```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 14
================================================================================
Generated: 2025-12-07T06:17:21.259895
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/core/orchestrator/evidence_assembler.py
================================================================================

from __future__ import annotations

import statistics
from typing import Any, Iterable, Literal

try:
    import structlog
    logger = structlog.get_logger(__name__)
except ImportError:
    import logging
    logger = logging.getLogger(__name__)

def _resolve_value(source: str, method_outputs: dict[str, Any]) -> Any:
    """Resolve dotted source paths from method_outputs."""
    if not source:
        return None
    parts = source.split(".")
    current: Any = method_outputs
    for idx, part in enumerate(parts):
        if idx == 0 and part in method_outputs:
            current = method_outputs[part]
            continue
        if isinstance(current, dict) and part in current:
            current = current[part]
        else:
            return None
    return current


class EvidenceAssembler:
    """
    Assemble evidence fields from method outputs using deterministic merge strategies.
    """

    MERGE_STRATEGIES = {
        "concat",
        "first",
        "last",
        "mean",
        "max",
        "min",
        "weighted_mean",
        "majority",
    }
```

```

    @staticmethod
    def assemble(
        method_outputs: dict[str, Any],
        assembly_rules: list[dict[str, Any]],
        signal_pack: Any | None = None,  # NEW: Optional signal pack for provenance
    ) -> dict[str, Any]:
        evidence: dict[str, Any] = {}
        trace: dict[str, Any] = {}

        # NEW: Track signal pack provenance if provided
        if signal_pack is not None:
            trace["signal_provenance"] = {
                "signal_pack_id": getattr(signal_pack, "id", None) or getattr(signal_pack, "pack_id", "unknown"),
                "policy_area": getattr(signal_pack, "policy_area", None) or getattr(signal_pack, "policy_area_id", None),
                "version": getattr(signal_pack, "version", "unknown"),
                "patterns_available": len(getattr(signal_pack, "patterns", [])),
                "source_hash": getattr(signal_pack, "source_hash", None),
            }
            logger.info(
                "signal_pack_attached",
                signal_pack_id=trace["signal_provenance"]["signal_pack_id"],
                policy_area=trace["signal_provenance"]["policy_area"],
            )

        if "_signal_usage" in method_outputs:
            logger.info("signal_consumption_trace", signals_used=method_outputs["_signal_usage"])
            trace["signal_usage"] = method_outputs["_signal_usage"]
            # Remove from method_outputs to not interfere with evidence assembly
            del method_outputs["_signal_usage"]

        for rule in assembly_rules:
            target = rule.get("target")
            sources: Iterable[str] = rule.get("sources", [])
            strategy: str = rule.get("merge_strategy", "first")
            weights: list[float] | None = rule.get("weights")
            default = rule.get("default")

            if strategy not in EvidenceAssembler.MERGE_STRATEGIES:
                raise ValueError(f"Unsupported merge_strategy '{strategy}' for target '{target}'")

            values = []
            for src in sources:
                val = _resolve_value(src, method_outputs)
                if val is not None:
                    values.append(val)

            merged = EvidenceAssembler._merge(values, strategy, weights, default)
            evidence[target] = merged
            trace[target] = {"sources": list(sources), "strategy": strategy, "values": values}

        return {"evidence": evidence, "trace": trace}

    @staticmethod
    def _merge(values: list[Any], strategy: str, weights: list[float] | None, default: Any) -> Any:
        if not values:
```

```
            return default
        if strategy == "first":
            return values[0]
        if strategy == "last":
            return values[-1]
        if strategy == "concat":
            merged: list[Any] = []
            for v in values:
                if isinstance(v, list):
                    merged.extend(v)
                else:
                    merged.append(v)
            return merged
        numeric_values = [float(v) for v in values if EvidenceAssembler._is_number(v)]
        if strategy == "mean":
            return statistics.fmean(numeric_values) if numeric_values else default
        if strategy == "max":
            return max(numeric_values) if numeric_values else default
        if strategy == "min":
            return min(numeric_values) if numeric_values else default
        if strategy == "weighted_mean":
            if not numeric_values:
                return default
            if not weights:
                weights = [1.0] * len(numeric_values)
            w = weights[: len(numeric_values)] or [1.0] * len(numeric_values)
            total = sum(w) or 1.0
            return sum(v * w_i for v, w_i in zip(numeric_values, w)) / total
        if strategy == "majority":
            counts: dict[Any, int] = {}
            for v in values:
                counts[v] = counts.get(v, 0) + 1
            return max(counts.items(), key=lambda item: item[1])[0] if counts else default
        return default

    @staticmethod
    def _is_number(value: Any) -> bool:
        try:
            float(value)
            return not isinstance(value, bool)
        except (TypeError, ValueError) as e:
            if logger.isEnabledFor(logging.DEBUG):
                logger.debug(f"Non-numeric value: {value!r} ({type(value).__name__}): {e}")
            return False



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/evidence_registry.py
================================================================================

"""
Evidence Registry: Append-Only JSONL Store with Hash Chain and Provenance DAG Export

This module implements a comprehensive evidence tracking system that:
1. Stores all evidence in append-only JSONL format for immutability
```

```
2. Maintains hash-based indexing for fast evidence lookup
3. Implements blockchain-style hash chaining for ledger integrity
4. Exports provenance DAG showing evidence lineage and dependencies
5. Provides cryptographic verification of evidence integrity

Architecture:
- JSONL Storage: One JSON object per line, append-only for audit trail
- Hash Index: SHA-256 hashes for content-addressable storage
- Hash Chain: Each entry links to previous via previous_hash and entry_hash
- Provenance DAG: Directed acyclic graph of evidence dependencies
- Verification: Cryptographic chain-of-custody validation with chain linkage checks

Hash Chain Security:
The registry implements a blockchain-style hash chain where each entry contains:
- content_hash: SHA-256 of the payload (for content verification)
- previous_hash: Hash of the previous entry’s entry_hash (creates the chain)
- entry_hash: SHA-256 of (content_hash + previous_hash + metadata)

This ensures that:
1. Any tampering with payload is detected via content_hash mismatch
2. Any tampering with previous_hash is detected via chain verification
3. Entries cannot be reordered without breaking the chain
4. The entire ledger history can be cryptographically verified
"""

from __future__ import annotations

import hashlib
import json
import logging
import time
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

logger = logging.getLogger(__name__)

@dataclass
class EvidenceRecord:
    """
    Immutable evidence record with provenance metadata and hash chain linkage.

    Each evidence record captures:
    - Unique identifier (hash-based)
    - Evidence payload (method result, analysis output, etc.)
    - Provenance metadata (source, dependencies, lineage)
    - Temporal metadata (timestamp, execution time)
    - Verification data (content hash, chain hashes)

    Hash Chain Fields:
    - content_hash: SHA-256 of payload (verifies content integrity)
    - previous_hash: entry_hash of previous record (creates chain linkage)
    - entry_hash: SHA-256 of (content + previous_hash + metadata) (unique entry ID)

```

```
    The hash chain ensures that:
    1. Tampering with payload breaks content_hash
    2. Tampering with previous_hash breaks chain verification
    3. Entire ledger history is cryptographically verifiable
    """

    # Identification
    evidence_id: str  # SHA-256 hash of content
    evidence_type: str  # Type of evidence (e.g., "method_result", "analysis", "extraction")

    # Payload
    payload: dict[str, Any]

    # Provenance
    source_method: str | None = None  # FQN of method that produced this evidence
    parent_evidence_ids: list[str] = field(default_factory=list)  # Dependencies
    question_id: str | None = None
    document_id: str | None = None

    # Temporal
    timestamp: float = field(default_factory=time.time)
    execution_time_ms: float = 0.0

    # Verification
    content_hash: str | None = None  # Hash of payload for verification
    previous_hash: str | None = None  # Hash of previous entry in chain (for ledger integrity)
    entry_hash: str | None = None  # Hash of this entire entry including previous_hash

    # Metadata
    metadata: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):
        """Generate content hash and entry hash if not provided."""
        if self.content_hash is None:
            self.content_hash = self._compute_content_hash()
        if self.entry_hash is None:
            self.entry_hash = self._compute_entry_hash()

    def _canonical_dump(self, obj: Any) -> str:
        """
        Create canonical JSON representation for deterministic hashing.

        This method ensures:
        - Keys are sorted alphabetically
        - No whitespace in output
        - Consistent handling of None, booleans, numbers
        - Deterministic ordering for nested structures
        - Unicode normalization

        Uses a custom JSON serialization handler to support non-standard types
        commonly found in evidence payloads (dataclasses, NumPy arrays, custom objects).
        The handler converts objects to dicts via __dict__ or falls back to string
        representation, ensuring all evidence can be serialized without exceptions.

        Args:
            obj: Object to serialize
```

```

        Returns:
            Canonical JSON string
        """
        # Use separators with no spaces and sort keys for determinism
        # ensure_ascii=True ensures consistent output across platforms
        # Custom handler for non-serializable types (dataclasses, NumPy arrays, etc.)
        def default_handler(o):
            if hasattr(o, '__dict__'):
                return o.__dict__
            return str(o)

        return json.dumps(
            obj,
            sort_keys=True,
            separators=(',', ':'),
            ensure_ascii=True,
            default=default_handler
        )

    def _compute_content_hash(self) -> str:
        """
        Compute SHA-256 hash of payload for content-addressable storage.

        Uses canonical JSON serialization to ensure deterministic hashing
        across different Python versions and platforms.

        Returns:
            Hex digest of SHA-256 hash
        """
        # Create deterministic JSON representation using canonical dump
        payload_json = self._canonical_dump(self.payload)

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

    def _compute_entry_hash(self) -> str:
        """
        Compute SHA-256 hash of the entire entry including previous_hash.
        This creates the hash chain linking entries together.

        Uses canonical JSON serialization for deterministic hashing.

        Returns:
            Hex digest of SHA-256 hash
        """
        # Combine content hash with previous hash to create chain
        # Use empty string for first entry (no predecessor)
        chain_data = {
            'content_hash': self.content_hash,
            'previous_hash': self.previous_hash if self.previous_hash is not None else '',
            'evidence_type': self.evidence_type,
            'timestamp': self.timestamp,
        }
        chain_json = self._canonical_dump(chain_data)
```

```
        hash_obj = hashlib.sha256(chain_json.encode('utf-8'))
        return hash_obj.hexdigest()

    def verify_integrity(self, previous_record: EvidenceRecord | None = None) -> bool:
        """
        Verify evidence integrity by recomputing hashes and checking chain linkage.

        Args:
            previous_record: The record that should precede this one in the chain

        Returns:
            True if all integrity checks pass, False otherwise
        """
        # Verify content hash matches
        current_content_hash = self._compute_content_hash()
        if current_content_hash != self.content_hash:
            return False

        # Verify entry hash matches
        current_entry_hash = self._compute_entry_hash()
        if current_entry_hash != self.entry_hash:
            return False

        # If previous record is provided, verify the chain linkage
        if previous_record is not None:
            # Verify that our previous_hash matches the actual hash of the previous record
            if self.previous_hash != previous_record.entry_hash:
                return False

        return True

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        return asdict(self)

    @classmethod
    def from_dict(cls, data: dict[str, Any]) -> EvidenceRecord:
        """Create evidence record from dictionary."""
        return cls(**data)

    @classmethod
    def create(
        cls,
        evidence_type: str,
        payload: dict[str, Any],
        source_method: str | None = None,
        parent_evidence_ids: list[str] | None = None,
        question_id: str | None = None,
        document_id: str | None = None,
        execution_time_ms: float = 0.0,
        metadata: dict[str, Any] | None = None,
        previous_hash: str | None = None,
    ) -> EvidenceRecord:
        """
        Create a new evidence record with proper hash computation.

```

```
        This factory method ensures:
        - Proper initialization order
        - Deterministic hash computation
        - Validation of required fields

        Args:
            evidence_type: Type of evidence
            payload: Evidence data (must be JSON-serializable)
            source_method: FQN of method that produced evidence
            parent_evidence_ids: List of parent evidence IDs
            question_id: Question ID this evidence relates to
            document_id: Document ID this evidence relates to
            execution_time_ms: Execution time in milliseconds
            metadata: Additional metadata
            previous_hash: Hash of previous entry in chain (for chain linkage)

        Returns:
            New EvidenceRecord instance

        Raises:
            ValueError: If required fields are invalid or payload is not serializable
        """
        if not evidence_type:
            raise ValueError("evidence_type is required")

        if not isinstance(payload, dict):
            raise ValueError("payload must be a dictionary")

        # Test that payload is JSON-serializable
        try:
            json.dumps(payload)
        except (TypeError, ValueError) as e:
            raise ValueError(f"payload must be JSON-serializable: {e}")

        # Create record with temporary evidence_id
        record = cls(
            evidence_id="",
            evidence_type=evidence_type,
            payload=payload,
            source_method=source_method,
            parent_evidence_ids=parent_evidence_ids or [],
            question_id=question_id,
            document_id=document_id,
            execution_time_ms=execution_time_ms,
            metadata=metadata or {},
            previous_hash=previous_hash,
        )

        # Set evidence_id to content hash (computed in __post_init__)
        record.evidence_id = record.content_hash or ""

        return record

@dataclass
class ProvenanceNode:
    """Node in provenance DAG."""
```

```

    evidence_id: str
    evidence_type: str
    source_method: str | None
    timestamp: float
    children: list[str] = field(default_factory=list)  # Evidence IDs that depend on this
    parents: list[str] = field(default_factory=list)   # Evidence IDs this depends on

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

@dataclass
class ProvenanceDAG:
    """
    Directed Acyclic Graph of evidence provenance.

    Captures the full lineage of evidence:
    - Which evidence produced which other evidence
    - Method invocation chains
    - Data flow dependencies
    """

    nodes: dict[str, ProvenanceNode] = field(default_factory=dict)

    # Index for fast queries
    by_method: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
    by_type: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
    by_question: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))

    def add_evidence(
        self,
        evidence: EvidenceRecord
    ) -> None:
        """
        Add evidence to provenance DAG.

        Args:
            evidence: Evidence record to add
        """
        # Create node
        node = ProvenanceNode(
            evidence_id=evidence.evidence_id,
            evidence_type=evidence.evidence_type,
            source_method=evidence.source_method,
            timestamp=evidence.timestamp,
            parents=evidence.parent_evidence_ids.copy(),
        )

        # Add to nodes
        self.nodes[evidence.evidence_id] = node

        # Update parent-child relationships
        for parent_id in evidence.parent_evidence_ids:
            if parent_id in self.nodes:
                self.nodes[parent_id].children.append(evidence.evidence_id)
```

```

        # Update indices
        if evidence.source_method:
            self.by_method[evidence.source_method].append(evidence.evidence_id)
        self.by_type[evidence.evidence_type].append(evidence.evidence_id)
        if evidence.question_id:
            self.by_question[evidence.question_id].append(evidence.evidence_id)

    def get_ancestors(self, evidence_id: str) -> set[str]:
        """
        Get all ancestor evidence IDs (transitive parents).

