>> hash1 == hash2
        True
    """
    canonical_json = json.dumps(
        data, sort_keys=True, ensure_ascii=True, separators=(",", ":")
    )
    return hashlib.sha256(canonical_json.encode("utf-8")).hexdigest()

```

```
4027:
4028:
4029: ================================================================================
4030: FILE: src/farfan_pipeline/utils/json_contract_loader.py
4031: ================================================================================
4032:
4033: """Utility helpers to load and validate JSON contract documents."""
4034: from __future__ import annotations
4035:
4036: import hashlib
4037: import json
4038: from dataclasses import dataclass
4039: from pathlib import Path
4040: from typing import TYPE_CHECKING, Union
4041: from farfan_pipeline.core.calibration.decorators import calibrated_method
4042:
4043: if TYPE_CHECKING:
4044:     from collections.abc import Iterable, Mapping
4045:
4046: PathLike = Union[str, Path]
4047:
4048: def _canonical_dump(payload: Mapping[str, object]) -> str:
4049:     return json.dumps(payload, ensure_ascii=False, sort_keys=True, separators=(",", ":"))
4050:
4051: @dataclass(frozen=True)
4052: class ContractDocument:
4053:     """Materialized JSON contract with checksum information."""
4054:
4055:     path: Path
4056:     payload: dict[str, object]
4057:     checksum: str
4058:
4059: @dataclass
4060: class ContractLoadReport:
4061:     """Result of attempting to load multiple contract documents."""
4062:
4063:     documents: dict[str, ContractDocument]
4064:     errors: list[str]
4065:
4066:     @property
4067:     @calibrated_method("farfan_core.utils.json_contract_loader.ContractLoadReport.is_successful")
4068:     def is_successful(self) -> bool:
4069:         return not self.errors
4070:
4071:     @calibrated_method("farfan_core.utils.json_contract_loader.ContractLoadReport.summary")
4072:     def summary(self) -> str:
4073:         parts = [f"contracts={len(self.documents)}"]
4074:         if self.errors:
4075:             parts.append(f"errors={len(self.errors)}")
4076:         return ", ".join(parts)
4077:
4078: class JSONContractLoader:
4079:     """Load JSON contract files and compute integrity metadata.
4080:
4081:     ARCHITECTURAL BOUNDARY: This loader is for generic JSON contracts ONLY.
4082:     It must NOT be used to load questionnaire_monolith.json directly.
```

```
4083:
4084:     For questionnaire access, use:
4085:     - factory.load_questionnaire() for canonical loading (returns CanonicalQuestionnaire)
4086:     - QuestionnaireResourceProvider for pattern extraction
4087:     """
4088:
4089:     def __init__(self, base_path: Path | None = None) -> None:
4090:         self.base_path = base_path or Path(__file__).resolve().parent
4091:
4092:     @calibrated_method("farfan_core.utils.json_contract_loader.JSONContractLoader.load")
4093:     def load(self, paths: Iterable[PathLike]) -> ContractLoadReport:
4094:         documents: dict[str, ContractDocument] = {}
4095:         errors: list[str] = []
4096:         for raw in paths:
4097:             path = self._resolve_path(raw)
4098:             try:
4099:                 payload = self._read_payload(path)
4100:             except (FileNotFoundError, json.JSONDecodeError, ValueError) as exc:
4101:                 errors.append(f"{path}: {exc}")
4102:                 continue
4103:
4104:             checksum = hashlib.sha256(_canonical_dump(payload).encode("utf-8")).hexdigest()
4105:             documents[str(path)] = ContractDocument(path=path, payload=payload, checksum=checksum)
4106:         return ContractLoadReport(documents=documents, errors=errors)
4107:
4108:     @calibrated_method("farfan_core.utils.json_contract_loader.JSONContractLoader.load_directory")
4109:     def load_directory(self, relative_directory: PathLike, pattern: str = "*.json") -> ContractLoadReport:
4110:         directory = self._resolve_path(relative_directory)
4111:         if not directory.exists():
4112:             return ContractLoadReport(documents={}, errors=[f"Directory not found: {directory}"])
4113:         if not directory.is_dir():
4114:             return ContractLoadReport(documents={}, errors=[f"Not a directory: {directory}"])
4115:
4116:         paths = sorted(directory.glob(pattern))
4117:         return self.load(paths)
4118:
4119:     # ------------------------------------------------------------------
4120:     # Helpers
4121:     # ------------------------------------------------------------------
4122:     @calibrated_method("farfan_core.utils.json_contract_loader.JSONContractLoader._resolve_path")
4123:     def _resolve_path(self, raw: PathLike) -> Path:
4124:         path = Path(raw)
4125:         if not path.is_absolute():
4126:             path = self.base_path / path
4127:         return path
4128:
4129:     @staticmethod
4130:     def _read_payload(path: Path) -> dict[str, object]:
4131:         # ARCHITECTURAL GUARD: Block unauthorized questionnaire monolith access
4132:         if path.name == "questionnaire_monolith.json":
4133:             raise ValueError(
4134:                 "ARCHITECTURAL VIOLATION: questionnaire_monolith.json must ONLY be "
4135:                 "loaded via factory.load_questionnaire() which enforces hash verification. "
4136:                 "Use factory.load_questionnaire() for canonical loading."
4137:             )
4138:
```

```
4139:         text = path.read_text(encoding="utf-8")
4140:         data = json.loads(text)
4141:         if not isinstance(data, dict):
4142:             raise ValueError("Contract document must be a JSON object")
4143:         return data
4144:
4145: __all__ = [
4146:     "ContractDocument",
4147:     "ContractLoadReport",
4148:     "JSONContractLoader",
4149: ]
4150:
4151:
```