        Args:
            evidence_id: Evidence ID to trace

        Returns:
            Set of ancestor evidence IDs
        """
        ancestors = set()
        visited = set()

        def traverse(eid: str) -> None:
            if eid in visited:
                return
            visited.add(eid)

            if eid not in self.nodes:
                return

            node = self.nodes[eid]
            for parent_id in node.parents:
                ancestors.add(parent_id)
                traverse(parent_id)

        traverse(evidence_id)
        return ancestors

    def get_descendants(self, evidence_id: str) -> set[str]:
        """
        Get all descendant evidence IDs (transitive children).

        Args:
            evidence_id: Evidence ID to trace

        Returns:
            Set of descendant evidence IDs
        """
        descendants = set()
        visited = set()

        def traverse(eid: str) -> None:
            if eid in visited:
                return
            visited.add(eid)

            if eid not in self.nodes:
```

```
                return

            node = self.nodes[eid]
            for child_id in node.children:
                descendants.add(child_id)
                traverse(child_id)

        traverse(evidence_id)
        return descendants

    def get_lineage(self, evidence_id: str) -> dict[str, Any]:
        """
        Get complete lineage for evidence (ancestors + descendants).

        Args:
            evidence_id: Evidence ID to trace

        Returns:
            Dictionary with lineage information
        """
        return {
            "evidence_id": evidence_id,
            "ancestors": list(self.get_ancestors(evidence_id)),
            "descendants": list(self.get_descendants(evidence_id)),
            "ancestor_count": len(self.get_ancestors(evidence_id)),
            "descendant_count": len(self.get_descendants(evidence_id)),
        }

    def export_dot(self) -> str:
        """
        Export DAG in GraphViz DOT format.

        Returns:
            DOT format string
        """
        lines = ["digraph ProvenanceDAG {"]
        lines.append("  rankdir=LR;")
        lines.append("  node [shape=box];")
        lines.append("")

        # Add nodes
        for eid, node in self.nodes.items():
            label = f"{node.evidence_type}\\n{eid[:8]}..."
            if node.source_method:
                label += f"\\n{node.source_method}"
            lines.append(f'  "{eid}" [label="{label}"];')

        lines.append("")

        # Add edges
        for eid, node in self.nodes.items():
            for child_id in node.children:
                lines.append(f'  "{eid}" -> "{child_id}";')

        lines.append("}")
        return "\n".join(lines)
```

```

    def to_dict(self) -> dict[str, Any]:
        """Export DAG to dictionary."""
        return {
            "nodes": {eid: node.to_dict() for eid, node in self.nodes.items()},
            "stats": {
                "total_nodes": len(self.nodes),
                "by_method": {k: len(v) for k, v in self.by_method.items()},
                "by_type": {k: len(v) for k, v in self.by_type.items()},
                "by_question": {k: len(v) for k, v in self.by_question.items()},
            }
        }

class EvidenceRegistry:
    """
    Append-only evidence registry with hash indexing and provenance tracking.

    Features:
    - JSONL append-only storage for immutability
    - Content-addressable hash indexing
    - Provenance DAG for lineage tracking
    - Cryptographic verification
    - Fast queries by hash, type, method, question
    """

    def __init__(
        self,
        storage_path: Path | None = None,
        enable_dag: bool = True,
    ) -> None:
        """
        Initialize evidence registry.

        Args:
            storage_path: Path to JSONL storage file (default: evidence_registry.jsonl)
            enable_dag: Enable provenance DAG tracking
        """
        self.storage_path = storage_path or Path("evidence_registry.jsonl")
        self.enable_dag = enable_dag

        # Hash index: hash -> evidence record
        self.hash_index: dict[str, EvidenceRecord] = {}

        # Type index: type -> list of hashes
        self.type_index: dict[str, list[str]] = defaultdict(list)

        # Method index: method FQN -> list of hashes
        self.method_index: dict[str, list[str]] = defaultdict(list)

        # Question index: question ID -> list of hashes
        self.question_index: dict[str, list[str]] = defaultdict(list)

        # Provenance DAG
        self.dag = ProvenanceDAG() if enable_dag else None

        # Track the last entry in the ledger chain for hash chaining
```

```
        self.last_entry: EvidenceRecord | None = None

        # Load existing evidence
        self._load_from_storage()

        logger.info(
            f"EvidenceRegistry initialized with {len(self.hash_index)} records, "
            f"storage={self.storage_path}, dag={'enabled' if enable_dag else 'disabled'}"
        )

    def _load_from_storage(self) -> None:
        """
        Load evidence from JSONL storage with chain verification.

        Ensures:
        - Evidence is loaded in the order it was written
        - Chain linkage is validated during load
        - Index ordering is preserved
        """
        if not self.storage_path.exists():
            logger.info(f"No existing evidence storage found at {self.storage_path}")
            return

        loaded_count = 0
        BATCH_SIZE = 1000
        batch_records = []

        try:
            with open(self.storage_path, encoding='utf-8') as f:
                for line_num, line in enumerate(f, 1):
                    try:
                        data = json.loads(line.strip())
                        evidence = EvidenceRecord.from_dict(data)
                        batch_records.append((line_num, evidence))
                        loaded_count += 1

                        if len(batch_records) >= BATCH_SIZE:
                            self._assert_chain(batch_records)
                            for _, ev in batch_records:
                                self._index_evidence(ev, persist=False)
                            batch_records.clear()

                    except json.JSONDecodeError as e:
                        logger.warning(f"Failed to parse line {line_num}: {e}")
                    except Exception as e:
                        logger.warning(f"Failed to load evidence on line {line_num}: {e}")

            if batch_records:
                self._assert_chain(batch_records)
                for _, ev in batch_records:
                    self._index_evidence(ev, persist=False)
                batch_records.clear()

            logger.info(f"Loaded {loaded_count} evidence records from storage")

        except Exception as e:
```

```
            logger.error(f"Failed to load evidence storage: {e}")

    def _assert_chain(self, records: list[tuple[int, EvidenceRecord]]) -> None:
        """
        Assert that the chain of evidence records is valid.

        Validates:
        - First record has no previous_hash or previous_hash is None
        - Each subsequent record’s previous_hash matches the prior record’s entry_hash
        - Records are in the correct sequential order

        Args:
            records: List of (line_number, EvidenceRecord) tuples in load order

        Raises:
            ValueError: If chain validation fails
        """
        if not records:
            return

        previous_record = None

        for idx, (line_num, record) in enumerate(records):
            if idx == 0:
                # First record should have no previous_hash or None
                if record.previous_hash and record.previous_hash != ’’:
                    logger.warning(
                        f"Line {line_num}: First record has previous_hash={record.previous_hash}, "
                        f"expected None or empty string. Chain may have been corrupted or truncated."
                    )
            # Subsequent records should link to the previous record
            elif previous_record is not None:
                expected_previous_hash = previous_record.entry_hash
                actual_previous_hash = record.previous_hash

                if actual_previous_hash != expected_previous_hash:
                    raise ValueError(
                        f"Chain broken at line {line_num}: "
                        f"expected previous_hash={expected_previous_hash}, "
                        f"got previous_hash={actual_previous_hash}. "
                        f"Evidence ordering may be corrupted."
                    )

            previous_record = record

    def _index_evidence(
        self,
        evidence: EvidenceRecord,
        persist: bool = True
    ) -> None:
        """
        Index evidence record in all indices.

        Args:
            evidence: Evidence to index
            persist: If True, append to JSONL storage
```

```
        """
        # Hash index
        self.hash_index[evidence.evidence_id] = evidence

        # Type index
        self.type_index[evidence.evidence_type].append(evidence.evidence_id)

        # Method index
        if evidence.source_method:
            self.method_index[evidence.source_method].append(evidence.evidence_id)

        # Question index
        if evidence.question_id:
            self.question_index[evidence.question_id].append(evidence.evidence_id)

        # DAG
        if self.enable_dag and self.dag:
            self.dag.add_evidence(evidence)

        # Update last entry for hash chaining
        self.last_entry = evidence

        # Persist to storage
        if persist:
            self._append_to_storage(evidence)

    def _append_to_storage(self, evidence: EvidenceRecord) -> None:
        """
        Append evidence to JSONL storage.

        Args:
            evidence: Evidence to append
        """
        try:
            # Ensure parent directory exists
            self.storage_path.parent.mkdir(parents=True, exist_ok=True)

            # Append to JSONL
            with open(self.storage_path, 'a', encoding='utf-8') as f:
                json_line = json.dumps(evidence.to_dict(), separators=(',', ':'))
                f.write(json_line + '\n')

        except Exception as e:
            logger.error(f"Failed to append evidence to storage: {e}")
            raise

    def record_evidence(
        self,
        evidence_type: str,
        payload: dict[str, Any],
        source_method: str | None = None,
        parent_evidence_ids: list[str] | None = None,
        question_id: str | None = None,
        document_id: str | None = None,
        execution_time_ms: float = 0.0,
        metadata: dict[str, Any] | None = None,
```

```
    ) -> str:
        """
        Record new evidence in registry.

        Args:
            evidence_type: Type of evidence
            payload: Evidence data
            source_method: FQN of method that produced evidence
            parent_evidence_ids: List of parent evidence IDs
            question_id: Question ID this evidence relates to
            document_id: Document ID this evidence relates to
            execution_time_ms: Execution time
            metadata: Additional metadata

        Returns:
            Evidence ID (hash)
        """
        # Determine previous_hash from last entry in the chain
        previous_hash = self.last_entry.entry_hash if self.last_entry else None

        # Normalize metadata and ensure recorded_at timestamp
        metadata_dict: dict[str, Any] = dict(metadata) if metadata else {}
        metadata_dict.setdefault(
            "recorded_at",
            datetime.now(timezone.utc).isoformat(),
        )

        # Create evidence record with deterministic hash-based ID
        evidence = EvidenceRecord.create(
            evidence_type=evidence_type,
            payload=payload,
            source_method=source_method,
            parent_evidence_ids=parent_evidence_ids,
            question_id=question_id,
            document_id=document_id,
            execution_time_ms=execution_time_ms,
            metadata=metadata_dict,
            previous_hash=previous_hash,
        )

        # Check for duplicate
        if evidence.evidence_id in self.hash_index:
            logger.debug(f"Evidence {evidence.evidence_id} already exists, skipping")
            return evidence.evidence_id

        # Index evidence
        self._index_evidence(evidence, persist=True)

        logger.debug(f"Recorded evidence {evidence.evidence_id} of type {evidence_type}")

        return evidence.evidence_id

    def get_evidence(self, evidence_id: str) -> EvidenceRecord | None:
        """
        Retrieve evidence by ID.

```

```
        Args:
            evidence_id: Evidence hash

        Returns:
            EvidenceRecord or None
        """
        return self.hash_index.get(evidence_id)

    def query_by_type(self, evidence_type: str) -> list[EvidenceRecord]:
        """Query evidence by type."""
        evidence_ids = self.type_index.get(evidence_type, [])
        return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]

    def query_by_method(self, method_fqn: str) -> list[EvidenceRecord]:
        """Query evidence by source method."""
        evidence_ids = self.method_index.get(method_fqn, [])
        return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]

    def query_by_question(self, question_id: str) -> list[EvidenceRecord]:
        """Query evidence by question ID."""
        evidence_ids = self.question_index.get(question_id, [])
        return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]

    def verify_evidence(self, evidence_id: str, verify_chain: bool = True) -> bool:
        """
        Verify evidence integrity and optionally chain linkage.

        Args:
            evidence_id: Evidence hash
            verify_chain: If True, verify chain linkage with previous entry

        Returns:
            True if evidence is valid
        """
        evidence = self.get_evidence(evidence_id)
        if evidence is None:
            return False

        # Get previous record if chain verification is requested
        previous_record = None
        if verify_chain and evidence.previous_hash:
            # Find the record with entry_hash matching our previous_hash
            for record in self.hash_index.values():
                if record.entry_hash == evidence.previous_hash:
                    previous_record = record
                    break

        return evidence.verify_integrity(previous_record=previous_record)

    def verify_chain_integrity(self) -> tuple[bool, list[str]]:
        """
        Verify the integrity of the entire evidence chain.

        Returns:
            Tuple of (is_valid, list of errors)
        """
```

```
        errors = []

        # Build the chain by reading from storage in order
        if not self.storage_path.exists():
            return True, []  # Empty chain is valid

        try:
            previous_record = None
            with open(self.storage_path, encoding='utf-8') as f:
                for line_num, line in enumerate(f, 1):
                    try:
                        data = json.loads(line.strip())
                        evidence = EvidenceRecord.from_dict(data)

                        # Verify the record's integrity
                        if not evidence.verify_integrity(previous_record=previous_record):
                            if previous_record and evidence.previous_hash != previous_record.entry_hash:
                                errors.append(
                                    f"Line {line_num}: Chain broken - previous_hash mismatch. "
                                    f"Expected {previous_record.entry_hash}, got {evidence.previous_hash}"
                                )
                            else:
                                errors.append(
                                    f"Line {line_num}: Hash integrity check failed for evidence {evidence.evidence_id}"
                                )

                        previous_record = evidence

                    except json.JSONDecodeError as e:
                        errors.append(f"Line {line_num}: JSON parsing error - {e}")
                    except Exception as e:
                        errors.append(f"Line {line_num}: Verification error - {e}")

            return len(errors) == 0, errors

        except Exception as e:
            return False, [f"Failed to verify chain: {e}"]

    def get_provenance(self, evidence_id: str) -> dict[str, Any] | None:
        """
        Get provenance information for evidence.

        Args:
            evidence_id: Evidence hash

        Returns:
            Provenance dictionary or None
        """
        if not self.enable_dag or self.dag is None:
            return None

        return self.dag.get_lineage(evidence_id)

    def export_provenance_dag(
        self,
        format: str = "dict",
```

```
        output_path: Path | None = None
    ) -> Any:
        """
        Export provenance DAG.

        Args:
            format: Export format ("dict", "dot", "json")
            output_path: Optional path to write output

        Returns:
            Exported DAG in requested format
        """
        if not self.enable_dag or self.dag is None:
            raise ValueError("DAG tracking is not enabled")

        if format == "dot":
            result = self.dag.export_dot()
        elif format == "dict":
            result = self.dag.to_dict()
        elif format == "json":
            result = json.dumps(self.dag.to_dict(), indent=2)
        else:
            raise ValueError(f"Unsupported format: {format}")

        # Write to file if path provided
        if output_path:
            output_path.parent.mkdir(parents=True, exist_ok=True)
            if isinstance(result, str):
                output_path.write_text(result, encoding='utf-8')
            else:
                output_path.write_text(json.dumps(result, indent=2), encoding='utf-8')
            logger.info(f"Exported provenance DAG to {output_path}")

        return result

    def get_statistics(self) -> dict[str, Any]:
        """
        Get registry statistics.

        Returns:
            Statistics dictionary
        """
        stats = {
            "total_evidence": len(self.hash_index),
            "by_type": {k: len(v) for k, v in self.type_index.items()},
            "by_method": {k: len(v) for k, v in self.method_index.items()},
            "by_question": {k: len(v) for k, v in self.question_index.items()},
            "storage_path": str(self.storage_path),
            "dag_enabled": self.enable_dag,
        }

        if self.enable_dag and self.dag:
            stats["dag_nodes"] = len(self.dag.nodes)

        return stats
```

```
    def stats(self) -> dict[str, int]:
        """Get simplified evidence registry statistics.

        Returns:
            Dict with counts for records, types, methods, and questions.
        """
        return {
            "records": len(self.hash_index),
            "types": len(self.type_index),
            "methods": len(self.method_index),
            "questions": len(self.question_index),
        }

# Global registry instance
_global_registry: EvidenceRegistry | None = None

def get_global_registry() -> EvidenceRegistry:
    """Get or create global evidence registry."""
    global _global_registry
    if _global_registry is None:
        _global_registry = EvidenceRegistry()
    return _global_registry

__all__ = [
    "EvidenceRecord",
    "ProvenanceNode",
    "ProvenanceDAG",
    "EvidenceRegistry",
    "get_global_registry",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/evidence_validator.py
================================================================================

from __future__ import annotations

import re
from typing import Any, Iterable


class EvidenceValidator:
    """Validate assembled evidence with configurable rules."""

    @staticmethod
    def validate(
        evidence: dict[str, Any],
        rules_object: dict[str, Any],
        failure_contract: dict[str, Any] | None = None,  # NEW: Signal-based failure contract
    ) -> dict[str, Any]:
        """
        Validates evidence against a rules object from a V2 contract.

        Args:
```

```
            evidence: The assembled evidence dictionary.
            rules_object: The validation object from the contract, containing
                          'rules' (a list) and 'na_policy' (a string).
            failure_contract: Optional failure contract from signal pack containing
                              'abort_if' conditions and 'emit_code' for abort signal.
        """
        validation_rules = rules_object.get("rules", [])
        na_policy = rules_object.get("na_policy", "abort_on_critical")
        errors: list[str] = []
        warnings: list[str] = []
        abort_code: str | None = None

        for rule in validation_rules:
            field = rule.get("field")
            value = EvidenceValidator._resolve(field, evidence)

            # --- New Rich Rule Logic ---
            if rule.get("must_contain"):
                must_contain = rule["must_contain"]
                required_elements = set(must_contain.get("elements", []))
                present_elements = set(value) if isinstance(value, list) else set()
                missing_elements = required_elements - present_elements
                if missing_elements:
                    errors.append(f"Field '{field}' is missing required elements: {', '.join(sorted(missing_elements))}")

                required_count = must_contain.get("count")
                if required_count and len(present_elements.intersection(required_elements)) < required_count:
                    errors.append(f"Field '{field}' did not meet the required count of {required_count} for elements: {', '.join(sorted(required_elements))}")

            if rule.get("should_contain"):
                should_contain = rule["should_contain"]
                present_elements = set(value) if isinstance(value, list) else set()
                for requirement in should_contain:
                    elements_to_check = set(requirement.get("elements", []))
                    min_count = requirement.get("minimum", 1)
                    found_count = len(present_elements.intersection(elements_to_check))
                    if found_count < min_count:
                        warnings.append(f"Field '{field}' only has {found_count}/{min_count} of recommended elements: {', '.join(sorted(elements_to_check))}")

            # --- Original Simple Rule Logic ---
            missing = value is None
            if rule.get("required") and missing:
                errors.append(f"Missing required field '{field}'")
                continue
            if missing:
                continue

            if rule.get("type", "any") != "any" and not EvidenceValidator._check_type(value, rule["type"]):
                errors.append(f"Field '{field}' has incorrect type (expected {rule['type']})")
                continue

            if rule.get("min_length") is not None and EvidenceValidator._has_length(value) and len(value) < rule["min_length"]:
                errors.append(f"Field '{field}' length below min_length {rule['min_length']}")

```

```
            if rule.get("pattern") and isinstance(value, str) and not re.search(rule["pattern"], value):
                errors.append(f"Field '{field}' does not match pattern")

        # NEW: Process failure_contract from signal pack
        if failure_contract and errors:
            abort_conditions = failure_contract.get("abort_if", [])
            emit_code = failure_contract.get("emit_code", "SIGNAL_ABORT")
            severity = failure_contract.get("severity", "ERROR")

            for condition in abort_conditions:
                condition_triggered = False
                if condition == "missing_required_element":
                    condition_triggered = any("missing required" in e.lower() for e in errors)
                elif condition == "type_mismatch":
                    condition_triggered = any("incorrect type" in e.lower() for e in errors)
                elif condition == "pattern_mismatch":
                    condition_triggered = any("does not match pattern" in e.lower() for e in errors)
                elif condition == "any_error":
                    condition_triggered = len(errors) > 0

                if condition_triggered:
                    abort_code = emit_code
                    if severity == "CRITICAL":
                        raise ValueError(
                            f"ABORT[{emit_code}]: Failure contract triggered by condition '{condition}'. Errors: {'; '.join(errors)}"
                        )
                    break

        valid = not errors
        if errors and na_policy == "abort_on_critical" and not abort_code:
            raise ValueError(f"Evidence validation failed with critical errors: {'; '.join(errors)}")

        return {
            "valid": valid,
            "errors": errors,
            "warnings": warnings,
            "abort_code": abort_code,  # NEW: Track signal-based abort code
            "failure_contract_triggered": abort_code is not None,
        }

    @staticmethod
    def _resolve(path: str, evidence: dict[str, Any]) -> Any:
        if not path:
            return None
        parts = path.split(".")
        current: Any = evidence
        for part in parts:
            if isinstance(current, dict) and part in current:
                current = current[part]
            else:
                return None
        return current

    @staticmethod
    def _check_type(value: Any, expected: str) -> bool:
        mapping = {
```

```
            "array": (list, tuple),
            "integer": (int,),
            "float": (float, int),
            "string": (str,),
            "boolean": (bool,),
            "object": (dict,),
            "any": (object,),
        }
        return isinstance(value, mapping.get(expected, (object,)))

    @staticmethod
    def _has_length(value: Any) -> bool:
        return hasattr(value, "__len__")

    @staticmethod
    def _is_number(value: Any) -> bool:
        try:
            float(value)
            return not isinstance(value, bool)
        except (TypeError, ValueError):
            return False



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/executor_config.py
================================================================================

from __future__ import annotations

from dataclasses import dataclass
from typing import Any


@dataclass
class ExecutorConfig:
    """
    Lightweight configuration for executors.

    This is intentionally minimal and only covers the parameters currently
    referenced by wiring/bootstrap code. Extend cautiously if new executor
    settings are required.
    """

    max_tokens: int | None = None
    temperature: float | None = None
    timeout_s: float | None = None
    retry: int | None = None
    seed: int | None = None
    extra: dict[str, Any] | None = None

    def __post_init__(self) -> None:
        # Basic type guards without altering semantics
        if self.max_tokens is not None and self.max_tokens <= 0:
            raise ValueError("max_tokens must be positive when provided")
        if self.retry is not None and self.retry < 0:
```

```
            raise ValueError("retry must be non-negative when provided")


__all__ = ["ExecutorConfig"]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/executor_profiler.py
================================================================================

"""Executor performance profiling framework with regression detection and dispensary analytics.

This module provides comprehensive profiling for executor performance including:
- Per-executor timing, memory, and serialization metrics
- Method call tracking with granular statistics
- Baseline comparison for regression detection
- Performance report generation identifying bottlenecks
- Integration with BaseExecutor for automatic capture
- **METHOD DISPENSARY PATTERN AWARENESS** for tracking monolith reuse

Architecture:
- ExecutorMetrics: Per-executor performance data
- MethodCallMetrics: Per-method call statistics
- ExecutorProfiler: Main profiler with baseline management + dispensary analytics
- ProfilerContext: Context manager for automatic profiling
- PerformanceReport: Structured report with bottleneck analysis

METHOD DISPENSARY INTEGRATION:
==============================
This profiler is aware of the factory's method dispensary pattern where:
- 30 executors orchestrate methods from ~20 monolith classes
- Methods are called via MethodExecutor.execute(class_name, method_name, **payload)
- Same methods are PARTIALLY reused across different executors
- Dispensary classes: PDETMunicipalPlanAnalyzer (52+ methods), CausalExtractor (28), etc.

The profiler tracks:
1. Which dispensary classes are used by each executor
2. Method reuse patterns across executors
3. Performance hotspots within dispensary classes
4. Executor-specific vs dispensary-wide bottlenecks

Usage:
    # Basic profiling
    profiler = ExecutorProfiler()
    with profiler.profile_executor("D1-Q1"):
        result = executor.execute(context)
    report = profiler.generate_report()

    # With dispensary analytics
    dispensary_stats = profiler.get_dispensary_usage_stats()
    # Shows: PDETMunicipalPlanAnalyzer used by 15 executors, avg 245ms/call
"""

from __future__ import annotations

```

```
import gc
import json
import logging
import pickle
import time
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

logger = logging.getLogger(__name__)

# Performance thresholds (loaded from canonical_method_catalogue_v2.json via calibration system)
# These are DEFAULT values only - actual thresholds come from method_parameters.json
DEFAULT_HIGH_EXECUTION_TIME_MS = 1000
DEFAULT_HIGH_MEMORY_MB = 100
DEFAULT_HIGH_SERIALIZATION_MS = 100

# Known dispensary classes from the method dispensary pattern
KNOWN_DISPENSARY_CLASSES = {
    "PDETMunicipalPlanAnalyzer",
    "IndustrialPolicyProcessor",
    "CausalExtractor",
    "FinancialAuditor",
    "BayesianMechanismInference",
    "BayesianCounterfactualAuditor",
    "TextMiningEngine",
    "SemanticAnalyzer",
    "PerformanceAnalyzer",
    "PolicyContradictionDetector",
    "BayesianNumericalAnalyzer",
    "TemporalLogicVerifier",
    "OperationalizationAuditor",
    "PolicyAnalysisEmbedder",
    "SemanticProcessor",
    "AdvancedDAGValidator",
    "TeoriaCambio",
    "ReportingEngine",
    "HierarchicalGenerativeModel",
    "AdaptivePriorCalculator",
    "PolicyTextProcessor",
    "MechanismPartExtractor",
    "CausalInferenceSetup",
    "BeachEvidentialTest",
    "BayesFactorTable",
    "ConfigLoader",
    "CDAFFramework",
    "IndustrialGradeValidator",
    "BayesianConfidenceCalculator",
    "PDFProcessor",
}


@dataclass
class MethodCallMetrics:
```

```
    """Metrics for a single method call within an executor.

    Enhanced to track dispensary pattern usage.
    """

    class_name: str
    method_name: str
    execution_time_ms: float
    memory_delta_mb: float
    call_count: int = 1
    success: bool = True
    error: str | None = None
    timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )

    @property
    def is_dispensary_method(self) -> bool:
        """Check if this method comes from a known dispensary class."""
        return self.class_name in KNOWN_DISPENSARY_CLASSES

    @property
    def full_method_name(self) -> str:
        """Get full method name as class.method."""
        return f"{self.class_name}.{self.method_name}"

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        data = asdict(self)
        data["is_dispensary_method"] = self.is_dispensary_method
        data["full_method_name"] = self.full_method_name
        return data


@dataclass
class ExecutorMetrics:
    """Comprehensive metrics for a single executor execution.

    Enhanced with dispensary usage tracking.
    """

    executor_id: str
    execution_time_ms: float
    memory_footprint_mb: float
    memory_peak_mb: float
    serialization_time_ms: float
    serialization_size_bytes: int
    method_calls: list[MethodCallMetrics] = field(default_factory=list)
    call_count: int = 1
    success: bool = True
    error: str | None = None
    timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )
    metadata: dict[str, Any] = field(default_factory=dict)

```

```
    @property
    def total_method_calls(self) -> int:
        """Total number of method calls during execution."""
        return sum(m.call_count for m in self.method_calls)

    @property
    def dispensary_method_calls(self) -> int:
        """Number of calls to dispensary methods."""
        return sum(m.call_count for m in self.method_calls if m.is_dispensary_method)

    @property
    def dispensary_usage_ratio(self) -> float:
        """Ratio of dispensary calls to total calls."""
        total = self.total_method_calls
        return self.dispensary_method_calls / total if total > 0 else 0.0

    @property
    def unique_dispensaries_used(self) -> set[str]:
        """Set of unique dispensary classes used."""
        return {m.class_name for m in self.method_calls if m.is_dispensary_method}

    @property
    def average_method_time_ms(self) -> float:
        """Average method execution time."""
        if not self.method_calls:
            return 0.0
        return sum(m.execution_time_ms for m in self.method_calls) / len(
            self.method_calls
        )

    @property
    def slowest_method(self) -> MethodCallMetrics | None:
        """Identify the slowest method call."""
        if not self.method_calls:
            return None
        return max(self.method_calls, key=lambda m: m.execution_time_ms)

    @property
    def memory_intensive_method(self) -> MethodCallMetrics | None:
        """Identify the most memory-intensive method call."""
        if not self.method_calls:
            return None
        return max(self.method_calls, key=lambda m: abs(m.memory_delta_mb))

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        data = asdict(self)
        data["method_calls"] = [m.to_dict() for m in self.method_calls]
        data["total_method_calls"] = self.total_method_calls
        data["dispensary_method_calls"] = self.dispensary_method_calls
        data["dispensary_usage_ratio"] = self.dispensary_usage_ratio
        data["unique_dispensaries_used"] = list(self.unique_dispensaries_used)
        data["average_method_time_ms"] = self.average_method_time_ms
        slowest = self.slowest_method
        data["slowest_method"] = (
            f"{slowest.class_name}.{slowest.method_name}" if slowest else None
```

```
        )
        memory_intensive = self.memory_intensive_method
        data["memory_intensive_method"] = (
            f"{memory_intensive.class_name}.{memory_intensive.method_name}"
            if memory_intensive
            else None
        )
        return data


@dataclass
class PerformanceRegression:
    """Detected performance regression for an executor."""

    executor_id: str
    metric_name: str
    baseline_value: float
    current_value: float
    delta_percent: float
    severity: str
    threshold_exceeded: bool
    recommendation: str

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        return asdict(self)


@dataclass
class PerformanceReport:
    """Comprehensive performance report with bottleneck analysis.

    Enhanced with dispensary pattern analytics.
    """

    timestamp: str
    total_executors: int
    total_execution_time_ms: float
    total_memory_mb: float
    regressions: list[PerformanceRegression] = field(default_factory=list)
    bottlenecks: list[dict[str, Any]] = field(default_factory=list)
    summary: dict[str, Any] = field(default_factory=dict)
    executor_rankings: dict[str, list[str]] = field(default_factory=dict)
    dispensary_analytics: dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        data = asdict(self)
        data["regressions"] = [r.to_dict() for r in self.regressions]
        return data


class ExecutorProfiler:
    """Performance profiler with baseline management and regression detection.

    Tracks per-executor metrics including timing, memory, serialization overhead,
```

```
    and method call counts. Supports baseline comparison for regression detection
    and generates comprehensive performance reports.

    ENHANCED: Tracks method dispensary pattern usage for monolith reuse analysis.
    """

    def __init__(
        self,
        baseline_path: Path | str | None = None,
        auto_save_baseline: bool = False,
        memory_tracking: bool = True,
        track_dispensary_usage: bool = True,
        performance_thresholds: dict[str, float] | None = None,
    ) -> None:
        """Initialize the profiler.

        Args:
            baseline_path: Path to baseline metrics file (JSON)
            auto_save_baseline: Automatically update baseline after each run
            memory_tracking: Enable memory tracking (adds overhead)
            track_dispensary_usage: Track dispensary class usage patterns
            performance_thresholds: Performance thresholds from canonical config
                                  (execution_time_ms, memory_mb, serialization_ms)
                                  If None, uses defaults from method_parameters.json
        """
        self.baseline_path = Path(baseline_path) if baseline_path else None
        self.auto_save_baseline = auto_save_baseline
        self.memory_tracking = memory_tracking
        self.track_dispensary_usage = track_dispensary_usage

        # Load thresholds from canonical config or use defaults
        self.thresholds = performance_thresholds or self._load_default_thresholds()

        self.metrics: dict[str, list[ExecutorMetrics]] = defaultdict(list)
        self.baseline_metrics: dict[str, ExecutorMetrics] = {}
        self.regressions: list[PerformanceRegression] = []

        # Dispensary usage tracking
        self.dispensary_call_counts: dict[str, int] = defaultdict(int)
        self.dispensary_execution_times: dict[str, list[float]] = defaultdict(list)
        self.executor_dispensary_usage: dict[str, set[str]] = defaultdict(set)

    def _load_default_thresholds(self) -> dict[str, float]:
        """Load default thresholds (can be overridden by canonical config)."""
        return {
            "execution_time_ms": DEFAULT_HIGH_EXECUTION_TIME_MS,
            "memory_mb": DEFAULT_HIGH_MEMORY_MB,
            "serialization_ms": DEFAULT_HIGH_SERIALIZATION_MS,
        }

        self._psutil = None
        self._psutil_process = None
        if memory_tracking:
            try:
                import psutil

```

```
                self._psutil = psutil
                self._psutil_process = psutil.Process()
            except ImportError:
                logger.warning(
                    "psutil not available, memory tracking disabled. "
                    "Install with: pip install psutil"
                )
                self.memory_tracking = False

        if self.baseline_path and self.baseline_path.exists():
            self.load_baseline(self.baseline_path)

    def _get_memory_usage_mb(self) -> float:
        """Get current memory usage in MB."""
        if not self.memory_tracking or not self._psutil_process:
            return 0.0
        try:
            return self._psutil_process.memory_info().rss / (1024 * 1024)
        except Exception as exc:
            logger.warning(f"Failed to get memory usage: {exc}")
            return 0.0

    def profile_executor(self, executor_id: str) -> ProfilerContext:
        """Create a profiling context for an executor.

        Args:
            executor_id: Unique executor identifier (e.g., "D1-Q1")

        Returns:
            ProfilerContext for use in with statement

        Example:
            with profiler.profile_executor("D1-Q1") as ctx:
                result = executor.execute(context)
                ctx.add_method_call("TextMiner", "extract", 45.2, 2.1)
        """
        return ProfilerContext(self, executor_id)

    def record_executor_metrics(
        self, executor_id: str, metrics: ExecutorMetrics
    ) -> None:
        """Record metrics for an executor execution.

        Args:
            executor_id: Unique executor identifier
            metrics: Collected metrics for the execution
        """
        self.metrics[executor_id].append(metrics)

        # Track dispensary usage
        if self.track_dispensary_usage:
            self._update_dispensary_stats(executor_id, metrics)

        if self.baseline_path and self.auto_save_baseline:
            self._update_baseline(executor_id, metrics)

```

```
    def _update_dispensary_stats(
        self, executor_id: str, metrics: ExecutorMetrics
    ) -> None:
        """Update dispensary usage statistics.

        Args:
            executor_id: Executor identifier
            metrics: Metrics containing method calls
        """
        for method_call in metrics.method_calls:
            if method_call.is_dispensary_method:
                class_name = method_call.class_name

                # Track call counts
                self.dispensary_call_counts[class_name] += method_call.call_count

                # Track execution times
                self.dispensary_execution_times[class_name].append(
                    method_call.execution_time_ms
                )

                # Track executorâ\206\222dispensary usage
                self.executor_dispensary_usage[executor_id].add(class_name)

    def _update_baseline(self, executor_id: str, metrics: ExecutorMetrics) -> None:
        """Update baseline with new metrics (running average).

        Args:
            executor_id: Executor identifier
            metrics: New metrics to incorporate
        """
        if executor_id not in self.baseline_metrics:
            self.baseline_metrics[executor_id] = metrics
        else:
            baseline = self.baseline_metrics[executor_id]
            baseline.execution_time_ms = (
                baseline.execution_time_ms * 0.8 + metrics.execution_time_ms * 0.2
            )
            baseline.memory_footprint_mb = (
                baseline.memory_footprint_mb * 0.8 + metrics.memory_footprint_mb * 0.2
            )
            baseline.serialization_time_ms = (
                baseline.serialization_time_ms * 0.8
                + metrics.serialization_time_ms * 0.2
            )
            baseline.call_count += 1

    def detect_regressions(
        self,
        thresholds: dict[str, float] | None = None,
    ) -> list[PerformanceRegression]:
        """Detect performance regressions against baseline.

        Args:
            thresholds: Regression thresholds for each metric
                       (default: {"execution_time_ms": 20.0, "memory_footprint_mb": 30.0})
```

```
        Returns:
            List of detected regressions
        """
        if thresholds is None:
            thresholds = {
                "execution_time_ms": 20.0,
                "memory_footprint_mb": 30.0,
                "serialization_time_ms": 50.0,
            }

        regressions: list[PerformanceRegression] = []

        for executor_id, metric_list in self.metrics.items():
            if not metric_list:
                continue

            if executor_id not in self.baseline_metrics:
                continue

            baseline = self.baseline_metrics[executor_id]
            current = metric_list[-1]

            for metric_name, threshold in thresholds.items():
                baseline_val = getattr(baseline, metric_name, 0.0)
                current_val = getattr(current, metric_name, 0.0)

                if baseline_val == 0:
                    continue

                delta_percent = ((current_val - baseline_val) / baseline_val) * 100

                if delta_percent > threshold:
                    severity = (
                        "critical" if delta_percent > threshold * 2 else "warning"
                    )
                    recommendation = self._generate_recommendation(
                        executor_id, metric_name, delta_percent, current
                    )

                    regression = PerformanceRegression(
                        executor_id=executor_id,
                        metric_name=metric_name,
                        baseline_value=baseline_val,
                        current_value=current_val,
                        delta_percent=delta_percent,
                        severity=severity,
                        threshold_exceeded=True,
                        recommendation=recommendation,
                    )
                    regressions.append(regression)

        self.regressions = regressions
        return regressions

    def _generate_recommendation(
```

```
        self,
        executor_id: str,
        metric_name: str,
        delta_percent: float,
        metrics: ExecutorMetrics | None = None,
    ) -> str:
        """Generate optimization recommendation for a regression with dispensary awareness."""
        base_recommendations = {
            "execution_time_ms": (
                f"Executor {executor_id} execution time increased by {delta_percent:.1f}%. "
            ),
            "memory_footprint_mb": (
                f"Executor {executor_id} memory usage increased by {delta_percent:.1f}%. "
            ),
            "serialization_time_ms": (
                f"Executor {executor_id} serialization overhead increased by {delta_percent:.1f}%. "
            ),
        }

        recommendation = base_recommendations.get(
            metric_name,
            f"Performance degradation detected in {metric_name} ({delta_percent:.1f}%)",
        )

        # Add dispensary-specific suggestions
        if metrics and self.track_dispensary_usage:
            if metric_name == "execution_time_ms" and metrics.slowest_method:
                slowest = metrics.slowest_method
                if slowest.is_dispensary_method:
                    shared_count = len(
                        [
                            eid
                            for eid, dispensaries in self.executor_dispensary_usage.items()
                            if slowest.class_name in dispensaries
                        ]
                    )
                    recommendation += (
                        f"Bottleneck in dispensary method {slowest.full_method_name} "
                        f"({slowest.execution_time_ms:.1f}ms). "
                        f"Consider optimizing this method as it’s shared across "
                        f"{shared_count} executors."
                    )
                else:
                    recommendation += f"Review method call sequence or optimize {slowest.full_method_name}."
            elif metric_name == "memory_footprint_mb":
                recommendation += "Check for memory leaks, optimize data structures, or implement streaming."
            elif metric_name == "serialization_time_ms":
                recommendation += "Reduce result payload size or use more efficient serialization format."

        return recommendation

    def identify_bottlenecks(self, top_n: int = 10) -> list[dict[str, Any]]:
        """Identify top bottleneck executors requiring optimization.

        Args:
            top_n: Number of top bottlenecks to return
```

```

        Returns:
            List of bottleneck descriptors with metrics and recommendations
        """
        bottlenecks: list[dict[str, Any]] = []

        for executor_id, metric_list in self.metrics.items():
            if not metric_list:
                continue

            avg_metrics = self._compute_average_metrics(metric_list)

            bottleneck_score = (
                avg_metrics["execution_time_ms"] * 0.5
                + avg_metrics["memory_footprint_mb"] * 0.3
                + avg_metrics["serialization_time_ms"] * 0.2
            )

            bottleneck = {
                "executor_id": executor_id,
                "bottleneck_score": bottleneck_score,
                "avg_execution_time_ms": avg_metrics["execution_time_ms"],
                "avg_memory_mb": avg_metrics["memory_footprint_mb"],
                "avg_serialization_ms": avg_metrics["serialization_time_ms"],
                "total_method_calls": avg_metrics["total_method_calls"],
                "dispensary_usage_ratio": avg_metrics.get(
                    "dispensary_usage_ratio", 0.0
                ),
                "unique_dispensaries": list(
                    self.executor_dispensary_usage.get(executor_id, set())
                ),
                "slowest_method": avg_metrics["slowest_method"],
                "memory_intensive_method": avg_metrics["memory_intensive_method"],
                "recommendation": self._generate_bottleneck_recommendation(
                    executor_id, avg_metrics
                ),
            }
            bottlenecks.append(bottleneck)

        bottlenecks.sort(key=lambda x: x["bottleneck_score"], reverse=True)
        return bottlenecks[:top_n]

    def _compute_average_metrics(
        self, metric_list: list[ExecutorMetrics]
    ) -> dict[str, Any]:
        """Compute average metrics from a list of executor metrics."""
        if not metric_list:
            return {}

        return {
            "execution_time_ms": sum(m.execution_time_ms for m in metric_list)
            / len(metric_list),
            "memory_footprint_mb": sum(m.memory_footprint_mb for m in metric_list)
            / len(metric_list),
            "serialization_time_ms": sum(m.serialization_time_ms for m in metric_list)
            / len(metric_list),
```

```
            "total_method_calls": sum(m.total_method_calls for m in metric_list)
            / len(metric_list),
            "dispensary_usage_ratio": sum(m.dispensary_usage_ratio for m in metric_list)
            / len(metric_list),
            "slowest_method": (
                metric_list[-1].slowest_method.class_name
                + "."
                + metric_list[-1].slowest_method.method_name
                if metric_list[-1].slowest_method
                else None
            ),
            "memory_intensive_method": (
                metric_list[-1].memory_intensive_method.class_name
                + "."
                + metric_list[-1].memory_intensive_method.method_name
                if metric_list[-1].memory_intensive_method
                else None
            ),
        }

    def _generate_bottleneck_recommendation(
        self, _executor_id: str, avg_metrics: dict[str, Any]
    ) -> str:
        """Generate optimization recommendation for a bottleneck with dispensary awareness."""
        recommendations = []

        if avg_metrics["execution_time_ms"] > self.thresholds["execution_time_ms"]:
            slowest = avg_metrics["slowest_method"]
            if slowest and any(
                dispensary in slowest for dispensary in KNOWN_DISPENSARY_CLASSES
            ):
                # Extract class name
                class_name = slowest.split(".")[0]
                shared_count = len(
                    [
                        eid
                        for eid, dispensaries in self.executor_dispensary_usage.items()
                        if class_name in dispensaries
                    ]
                )
                recommendations.append(
                    f"High execution time ({avg_metrics['execution_time_ms']:.1f}ms): "
                    f"dispensary method {slowest} shared by {shared_count} executors - "
                    f"optimization here benefits multiple executors"
                )
            else:
                recommendations.append(
                    f"High execution time ({avg_metrics['execution_time_ms']:.1f}ms): "
                    f"optimize {slowest or 'slow methods'}"
                )

        if avg_metrics["memory_footprint_mb"] > self.thresholds["memory_mb"]:
            recommendations.append(
                f"High memory usage ({avg_metrics['memory_footprint_mb']:.1f}MB): "
                f"review {avg_metrics['memory_intensive_method'] or 'data structures'}"
            )
```

```
        if avg_metrics["serialization_time_ms"] > self.thresholds["serialization_ms"]:
            recommendations.append(
                f"High serialization overhead ({avg_metrics['serialization_time_ms']:.1f}ms): "
                "reduce payload size"
            )

        if not recommendations:
            return "Performance acceptable, monitor for regressions"

        return "; ".join(recommendations)

    def get_dispensary_usage_stats(self) -> dict[str, Any]:
        """Get comprehensive dispensary usage statistics."""
        if not self.track_dispensary_usage:
            return {
                "tracking_enabled": False,
                "message": "Dispensary tracking disabled. Enable with track_dispensary_usage=True",
            }

        dispensary_stats = {}

        for dispensary_class in KNOWN_DISPENSARY_CLASSES:
            if dispensary_class not in self.dispensary_call_counts:
                continue

            call_count = self.dispensary_call_counts[dispensary_class]
            exec_times = self.dispensary_execution_times.get(dispensary_class, [])

            avg_time = sum(exec_times) / len(exec_times) if exec_times else 0.0
            total_time = sum(exec_times)

            # Find which executors use this dispensary
            using_executors = [
                eid
                for eid, dispensaries in self.executor_dispensary_usage.items()
                if dispensary_class in dispensaries
            ]

            dispensary_stats[dispensary_class] = {
                "total_calls": call_count,
                "avg_execution_time_ms": avg_time,
                "total_execution_time_ms": total_time,
                "used_by_executor_count": len(using_executors),
                "using_executors": using_executors,
                "reuse_factor": call_count / max(len(using_executors), 1),
            }

        # Sort by total execution time
        sorted_dispensaries = sorted(
            dispensary_stats.items(),
            key=lambda x: x[1]["total_execution_time_ms"],
            reverse=True,
        )

        return {
```

```
            "tracking_enabled": True,
            "total_dispensaries_used": len(dispensary_stats),
            "total_dispensary_calls": sum(self.dispensary_call_counts.values()),
            "dispensaries": dict(sorted_dispensaries),
            "hottest_dispensaries": [
                {
                    "class": name,
                    "total_time_ms": stats["total_execution_time_ms"],
                    "avg_time_ms": stats["avg_execution_time_ms"],
                    "executor_count": stats["used_by_executor_count"],
                    "reuse_factor": stats["reuse_factor"],
                }
                for name, stats in sorted_dispensaries[:5]
            ],
        }

    def generate_report(
        self, include_regressions: bool = True, include_bottlenecks: bool = True
    ) -> PerformanceReport:
        """Generate comprehensive performance report.

        Args:
            include_regressions: Include regression detection
            include_bottlenecks: Include bottleneck analysis

        Returns:
            PerformanceReport with analysis and recommendations
        """
        regressions = []
        if include_regressions:
            regressions = self.detect_regressions()

        bottlenecks = []
        if include_bottlenecks:
            bottlenecks = self.identify_bottlenecks()

        total_execution_time = sum(
            m.execution_time_ms for metrics in self.metrics.values() for m in metrics
        )
        total_memory = sum(
            m.memory_footprint_mb for metrics in self.metrics.values() for m in metrics
        )

        executor_rankings = {
            "slowest": self._rank_executors_by("execution_time_ms"),
            "memory_intensive": self._rank_executors_by("memory_footprint_mb"),
            "serialization_heavy": self._rank_executors_by("serialization_time_ms"),
        }

        summary = {
            "total_executors_profiled": len(self.metrics),
            "total_executions": sum(len(m) for m in self.metrics.values()),
            "regressions_detected": len(regressions),
            "critical_regressions": sum(
                1 for r in regressions if r.severity == "critical"
            ),
```

```
            "bottlenecks_identified": len(bottlenecks),
            "avg_execution_time_ms": total_execution_time
            / max(1, sum(len(m) for m in self.metrics.values())),
            "avg_memory_mb": total_memory
            / max(1, sum(len(m) for m in self.metrics.values())),
        }

        # Add dispensary analytics to report
        dispensary_analytics = {}
        if self.track_dispensary_usage:
            dispensary_analytics = self.get_dispensary_usage_stats()

        return PerformanceReport(
            timestamp=datetime.now(timezone.utc).isoformat(),
            total_executors=len(self.metrics),
            total_execution_time_ms=total_execution_time,
            total_memory_mb=total_memory,
            regressions=regressions,
            bottlenecks=bottlenecks,
            summary=summary,
            executor_rankings=executor_rankings,
            dispensary_analytics=dispensary_analytics,
        )

    def _rank_executors_by(self, metric_name: str, top_n: int = 10) -> list[str]:
        """Rank executors by a specific metric.

        Args:
            metric_name: Metric to rank by
            top_n: Number of top executors to return

        Returns:
            List of executor IDs ranked by metric
        """
        rankings = []
        for executor_id, metric_list in self.metrics.items():
            if not metric_list:
                continue
            avg_value = sum(getattr(m, metric_name, 0.0) for m in metric_list) / len(
                metric_list
            )
            rankings.append((executor_id, avg_value))

        rankings.sort(key=lambda x: x[1], reverse=True)
        return [executor_id for executor_id, _ in rankings[:top_n]]

    def save_baseline(self, path: Path | str | None = None) -> None:
        """Save current metrics as baseline.

        Args:
            path: Path to save baseline (uses self.baseline_path if None)
        """
        path = Path(path) if path else self.baseline_path
        if not path:
            raise ValueError("No baseline path specified")

```

```
        path.parent.mkdir(parents=True, exist_ok=True)

        baseline_data = {
            executor_id: metrics.to_dict()
            for executor_id, metrics in self.baseline_metrics.items()
        }

        with open(path, "w", encoding="utf-8") as f:
            json.dump(baseline_data, f, indent=2)

        logger.info(f"Baseline saved to {path}")

    def load_baseline(self, path: Path | str) -> None:
        """Load baseline metrics from file.

        Args:
            path: Path to baseline file
        """
        path = Path(path)
        if not path.exists():
            logger.warning(f"Baseline file not found: {path}")
            return

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

        for executor_id, data in baseline_data.items():
            method_calls = [
                MethodCallMetrics(**m) for m in data.pop("method_calls", [])
            ]
            # Remove computed properties before reconstructing
            data.pop("total_method_calls", None)
            data.pop("dispensary_method_calls", None)
            data.pop("dispensary_usage_ratio", None)
            data.pop("unique_dispensaries_used", None)
            data.pop("average_method_time_ms", None)
            data.pop("slowest_method", None)
            data.pop("memory_intensive_method", None)

            metrics = ExecutorMetrics(**data, method_calls=method_calls)
            self.baseline_metrics[executor_id] = metrics

        logger.info(
            f"Baseline loaded from {path}: {len(self.baseline_metrics)} executors"
        )

    def export_report(
        self, report: PerformanceReport, path: Path | str, format: str = "json"
    ) -> None:
        """Export performance report to file.

        Args:
            report: Performance report to export
            path: Output path
            format: Output format ("json", "markdown", or "html")
        """
```

```
        path = Path(path)
        path.parent.mkdir(parents=True, exist_ok=True)

        if format == "json":
            with open(path, "w", encoding="utf-8") as f:
                json.dump(report.to_dict(), f, indent=2)

        elif format == "markdown":
            self._export_markdown(report, path)

        elif format == "html":
            self._export_html(report, path)

        else:
            raise ValueError(f"Unsupported format: {format}")

        logger.info(f"Report exported to {path}")

    def _export_markdown(self, report: PerformanceReport, path: Path) -> None:
        """Export report as Markdown."""
        lines = [
            "# Executor Performance Report",
            f"**Generated:** {report.timestamp}",
            "",
            "## Summary",
            f"- **Total Executors:** {report.total_executors}",
            f"- **Total Execution Time:** {report.total_execution_time_ms:.2f}ms",
            f"- **Total Memory:** {report.total_memory_mb:.2f}MB",
            f"- **Regressions Detected:** {report.summary.get('regressions_detected', 0)}",
            f"- **Bottlenecks Identified:** {report.summary.get('bottlenecks_identified', 0)}",
            "",
        ]

        # Dispensary analytics section
        if report.dispensary_analytics.get("tracking_enabled"):
            lines.extend(
                [
                    "## Dispensary Usage Analytics",
                    "",
                    f"- **Total Dispensaries Used:** {report.dispensary_analytics.get('total_dispensaries_used', 0)}",
                    f"- **Total Dispensary Calls:** {report.dispensary_analytics.get('total_dispensary_calls', 0)}",
                    "",
                    "### Hottest Dispensaries",
                    "",
                    "| Rank | Class | Total Time (ms) | Avg Time (ms) | Executors | Reuse Factor |",
                    "|------|-------|----------------|---------------|-----------|--------------|",
                ]
            )

            for i, disp in enumerate(
                report.dispensary_analytics.get("hottest_dispensaries", []), 1
            ):
                lines.append(
                    f"| {i} | {disp['class']} | {disp['total_time_ms']:.1f} | "
                    f"{disp['avg_time_ms']:.1f} | {disp['executor_count']} | "
                    f"{disp['reuse_factor']:.1f} |"
```

```
                )
            lines.append("")

        if report.regressions:
            lines.extend(
                [
                    "## Performance Regressions",
                    "",
                    "| Executor | Metric | Baseline | Current | Delta | Severity |",
                    "|----------|--------|----------|---------|-------|----------|",
                ]
            )
            for reg in report.regressions:
                lines.append(
                    f"| {reg.executor_id} | {reg.metric_name} | "
                    f"{reg.baseline_value:.2f} | {reg.current_value:.2f} | "
                    f"{reg.delta_percent:+.1f}% | {reg.severity} |"
                )
            lines.append("")

        if report.bottlenecks:
            lines.extend(
                [
                    "## Top Bottlenecks",
                    "",
                    "| Rank | Executor | Score | Exec Time | Memory | Dispensaries | Recommendation |",
                    "|------|----------|-------|-----------|--------|--------------|----------------|",
                ]
            )
            for i, bottleneck in enumerate(report.bottlenecks[:10], 1):
                disp_count = len(bottleneck.get("unique_dispensaries", []))
                lines.append(
                    f"| {i} | {bottleneck['executor_id']} | "
                    f"{bottleneck['bottleneck_score']:.1f} | "
                    f"{bottleneck['avg_execution_time_ms']:.1f}ms | "
                    f"{bottleneck['avg_memory_mb']:.1f}MB | "
                    f"{disp_count} | "
                    f"{bottleneck['recommendation'][:60]}... |"
                )
            lines.append("")

        with open(path, "w", encoding="utf-8") as f:
            f.write("\n".join(lines))

    def _export_html(self, report: PerformanceReport, path: Path) -> None:
        """Export report as HTML."""
        html = f"""<!DOCTYPE html>
<html>
<head>
    <title>Executor Performance Report</title>
    <style>
        body {{ font-family: Arial, sans-serif; margin: 20px; }}
        h1, h2 {{ color: #333; }}
        table {{ border-collapse: collapse; width: 100%; margin: 20px 0; }}
        th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}
        th {{ background-color: #4CAF50; color: white; }}
```

```
        .critical {{ color: red; font-weight: bold; }}
        .warning {{ color: orange; font-weight: bold; }}
    </style>
</head>
<body>
    <h1>Executor Performance Report</h1>
    <p><strong>Generated:</strong> {report.timestamp}</p>

    <h2>Summary</h2>
    <ul>
        <li><strong>Total Executors:</strong> {report.total_executors}</li>
        <li><strong>Total Execution Time:</strong> {report.total_execution_time_ms:.2f}ms</li>
        <li><strong>Total Memory:</strong> {report.total_memory_mb:.2f}MB</li>
        <li><strong>Regressions Detected:</strong> {report.summary.get('regressions_detected', 0)}</li>
        <li><strong>Bottlenecks Identified:</strong> {report.summary.get('bottlenecks_identified', 0)}</li>
    </ul>
"""

        # Add dispensary analytics
        if report.dispensary_analytics.get("tracking_enabled"):
            html += """
    <h2>Dispensary Usage Analytics</h2>
    <table>
        <tr>
            <th>Rank</th>
            <th>Dispensary Class</th>
            <th>Total Time (ms)</th>
            <th>Avg Time (ms)</th>
            <th>Executor Count</th>
            <th>Reuse Factor</th>
        </tr>
"""

            for i, disp in enumerate(
                report.dispensary_analytics.get("hottest_dispensaries", []), 1
            ):
                html += f"""
        <tr>
            <td>{i}</td>
            <td>{disp['class']}</td>
            <td>{disp['total_time_ms']:.1f}</td>
            <td>{disp['avg_time_ms']:.1f}</td>
            <td>{disp['executor_count']}</td>
            <td>{disp['reuse_factor']:.1f}</td>
        </tr>
"""
            html += "    </table>\n"

        if report.regressions:
            html += """
    <h2>Performance Regressions</h2>
    <table>
        <tr>
            <th>Executor</th>
            <th>Metric</th>
            <th>Baseline</th>
```

```
            <th>Current</th>
            <th>Delta</th>
            <th>Severity</th>
        </tr>
"""
            for reg in report.regressions:
                severity_class = reg.severity
                html += f"""
        <tr>
            <td>{reg.executor_id}</td>
            <td>{reg.metric_name}</td>
            <td>{reg.baseline_value:.2f}</td>
            <td>{reg.current_value:.2f}</td>
            <td>{reg.delta_percent:+.1f}%</td>
            <td class="{severity_class}">{reg.severity}</td>
        </tr>
"""
            html += "    </table>\n"

        if report.bottlenecks:
            html += """
    <h2>Top Bottlenecks</h2>
    <table>
        <tr>
            <th>Rank</th>
            <th>Executor</th>
            <th>Score</th>
            <th>Exec Time</th>
            <th>Memory</th>
            <th>Recommendation</th>
        </tr>
"""
            for i, bottleneck in enumerate(report.bottlenecks[:10], 1):
                html += f"""
        <tr>
            <td>{i}</td>
            <td>{bottleneck['executor_id']}</td>
            <td>{bottleneck['bottleneck_score']:.1f}</td>
            <td>{bottleneck['avg_execution_time_ms']:.1f}ms</td>
            <td>{bottleneck['avg_memory_mb']:.1f}MB</td>
            <td>{bottleneck['recommendation']}</td>
        </tr>
"""
            html += "    </table>\n"

        html += """
</body>
</html>
"""
        with open(path, "w", encoding="utf-8") as f:
            f.write(html)

    def clear_metrics(self) -> None:
        """Clear all collected metrics (but not baseline)."""
        self.metrics.clear()
        self.regressions.clear()
```

```
        self.dispensary_call_counts.clear()
        self.dispensary_execution_times.clear()
        self.executor_dispensary_usage.clear()


class ProfilerContext:
    """Context manager for automatic executor profiling.

    Automatically captures timing, memory, and serialization metrics
    when used with a 'with' statement.
    """

    def __init__(self, profiler: ExecutorProfiler, executor_id: str) -> None:
        """Initialize profiler context.

        Args:
            profiler: Parent profiler instance
            executor_id: Executor being profiled
        """
        self.profiler = profiler
        self.executor_id = executor_id
        self.start_time: float = 0.0
        self.start_memory: float = 0.0
        self.method_calls: list[MethodCallMetrics] = []
        self.result: Any = None
        self.error: str | None = None

    def __enter__(self) -> ProfilerContext:
        """Enter profiling context."""
        self.start_time = time.perf_counter()
        self.start_memory = self.profiler._get_memory_usage_mb()
        gc.collect()
        return self

    def __exit__(
        self,
        exc_type: type[BaseException] | None,
        exc_val: BaseException | None,
        exc_tb: object,
    ) -> None:
        """Exit profiling context and record metrics."""
        execution_time = (time.perf_counter() - self.start_time) * 1000
        end_memory = self.profiler._get_memory_usage_mb()
        memory_footprint = end_memory - self.start_memory
        memory_peak = max(end_memory, self.start_memory)

        serialization_time, serialization_size = self._measure_serialization()

        metrics = ExecutorMetrics(
            executor_id=self.executor_id,
            execution_time_ms=execution_time,
            memory_footprint_mb=memory_footprint,
            memory_peak_mb=memory_peak,
            serialization_time_ms=serialization_time,
            serialization_size_bytes=serialization_size,
            method_calls=self.method_calls,
```

```
            success=exc_type is None,
            error=str(exc_val) if exc_val else None,
        )

        self.profiler.record_executor_metrics(self.executor_id, metrics)

    def _measure_serialization(self) -> tuple[float, int]:
        """Measure serialization overhead for the result.

        Returns:
            Tuple of (serialization_time_ms, serialization_size_bytes)
        """
        if self.result is None:
            return 0.0, 0

        try:
            start = time.perf_counter()
            serialized = pickle.dumps(self.result, protocol=pickle.HIGHEST_PROTOCOL)
            serialization_time = (time.perf_counter() - start) * 1000
            serialization_size = len(serialized)
            return serialization_time, serialization_size
        except Exception as exc:
            logger.warning(f"Failed to measure serialization: {exc}")
            return 0.0, 0

    def add_method_call(
        self,
        class_name: str,
        method_name: str,
        execution_time_ms: float,
        memory_delta_mb: float = 0.0,
        success: bool = True,
        error: str | None = None,
    ) -> None:
        """Add a method call to the profiling context.

        Args:
            class_name: Class of the method
            method_name: Name of the method
            execution_time_ms: Execution time in milliseconds
            memory_delta_mb: Memory delta in MB
            success: Whether the call succeeded
            error: Error message if failed
        """
        metrics = MethodCallMetrics(
            class_name=class_name,
            method_name=method_name,
            execution_time_ms=execution_time_ms,
            memory_delta_mb=memory_delta_mb,
            success=success,
            error=error,
        )
        self.method_calls.append(metrics)

    def set_result(self, result: object) -> None:
        """Set the result for serialization measurement.
```

```

        Args:
            result: Execution result (can be any serializable object)
        """
        self.result = result


__all__ = [
    "ExecutorProfiler",
    "ProfilerContext",
    "ExecutorMetrics",
    "MethodCallMetrics",
    "PerformanceRegression",
    "PerformanceReport",
    "KNOWN_DISPENSARY_CLASSES",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/executors.py
================================================================================

"""
executors.py - Phase 2: Executor Orchestration for Policy Document Analysis

This module defines 30 executors (one per D{n}-Q{m} question) that orchestrate
methods from the core module to extract raw evidence from Colombian municipal
development plans (PDET/PDM documents).

Architecture:
- Each executor is independent and receives a canonical context package
- Methods execute in configured order; any failure causes executor failure
- Outputs are Python dicts/lists matching JSON contract specifications
- Executors are injected via MethodExecutor factory pattern

Usage:
    from factory import run_executor
    result = run_executor("D1-Q1", context_package)
"""

from __future__ import annotations

import sys
import logging
from typing import Dict, List, Any, Optional
from abc import ABC, abstractmethod
from dataclasses import dataclass

from farfan_pipeline.core.canonical_notation import CanonicalDimension, get_dimension_info
from farfan_pipeline.core.orchestrator.core import MethodExecutor
from farfan_pipeline.core.orchestrator.factory import build_processor
from farfan_pipeline.core.orchestrator.memory_safety import (
    MemorySafetyGuard,
    MemorySafetyConfig,
    ExecutorType,
```

```
    create_default_guard,
)
from farfan_pipeline.processing.policy_processor import CausalDimension

logger = logging.getLogger(__name__)

# Canonical question labels (only defined when verified in repo)
CANONICAL_QUESTION_LABELS = {
    "D3-Q2": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
    "D3-Q3": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
    "D3-Q4": "DIM03_Q04_TECHNICAL_FEASIBILITY",
    "D3-Q5": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
    "D4-Q1": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
    "D5-Q2": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
}

# Epistemic taxonomy per method (focused on executors expanded in this iteration)
EPISTEMIC_TAGS = {
    ("FinancialAuditor", "_calculate_sufficiency"): ["statistical", "normative"],
    ("FinancialAuditor", "_match_program_to_node"): ["structural"],
    ("FinancialAuditor", "_match_goal_to_budget"): ["structural", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability"): ["financial", "normative"],
    ("PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility"): ["financial", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_score_indicators"): ["normative", "semantic"],
    ("PDETMunicipalPlanAnalyzer", "_interpret_risk"): ["normative", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_consolidate_entities"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_syntax"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_ner"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "identify_responsible_entities"): ["semantic", "structural"],
    ("PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities"): ["statistical", "causal"],
    ("PDETMunicipalPlanAnalyzer", "construct_causal_dag"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "estimate_causal_effects"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "generate_counterfactuals"): ["causal"],
    ("PDETMunicipalPlanAnalyzer", "_identify_confounders"): ["causal", "consistency"],
    ("PDETMunicipalPlanAnalyzer", "_effect_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_scenario_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords"): ["semantic"],
    ("AdaptivePriorCalculator", "calculate_likelihood_adaptativo"): ["statistical", "bayesian"],
    ("AdaptivePriorCalculator", "_adjust_domain_weights"): ["statistical"],
    ("BayesianMechanismInference", "_test_sufficiency"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_test_necessity"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_log_refactored_components"): ["implementation"],
    ("BayesianMechanismInference", "_infer_activity_sequence"): ["causal"],
    ("BayesianMechanismInference", "infer_mechanisms"): ["causal", "bayesian"],
    ("AdvancedDAGValidator", "calculate_acyclicity_pvalue"): ["statistical", "consistency"],
    ("AdvancedDAGValidator", "_is_acyclic"): ["structural", "consistency"],
    ("AdvancedDAGValidator", "_calculate_bayesian_posterior"): ["statistical", "bayesian"],
    ("AdvancedDAGValidator", "_calculate_confidence_interval"): ["statistical"],
    ("AdvancedDAGValidator", "_calculate_statistical_power"): ["statistical"],
    ("AdvancedDAGValidator", "_generate_subgraph"): ["structural"],
    ("AdvancedDAGValidator", "_get_node_validator"): ["implementation"],
    ("AdvancedDAGValidator", "_create_empty_result"): ["descriptive"],
    ("AdvancedDAGValidator", "_initialize_rng"): ["implementation"],
    ("AdvancedDAGValidator", "get_graph_stats"): ["structural"],
```

```
    ("AdvancedDAGValidator", "_calculate_node_importance"): ["structural"],
    ("AdvancedDAGValidator", "export_nodes"): ["structural", "descriptive"],
    ("AdvancedDAGValidator", "add_node"): ["structural"],
    ("AdvancedDAGValidator", "add_edge"): ["structural"],
    ("IndustrialGradeValidator", "execute_suite"): ["implementation", "normative"],
    ("IndustrialGradeValidator", "validate_connection_matrix"): ["consistency"],
    ("IndustrialGradeValidator", "run_performance_benchmarks"): ["implementation"],
    ("IndustrialGradeValidator", "_benchmark_operation"): ["implementation"],
    ("IndustrialGradeValidator", "validate_causal_categories"): ["consistency"],
    ("IndustrialGradeValidator", "_log_metric"): ["implementation"],
    ("PerformanceAnalyzer", "analyze_performance"): ["implementation", "normative"],
    ("PerformanceAnalyzer", "_calculate_loss_functions"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_ess"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_likelihood"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_r_hat"): ["statistical"],
    ("ReportingEngine", "generate_accountability_matrix"): ["normative", "structural"],
    ("ReportingEngine", "_calculate_quality_score"): ["normative", "statistical"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "descriptive"],
    ("PolicyAnalysisEmbedder", "compare_policy_interventions"): ["normative"],
    ("PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency"): ["consistency", "statistical"],
    ("PolicyAnalysisEmbedder", "process_document"): ["semantic", "structural"],
    ("PolicyAnalysisEmbedder", "semantic_search"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_apply_mmr"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_generate_query_from_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_filter_by_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_extract_numerical_values"): ["statistical"],
    ("PolicyAnalysisEmbedder", "_compute_overall_confidence"): ["statistical", "normative"],
    ("PolicyAnalysisEmbedder", "_embed_texts"): ["semantic"],
    ("SemanticAnalyzer", "_classify_policy_domain"): ["semantic"],
    ("SemanticAnalyzer", "_empty_semantic_cube"): ["descriptive"],
    ("SemanticAnalyzer", "_classify_cross_cutting_themes"): ["semantic"],
    ("SemanticAnalyzer", "_classify_value_chain_link"): ["semantic"],
    ("SemanticAnalyzer", "_vectorize_segments"): ["semantic"],
    ("SemanticAnalyzer", "_calculate_semantic_complexity"): ["semantic"],
    ("SemanticAnalyzer", "_process_segment"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_entity_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_quality_to_dict"): ["descriptive", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_deduplicate_tables"): ["structural", "implementation"],
    ("PDETMunicipalPlanAnalyzer", "_indicator_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_generate_recommendations"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_simulate_intervention"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_identify_causal_nodes"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "_match_text_to_node"): ["semantic", "structural"],
    ("TeoriaCambio", "_validar_orden_causal"): ["causal", "consistency"],
    ("TeoriaCambio", "_generar_sugerencias_internas"): ["normative"],
    ("TeoriaCambio", "_extraer_categorias"): ["semantic"],
    ("BayesianMechanismInference", "_extract_observations"): ["semantic", "causal"],
    ("BayesianMechanismInference", "_generate_necessity_remediation"): ["normative", "causal"],
    ("BayesianMechanismInference", "_quantify_uncertainty"): ["statistical", "bayesian"],
    ("CausalExtractor", "_build_type_hierarchy"): ["structural"],
    ("CausalExtractor", "_check_structural_violation"): ["structural", "consistency"],
    ("CausalExtractor", "_calculate_type_transition_prior"): ["statistical", "bayesian"],
    ("CausalExtractor", "_calculate_textual_proximity"): ["semantic"],
    ("CausalExtractor", "_calculate_language_specificity"): ["semantic"],
    ("CausalExtractor", "_calculate_composite_likelihood"): ["statistical", "semantic"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
```

```
    ("CausalExtractor", "_calculate_semantic_distance"): ["semantic"],
    ("CausalExtractor", "_extract_goals"): ["semantic"],
    ("CausalExtractor", "_parse_goal_context"): ["semantic"],
    ("CausalExtractor", "_classify_goal_type"): ["semantic"],
    ("TemporalLogicVerifier", "_parse_temporal_marker"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_classify_temporal_type"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_extract_resources"): ["structural"],
    ("TemporalLogicVerifier", "_should_precede"): ["temporal", "consistency"],
    ("AdaptivePriorCalculator", "generate_traceability_record"): ["structural", "semantic"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "normative"],
    ("ReportingEngine", "generate_confidence_report"): ["normative", "descriptive"],
    ("PolicyTextProcessor", "segment_into_sentences"): ["semantic", "structural"],
    ("PolicyTextProcessor", "normalize_unicode"): ["implementation"],
    ("PolicyTextProcessor", "compile_pattern"): ["implementation"],
    ("PolicyTextProcessor", "extract_contextual_window"): ["semantic"],
    ("BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize"): ["causal", "normative"],
    ("BayesianCounterfactualAuditor", "refutation_and_sanity_checks"): ["causal", "consistency"],
    ("BayesianCounterfactualAuditor", "_evaluate_factual"): ["causal", "statistical"],
    ("BayesianCounterfactualAuditor", "_evaluate_counterfactual"): ["causal", "statistical"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("IndustrialPolicyProcessor", "_load_questionnaire"): ["descriptive", "implementation"],
    ("IndustrialPolicyProcessor", "_compile_pattern_registry"): ["structural", "semantic"],
    ("IndustrialPolicyProcessor", "_build_point_patterns"): ["semantic"],
    ("IndustrialPolicyProcessor", "_empty_result"): ["implementation"],
    ("IndustrialPolicyProcessor", "_compute_evidence_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_compute_avg_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_construct_evidence_bundle"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "generate_executive_report"): ["normative"],
    ("IndustrialPolicyProcessor", "export_results"): ["implementation"],
}


class BaseExecutor(ABC):
    """
    Base class for all executors with standardized execution template.
    All executors must implement execute() and return structured evidence.

    Includes systematic memory safety guards for processing large objects.
    """

    def __init__(self, executor_id: str, config: Dict[str, Any], method_executor: MethodExecutor, profiler: Any | None = None):
        self.executor_id = executor_id
        self.config = config
        if not isinstance(method_executor, MethodExecutor):
            raise RuntimeError("A valid MethodExecutor instance is required for executor injection.")
        self.method_executor = method_executor
        self.execution_log = []
        self._dimension_info = None

        memory_config = config.get("memory_safety", {})
        if isinstance(memory_config, dict):
            self.memory_guard = MemorySafetyGuard(MemorySafetyConfig(**memory_config))
        else:
            self.memory_guard = create_default_guard()

        self.executor_type = self._determine_executor_type()
```

```
        self._profiler = profiler
        self._profiler_context: Any | None = None

    def _determine_executor_type(self) -> ExecutorType:
        """Determine memory limit type based on executor ID and primary methods."""
        executor_id_lower = self.executor_id.lower()

        if "d3-q2" in executor_id_lower or "d3-q3" in executor_id_lower:
            return ExecutorType.ENTITY
        elif "d6-q1" in executor_id_lower or "d6-q2" in executor_id_lower:
            return ExecutorType.DAG
        elif "d5-q" in executor_id_lower or "d6-q3" in executor_id_lower:
            return ExecutorType.CAUSAL_EFFECTS
        elif "d1-q" in executor_id_lower or "d2-q" in executor_id_lower:
            return ExecutorType.SEMANTIC
        elif "d3-q" in executor_id_lower or "d4-q" in executor_id_lower:
            return ExecutorType.FINANCIAL
        else:
            return ExecutorType.GENERIC

    @property
    def dimension_info(self):
        """Lazy-loaded dimension information to avoid redundant metadata fetches."""
        if self._dimension_info is None:
            try:
                dim_key = self.executor_id.split("-")[0].replace("D", "D")
                self._dimension_info = get_dimension_info(dim_key)
            except (KeyError, ValueError, IndexError) as e:
                logger.warning(f"Failed to load dimension info for {self.executor_id}: {e}")
                self._dimension_info = None
        return self._dimension_info

    def _safe_process_object(self, obj: Any, label: str = "object") -> Any:
        """Process object with memory safety guards.

        Args:
            obj: Object to process (entities, DAG, causal effects, etc.)
            label: Human-readable label for logging

        Returns:
            Memory-safe processed object
        """
        processed_obj, metrics = self.memory_guard.check_and_process(
            obj, self.executor_type, label
        )

        if metrics.was_truncated or metrics.was_sampled:
            logger.info(
                f"{self.executor_id}: Applied {metrics.fallback_strategy} to {label} - "
                f"size reduced from {metrics.object_size_bytes / (1024*1024):.2f}MB "
                f"({metrics.elements_before} elements) to "
                f"{metrics.json_size_bytes / (1024*1024):.2f}MB "
                f"({metrics.elements_after} elements)"
            )

        return processed_obj
```

```
    def _safe_process_list(self, items: List[Any], label: str = "list") -> List[Any]:
        """Process list with memory safety guards."""
        return self._safe_process_object(items, label)

    def _safe_process_dict(self, data: Dict[str, Any], label: str = "dict") -> Dict[str, Any]:
        """Process dict with memory safety guards."""
        return self._safe_process_object(data, label)

    def _get_memory_metrics_summary(self) -> Dict[str, Any]:
        """Get summary of memory operations for this executor."""
        return self.memory_guard.get_metrics_summary()

    def _validate_context(self, context: Dict[str, Any]) -> None:
        """
        Fail fast on malformed contexts.

        Raises:
            ValueError: If required context keys are missing
        """
        if not isinstance(context, dict):
            raise ValueError(f"Context must be a dict, got {type(context).__name__}")

        required = ["document_text"]
        missing = [k for k in required if k not in context]
        if missing:
            raise ValueError(f"Context missing required keys: {missing}")

    @abstractmethod
    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute configured methods and return raw evidence.

        Args:
            context: Canonical package with document, tables, metadata

        Returns:
            Dict with raw_evidence, metadata, execution_metrics

        Raises:
            ExecutorFailure: If any method fails
        """
        pass

    def _log_method_execution(self, class_name: str, method_name: str,
                              success: bool, result: Any = None, error: str = None):
        """Track method execution for debugging and traceability."""
        self.execution_log.append({
            "class": class_name,
            "method": method_name,
            "success": success,
            "result_type": type(result).__name__ if result else None,
            "error": error
        })

    def _execute_method(self, class_name: str, method_name: str,
```

```
                       context: Dict[str, Any], **kwargs) -> Any:
        """
        Execute a single method with error handling and profiling.

        Raises:
            ExecutorFailure: If method execution fails
        """
        import time
        start_time = time.perf_counter()
        start_memory = 0.0

        if self._profiler and self._profiler.memory_tracking:
            start_memory = self._profiler._get_memory_usage_mb()

        try:
            method = self._get_method(class_name, method_name)
            result = method(context, **kwargs)

            execution_time = (time.perf_counter() - start_time) * 1000
            memory_delta = 0.0
            if self._profiler and self._profiler.memory_tracking:
                memory_delta = self._profiler._get_memory_usage_mb() - start_memory

            self._log_method_execution(class_name, method_name, True, result)

            if self._profiler_context:
                self._profiler_context.add_method_call(
                    class_name, method_name, execution_time, memory_delta, True
                )

            return result
        except Exception as e:
            execution_time = (time.perf_counter() - start_time) * 1000
            memory_delta = 0.0
            if self._profiler and self._profiler.memory_tracking:
                memory_delta = self._profiler._get_memory_usage_mb() - start_memory

            self._log_method_execution(class_name, method_name, False, error=str(e))

            if self._profiler_context:
                self._profiler_context.add_method_call(
                    class_name, method_name, execution_time, memory_delta, False, str(e)
                )

            raise ExecutorFailure(
                f"Executor {self.executor_id} failed: {class_name}.{method_name} - {str(e)}"
            ) from e

    def _get_method(self, class_name: str, method_name: str):
        """Retrieve method using MethodExecutor to enforce routed execution."""
        if not isinstance(self.method_executor, MethodExecutor):
            raise RuntimeError(f"Invalid method executor provided: {type(self.method_executor).__name__}")

        def _wrapped(context: Dict[str, Any], **kwargs: Any) -> Any:
            payload: Dict[str, Any] = {}
            if context:
```

```
                payload.update(context)
            if kwargs:
                payload.update(kwargs)
            return self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

        return _wrapped

    def execute_with_profiling(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """Execute with automatic profiling if profiler is attached.

        Args:
            context: Canonical package with document, tables, metadata

        Returns:
            Dict with raw_evidence, metadata, execution_metrics
        """
        if self._profiler:
            with self._profiler.profile_executor(self.executor_id) as ctx:
                self._profiler_context = ctx
                try:
                    result = self.execute(context)
                    ctx.set_result(result)
                    return result
                finally:
                    self._profiler_context = None
        else:
            return self.execute(context)


@dataclass
class ExecutorResult:
    """
    Standardized result container for executor execution.
    Ensures type safety and verifiable structure.
    """
    executor_id: str
    success: bool
    data: Optional[Dict[str, Any]]
    error: Optional[str]
    execution_time_ms: int
    memory_usage_mb: float


class ExecutorFailure(Exception):
    """Raised when any method in an executor fails."""
    pass


# ============================================================================
# DIMENSION 1: DIAGNOSTICS & INPUTS
# ============================================================================

```

```
class D1_Q1_QuantitativeBaselineExtractor(BaseExecutor):
    """
    Extracts numeric data, reference years, and official sources as baseline.

    Methods (from D1-Q1):
    - TextMiningEngine.diagnose_critical_links
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor.process
    - IndustrialPolicyProcessor._match_patterns_in_sentences
    - IndustrialPolicyProcessor._extract_point_evidence
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
    - FinancialAuditor._parse_amount
    - PDETMunicipalPlanAnalyzer._extract_financial_amounts
    - PDETMunicipalPlanAnalyzer._extract_from_budget_table
    - PolicyContradictionDetector._extract_quantitative_claims
    - PolicyContradictionDetector._parse_number
    - PolicyContradictionDetector._statistical_significance_test
    - BayesianNumericalAnalyzer.evaluate_policy_metric
    - BayesianNumericalAnalyzer.compare_policies
    - SemanticProcessor.chunk_text
    - SemanticProcessor.embed_single
    """

    def execute(self, context: Dict[str, Any] | None = None, **kwargs: Any) -> Dict[str, Any]:
        if context is None:
            context = dict(kwargs)
        raw_evidence: Dict[str, Any] = {}

        # The new implementation requires manual instantiation of some components
        # because the dependency injection logic was part of the old MethodExecutor.
        # This will be revisited when refactoring the executor base class.
        ontology = self.method_executor.shared_instances.get("MunicipalOntology")

        # Step 0: Initial processing of the document text
        raw_text = context.get("raw_text", "")
        sentences = self._execute_method("PolicyTextProcessor", "segment_into_sentences", context, text=raw_text)

        # Step 1: Semantic and Performance Analysis (Prerequisites for TextMiningEngine)
        semantic_cube = self._execute_method("SemanticAnalyzer", "extract_semantic_cube", context, document_segments=sentences)
        performance_analysis = self._execute_method("PerformanceAnalyzer", "analyze_performance", context, semantic_cube=semantic_cube)

        # Step 2: Identify critical data-bearing sections
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context,
            semantic_cube=semantic_cube,
            performance_analysis=performance_analysis
        )

        # The output of diagnose_critical_links is complex. Let’s assume it contains segments for _analyze_link_text
        link_analysis_segments = critical_links.get("critical_links", {}).get(next(iter(critical_links.get("critical_links", {})), None), {}).get("text_analysis", {}).get("keywords", [])
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context,
            segments=[{"text": s} for s in link_analysis_segments] # _analyze_link_text expects list of dicts
        )
```

```

        # Step 3: Extract structured quantitative claims from the whole document
        processed_sections = self._execute_method(
            "IndustrialPolicyProcessor", "process", context,
            raw_text=raw_text
        )
        # We need compiled patterns. This is a challenge. Let’s assume the processor can get them.
        compiled_patterns = self._execute_method("IndustrialPolicyProcessor", "_compile_pattern_registry", context)

        pattern_matches, _ = self._execute_method(
            "IndustrialPolicyProcessor", "_match_patterns_in_sentences", context,
            compiled_patterns=compiled_patterns.get(CausalDimension.D1_INSUMOS, {}).get('diagnostico_cuantitativo', []),
            relevant_sentences=sentences
        )

        point_evidence_list = []
        for point_code in self._execute_method("IndustrialPolicyProcessor", "_build_point_patterns", context):
             point_evidence = self._execute_method(
                "IndustrialPolicyProcessor", "_extract_point_evidence", context,
                text=raw_text,
                sentences=sentences,
                point_code=point_code
            )
             point_evidence_list.append(point_evidence)

        # Step 4: Parse numerical amounts and baseline data
        all_text = " ".join(pattern_matches)
        parsed_amounts = self._execute_method( "FinancialAuditor", "_parse_amount", context, value=all_text)

        financial_amounts = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_financial_amounts", context,
            text=raw_text, tables=context.get("tables", [])
        )

        # This method needs a dataframe, which is not available in the context.
        # I will skip this call for now.
        # budget_table_data = self._execute_method(
        #     "PDETMunicipalPlanAnalyzer", "_extract_from_budget_table", context
        # )
        budget_table_data = None

        # Step 5: Extract temporal context (reference years)
        goals = self._execute_method( "CausalExtractor", "_extract_goals", context, text=raw_text)

        goal_contexts = []
        if isinstance(goals, list):
            for goal in goals:
                goal_id = goal.id
                goal_context_str = goal.text
                if goal_id and goal_context_str:
                    res = self._execute_method(
                        "CausalExtractor", "_parse_goal_context", context,
                        goal_id=goal_id,
                        context=goal_context_str
                    )
                    if res:
```

```
                        goal_contexts.append(res)

        # Step 6: Validate quantitative claims
        quant_claims = self._execute_method( "PolicyContradictionDetector", "_extract_quantitative_claims", context, text=raw_text)

        parsed_numbers = []
        if isinstance(quant_claims, list):
            for claim in quant_claims:
                res = self._execute_method(
                    "PolicyContradictionDetector", "_parse_number", context,
                    text=claim.get("raw_text")
                )
                if res is not None:
                    parsed_numbers.append(res)

        significance_test = None
        if len(parsed_numbers) >= 2:
            # The method expects claims, not just numbers. Let’s create dummy claims.
            claim_a = {’value’: parsed_numbers[0]}
            claim_b = {’value’: parsed_numbers[1]}
            significance_test = self._execute_method(
                "PolicyContradictionDetector", "_statistical_significance_test", context,
                claim_a=claim_a,
                claim_b=claim_b
            )

        # Step 7: Evaluate baseline quality and compare
        metric_evaluation = self._execute_method(
            "BayesianNumericalAnalyzer", "evaluate_policy_metric", context,
            observed_values=parsed_numbers
        )

        policy_comparison = None
        if metric_evaluation and "posterior_samples" in metric_evaluation:
             policy_comparison = self._execute_method(
                "BayesianNumericalAnalyzer", "compare_policies", context,
                policy_a_values=[s[’coherence’] for s in metric_evaluation.get("posterior_samples", [])],
                policy_b_values=context.get("baseline_samples", []) # Assuming baseline samples exist
            )

        # Step 8: Semantic validation of sources
        text_chunks = self._execute_method( "SemanticProcessor", "chunk_text", context, text=raw_text, preserve_structure=True)

        embeddings = []
        if isinstance(text_chunks, list):
            texts_to_embed = [chunk[’content’] for chunk in text_chunks if ’content’ in chunk]
            embeddings = self._execute_method( "SemanticProcessor", "_embed_batch", context, texts=texts_to_embed)

        # Assemble raw evidence
        raw_evidence = {
            "numeric_data": parsed_numbers,
            "reference_years": [gc.year for gc in goal_contexts if gc and hasattr(gc, ’year’)],
            "official_sources": point_evidence_list,
            "financial_baseline": financial_amounts,
            "budget_tables": budget_table_data,
            "significance_results": significance_test,
```

```
            "metric_evaluation": metric_evaluation,
            "policy_comparison": policy_comparison,
            "goal_contexts": [res.text if res else None for res in goal_contexts],
            "quantitative_claims": quant_claims,
            "processed_sections": processed_sections,
            "pattern_matches": pattern_matches,
            "link_analysis": link_analysis,
            "source_embeddings": embeddings
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_numeric_claims": len(parsed_numbers or []),
                "sources_identified": len(point_evidence_list)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q2_ProblemDimensioningAnalyzer(BaseExecutor):
    """
    Quantifies problem magnitude, gaps, and identifies data limitations.

    Methods (from D1-Q2):
    - OperationalizationAuditor._audit_direct_evidence
    - OperationalizationAuditor._audit_systemic_risk
    - FinancialAuditor._detect_allocation_gaps
    - BayesianMechanismInference._detect_gaps
    - PDETMunicipalPlanAnalyzer._generate_optimal_remediations
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - BayesianCounterfactualAuditor.counterfactual_query
    - BayesianCounterfactualAuditor._test_effect_stability
    - PolicyContradictionDetector._detect_numerical_inconsistencies
    - PolicyContradictionDetector._calculate_numerical_divergence
    - BayesianConfidenceCalculator.calculate_posterior
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any] | None = None, **kwargs: Any) -> Dict[str, Any]:
        if context is None:
            context = dict(kwargs)
        raw_evidence: Dict[str, Any] = {}

        # Step 1: Audit evidence completeness
        direct_evidence_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_direct_evidence", context
        )
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )
```

```

        # Step 2: Detect gaps in resource allocation and mechanisms
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )
        mechanism_gaps = self._execute_method(
            "BayesianMechanismInference", "_detect_gaps", context
        )

        # Step 3: Generate optimal remediations and simulate interventions
        remediations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_optimal_remediations", context,
            gaps=allocation_gaps
        )
        simulation_results = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            remediations=remediations
        )

        # Step 4: Counterfactual analysis for problem dimensioning
        counterfactual = self._execute_method(
            "BayesianCounterfactualAuditor", "counterfactual_query", context
        )
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            counterfactual=counterfactual
        )

        # Step 5: Detect numerical inconsistencies
        numerical_inconsistencies = self._execute_method(
            "PolicyContradictionDetector", "_detect_numerical_inconsistencies", context
        )
        divergence_calc = self._execute_method(
            "PolicyContradictionDetector", "_calculate_numerical_divergence", context,
            inconsistencies=numerical_inconsistencies
        )

        # Step 6: Calculate confidence and analyze performance
        posterior_confidence = self._execute_method(
            "BayesianConfidenceCalculator", "calculate_posterior", context,
            evidence=direct_evidence_audit
        )
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "magnitude_indicators": {
                "allocation_gaps": allocation_gaps,
                "mechanism_gaps": mechanism_gaps,
                "numerical_inconsistencies": numerical_inconsistencies
            },
            "deficit_quantification": divergence_calc,
            "counterfactual_analysis": counterfactual,
            "effect_stability": effect_stability,
            "data_limitations": {
```

```
                "evidence_gaps": direct_evidence_audit.get("gaps", []),
                "systemic_risks": systemic_risk_audit
            },
            "simulation_results": simulation_results,
            "confidence_scores": posterior_confidence,
            "performance_metrics": performance_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "gaps_identified": len(allocation_gaps or []) + len(mechanism_gaps or []),
                "inconsistencies_found": len(numerical_inconsistencies or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q3_BudgetAllocationTracer(BaseExecutor):
    """
    Traces monetary resources assigned to programs in Investment Plan (PPI).

    Methods (from D1-Q3):
    - FinancialAuditor.trace_financial_allocation
    - FinancialAuditor._process_financial_table
    - FinancialAuditor._match_program_to_node
    - FinancialAuditor._match_goal_to_budget
    - FinancialAuditor._perform_counterfactual_budget_check
    - FinancialAuditor._calculate_sufficiency
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer._extract_budget_for_pillar
    - PDETMunicipalPlanAnalyzer._identify_funding_source
    - PDETMunicipalPlanAnalyzer._classify_tables
    - PDETMunicipalPlanAnalyzer._analyze_funding_sources
    - PDETMunicipalPlanAnalyzer._score_financial_component
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Trace complete financial allocation chain
        allocation_trace = self._execute_method(
            "FinancialAuditor", "trace_financial_allocation", context
        )
        processed_tables = self._execute_method(
            "FinancialAuditor", "_process_financial_table", context
        )

        # Step 2: Match programs to budget nodes
        program_matches = self._execute_method(
```

```
            "FinancialAuditor", "_match_program_to_node", context,
            tables=processed_tables
        )
        goal_budget_matches = self._execute_method(
            "FinancialAuditor", "_match_goal_to_budget", context,
            programs=program_matches
        )

        # Step 3: Counterfactual checks and sufficiency calculation
        counterfactual_check = self._execute_method(
            "FinancialAuditor", "_perform_counterfactual_budget_check", context,
            matches=goal_budget_matches
        )
        sufficiency_calc = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context,
            allocation=allocation_trace
        )

        # Step 4: Analyze financial feasibility
        feasibility_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        pillar_budgets = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_budget_for_pillar", context
        )
        funding_sources = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_funding_source", context
        )

        # Step 5: Classify and analyze tables
        table_classification = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
            tables=processed_tables
        )
        funding_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
            sources=funding_sources
        )
        financial_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
            analysis=funding_analysis
        )

        # Step 6: Aggregate risk and prioritize
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            sufficiency=sufficiency_calc,
            counterfactual=counterfactual_check
        )

        raw_evidence = {
            "budget_allocations": allocation_trace,
            "program_mappings": program_matches,
            "goal_budget_links": goal_budget_matches,
            "counterfactual_budget_check": counterfactual_check,
            "sufficiency_analysis": sufficiency_calc,
```

```
            "pillar_budgets": pillar_budgets,
            "funding_sources": funding_sources,
            "financial_feasibility": feasibility_analysis,
            "financial_score": financial_score,
            "table_classification": table_classification,
            "funding_analysis": funding_analysis,
            "risk_priorities": risk_aggregation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "programs_traced": len(program_matches or []),
                "funding_sources_identified": len(funding_sources or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q4_InstitutionalCapacityIdentifier(BaseExecutor):
    """
    Identifies installed capacity (entities, staff, equipment) and limitations.

    Methods (from D1-Q4):
    - PDETMunicipalPlanAnalyzer.identify_responsible_entities
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._classify_entity_type
    - PDETMunicipalPlanAnalyzer._score_entity_specificity
    - PDETMunicipalPlanAnalyzer._consolidate_entities
    - MechanismPartExtractor.extract_entity_activity
    - MechanismPartExtractor._normalize_entity
    - MechanismPartExtractor._validate_entity_activity
    - MechanismPartExtractor._calculate_ea_confidence
    - OperationalizationAuditor.audit_evidence_traceability
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Identify all responsible entities
        entities_identified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
        )

        # Step 2: Extract entities using NER and syntax
        ner_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context
        )
        syntax_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context
```

```
        )

        # Step 3: Classify and score entities
        entity_types = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_entity_type", context,
            entities=ner_entities + syntax_entities
        )
        specificity_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_entity_specificity", context,
            entities=entity_types
        )
        consolidated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
            entities=entity_types
        )

        # Step 4: Extract entity-activity relationships
        entity_activities = self._execute_method(
            "MechanismPartExtractor", "extract_entity_activity", context,
            entities=consolidated
        )
        normalized = self._execute_method(
            "MechanismPartExtractor", "_normalize_entity", context,
            activities=entity_activities
        )
        validated = self._execute_method(
            "MechanismPartExtractor", "_validate_entity_activity", context,
            normalized=normalized
        )
        ea_confidence = self._execute_method(
            "MechanismPartExtractor", "_calculate_ea_confidence", context,
            validated=validated
        )

        # Step 5: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            entity_activities=validated
        )

        raw_evidence = {
            "entities_identified": consolidated,
            "entity_types": entity_types,
            "specificity_scores": specificity_scores,
            "entity_activities": validated,
            "activity_confidence": ea_confidence,
            "capacity_indicators": {
                "staff_mentions": [e for e in consolidated if e.get("type") == "staff"],
                "equipment_mentions": [e for e in consolidated if e.get("type") == "equipment"],
                "organizational_units": [e for e in consolidated if e.get("type") == "organization"]
            },
            "limitations_identified": (traceability_audit or {}).get("gaps", []),
            "traceability_audit": traceability_audit
        }

        return {
```

```
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "entities_count": len(consolidated or []),
                "activities_extracted": len(validated or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q5_ScopeJustificationValidator(BaseExecutor):
    """
    Validates scope justification via legal framework and constraint recognition.

    Methods (from D1-Q5):
    - TemporalLogicVerifier._check_deadline_constraints
    - TemporalLogicVerifier.verify_temporal_consistency
    - CausalInferenceSetup.identify_failure_points
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - IndustrialPolicyProcessor._extract_metadata
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Verify temporal constraints
        deadline_constraints = self._execute_method(
            "TemporalLogicVerifier", "_check_deadline_constraints", context
        )
        temporal_consistency = self._execute_method(
            "TemporalLogicVerifier", "verify_temporal_consistency", context
        )

        # Step 2: Identify failure points in scope
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 3: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 4: Analyze link text for justifications
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context
        )

        # Step 5: Analyze causal dimensions and extract metadata
        causal_dimensions = self._execute_method(
```

```
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )
        metadata_extracted = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context,
            dimensions=causal_dimensions
        )

        raw_evidence = {
            "legal_framework_citations": metadata_extracted.get("legal_refs", []),
            "temporal_constraints": {
                "deadline_checks": deadline_constraints,
                "consistency": temporal_consistency,
                "coherence": temporal_coherence
            },
            "budgetary_constraints": metadata_extracted.get("budget_limits", []),
            "competence_constraints": metadata_extracted.get("competence_refs", []),
            "failure_points": failure_points,
            "scope_justifications": (link_analysis or {}).get("justifications", []),
            "causal_dimensions": causal_dimensions
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "constraints_identified": len(deadline_constraints or []),
                "legal_citations": len(metadata_extracted.get("legal_refs", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 2: ACTIVITY DESIGN
# ============================================================================

class D2_Q1_StructuredPlanningValidator(BaseExecutor):
    """
    Validates structured format of activities (table/matrix with required columns).

    Methods (from D2-Q1):
    - PDFProcessor.extract_tables
    - FinancialAuditor._process_financial_table
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._classify_tables
    - PDETMunicipalPlanAnalyzer._is_likely_header
    - PDETMunicipalPlanAnalyzer._clean_dataframe
    - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
```

```

        # Step 1: Extract all tables
        extracted_tables = self._execute_method(
            "PDFProcessor", "extract_tables", context
        )

        # Step 2: Process financial tables
        processed_tables = self._execute_method(
            "FinancialAuditor", "_process_financial_table", context,
            tables=extracted_tables
        )

        # Step 3: Deduplicate and classify tables
        deduplicated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=processed_tables
        )
        classified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
            tables=deduplicated
        )

        # Step 4: Identify headers and clean dataframes
        header_checks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_is_likely_header", context,
            tables=classified
        )
        cleaned = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_clean_dataframe", context,
            tables=classified
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
            tables=cleaned
        )

        raw_evidence = {
            "tables_extracted": len(extracted_tables),
            "activity_tables": [t for t in classified if t.get("type") == "activity"],
            "matrix_structure": accountability_matrix,
            "required_columns_present": {
                "responsible_entity": any("responsible" in str(t.get("columns", [])).lower()
                                         for t in cleaned),
                "deliverable": any("deliverable" in str(t.get("columns", [])).lower()
                                  for t in cleaned),
                "timeline": any("timeline" in str(t.get("columns", [])).lower()
                               for t in cleaned),
                "cost": any("cost" in str(t.get("columns", [])).lower()
                           for t in cleaned)
            },
            "table_quality": {
                "clean_tables": len(cleaned),
                "with_headers": sum(1 for h in header_checks if h)
            }
```

```
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_tables": len(extracted_tables),
                "activity_tables": len([t for t in classified if t.get("type") == "activity"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q2_InterventionLogicInferencer(BaseExecutor):
    """
    Infers intervention logic: instrument (how), target (who), causality (why).

    Methods (from D2-Q2):
    - BayesianMechanismInference.infer_mechanisms
    - BayesianMechanismInference._infer_single_mechanism
    - BayesianMechanismInference._infer_mechanism_type
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - CausalExtractor.extract_causal_hierarchy
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio._es_conexion_valida
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - BeachEvidentialTest.classify_test
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context
        )
        single_mechanisms = []
        for mech in mechanisms:
            single = self._execute_method(
                "BayesianMechanismInference", "_infer_single_mechanism", context,
                mechanism=mech
            )
            single_mechanisms.append(single)

        mechanism_types = self._execute_method(
            "BayesianMechanismInference", "_infer_mechanism_type", context,
            mechanisms=single_mechanisms
        )

        # Step 2: Test sufficiency and necessity
```

```
        sufficiency_tests = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context,
            mechanisms=single_mechanisms
        )
        necessity_tests = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context,
            mechanisms=single_mechanisms
        )

        # Step 3: Extract causal hierarchy
        causal_hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        # Step 4: Build causal graph
        causal_graph = self._execute_method(
            "TeoriaCambio", "construir_grafo_causal", context,
            hierarchy=causal_hierarchy
        )
        connection_validation = self._execute_method(
            "TeoriaCambio", "_es_conexion_valida", context,
            graph=causal_graph
        )

        # Step 5: Construct DAG
        causal_dag = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
            graph=causal_graph
        )

        # Step 6: Classify evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "classify_test", context,
            mechanisms=single_mechanisms
        )

        # Step 7: Analyze causal dimensions
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )

        raw_evidence = {
            "intervention_instruments": [m.get("instrument") for m in single_mechanisms],
            "target_populations": [m.get("target") for m in single_mechanisms],
            "causal_logic": {
                "mechanisms": single_mechanisms,
                "mechanism_types": mechanism_types,
                "sufficiency": sufficiency_tests,
                "necessity": necessity_tests
            },
            "causal_hierarchy": causal_hierarchy,
            "causal_graph": causal_graph,
            "causal_dag": causal_dag,
            "evidential_strength": evidential_tests,
            "connection_validation": connection_validation,
            "dimensions": causal_dimensions
```

```
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(single_mechanisms or []),
                "instruments_found": len([m for m in single_mechanisms if m.get("instrument")]),
                "connections_valid": bool(connection_validation)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q3_RootCauseLinkageAnalyzer(BaseExecutor):
    """
    Analyzes linkage between activities and root causes/structural determinants.

    Methods (from D2-Q3):
    - CausalExtractor._extract_causal_links
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._initialize_prior
    - CausalExtractor._calculate_type_transition_prior
    - PDETMunicipalPlanAnalyzer._identify_causal_edges
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - BayesianCounterfactualAuditor.construct_scm
    - BayesianCounterfactualAuditor._create_default_equations
    - SemanticAnalyzer.extract_semantic_cube
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract causal links
        causal_links = self._execute_method(
            "CausalExtractor", "_extract_causal_links", context
        )

        # Step 2: Calculate likelihoods
        composite_likelihood = self._execute_method(
            "CausalExtractor", "_calculate_composite_likelihood", context,
            links=causal_links
        )
        prior_init = self._execute_method(
            "CausalExtractor", "_initialize_prior", context
        )
        type_transition_prior = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            links=causal_links
        )

        # Step 3: Identify and refine causal edges
```

```
        causal_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_edges", context,
            links=causal_links
        )
        refined_probabilities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            edges=causal_edges
        )

        # Step 4: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            edges=refined_probabilities
        )
        default_equations = self._execute_method(
            "BayesianCounterfactualAuditor", "_create_default_equations", context,
            scm=scm
        )

        # Step 5: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        raw_evidence = {
            "root_causes_identified": [link.get("root_cause") for link in (causal_links or [])],
            "activity_linkages": causal_links,
            "link_probabilities": refined_probabilities,
            "composite_likelihood": composite_likelihood,
            "prior_initialization": prior_init,
            "type_transition_prior": type_transition_prior,
            "structural_model": scm,
            "model_equations": default_equations,
            "semantic_relationships": semantic_cube,
            "determinants_addressed": [link for link in (causal_links or []) if link.get("addresses_determinant")]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "causal_links_found": len(causal_links or []),
                "root_causes_count": len(set(link.get("root_cause") for link in (causal_links or [])))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q4_RiskManagementAnalyzer(BaseExecutor):
    """
    Identifies implementation risks and mitigation measures.

```

```
    Methods (from D2-Q4):
    - PDETMunicipalPlanAnalyzer._bayesian_risk_inference
    - PDETMunicipalPlanAnalyzer.sensitivity_analysis
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Bayesian risk inference
        risk_inference = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_bayesian_risk_inference", context
        )

        # Step 2: Sensitivity analysis
        sensitivity = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "sensitivity_analysis", context,
            risks=risk_inference
        )

        # Step 3: Risk interpretation
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            inference=risk_inference
        )

        # Step 4: Compute robustness metrics
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            sensitivity=sensitivity
        )
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            robustness=robustness
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            sensitivity=sensitivity
        )

        # Step 5: Audit systemic risks
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 6: Aggregate and prioritize risks
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            risks=risk_inference
```

```
        )

        # Step 7: Refutation and sanity checks
        refutation_checks = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            aggregation=risk_aggregation
        )

        # Step 8: Additional sensitivity analysis
        adaptive_sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=risk_inference
        )

        raw_evidence = {
            "operational_risks": [r for r in risk_inference if r.get("type") == "operational"],
            "social_risks": [r for r in risk_inference if r.get("type") == "social"],
            "security_risks": [r for r in risk_inference if r.get("type") == "security"],
            "mitigation_measures": (risk_interpretation or {}).get("mitigations", []),
            "risk_priorities": risk_aggregation,
            "robustness_metrics": {
                "robustness_value": robustness,
                "e_value": e_value
            },
            "sensitivity_analysis": sensitivity,
            "systemic_risks": systemic_risk_audit,
            "validation_checks": refutation_checks,
            "sensitivity_interpretation": sensitivity_interpretation,
            "adaptive_sensitivity": adaptive_sensitivity,
            "risk_interpretation": risk_interpretation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "risks_identified": len(risk_inference or []),
                "mitigations_proposed": len((risk_interpretation or {}).get("mitigations", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q5_StrategicCoherenceEvaluator(BaseExecutor):
    """
    Evaluates strategic coherence: complementarity and logical sequence.

    Methods (from D2-Q5):
    - PolicyContradictionDetector._detect_logical_incompatibilities
    - PolicyContradictionDetector._calculate_coherence_metrics
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._calculate_graph_fragmentation
```

```
    - OperationalizationAuditor.audit_sequence_logic
    - BayesianMechanismInference._calculate_coherence_factor
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Detect logical incompatibilities
        incompatibilities = self._execute_method(
            "PolicyContradictionDetector", "_detect_logical_incompatibilities", context
        )

        # Step 2: Calculate coherence metrics
        coherence_metrics = self._execute_method(
            "PolicyContradictionDetector", "_calculate_coherence_metrics", context
        )
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )
        graph_fragmentation = self._execute_method(
            "PolicyContradictionDetector", "_calculate_graph_fragmentation", context
        )

        # Step 3: Audit sequence logic
        sequence_audit = self._execute_method(
            "OperationalizationAuditor", "audit_sequence_logic", context
        )

        # Step 4: Calculate coherence factors
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            metrics=coherence_metrics
        )
        causal_coherence_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 5: Adaptive likelihood calculation
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context,
            coherence=causal_coherence_score
        )

        raw_evidence = {
            "complementarity_evidence": coherence_metrics.get("complementarity", []),
            "sequential_logic": sequence_audit,
            "logical_incompatibilities": incompatibilities,
            "coherence_scores": {
                "overall_coherence": coherence_metrics,
                "objective_alignment": objective_alignment,
                "causal_coherence": causal_coherence_score,
                "coherence_factor": coherence_factor
            },
            "graph_metrics": {
```

```
                "fragmentation": graph_fragmentation
            },
            "adaptive_likelihood": adaptive_likelihood
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "incompatibilities_found": len(incompatibilities),
                "coherence_score": coherence_metrics.get("score", 0)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 3: PRODUCTS & OUTPUTS
# ============================================================================

class D3_Q1_IndicatorQualityValidator(BaseExecutor):
    """
    Validates indicator quality: baseline, target, source of verification.

    Methods (from D3-Q1):
    - PDETMunicipalPlanAnalyzer._score_indicators
    - OperationalizationAuditor.audit_evidence_traceability
    - CausalInferenceSetup.assign_probative_value
    - BeachEvidentialTest.apply_test_logic
    - TextMiningEngine.diagnose_critical_links
    - IndustrialPolicyProcessor._extract_metadata
    - IndustrialPolicyProcessor._calculate_quality_score
    - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score indicators
        indicator_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )

        # Step 2: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            indicators=indicator_scores
        )

        # Step 3: Assign probative value
        probative_values = self._execute_method(
            "CausalInferenceSetup", "assign_probative_value", context,
```

```
            indicators=indicator_scores
        )

        # Step 4: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context,
            indicators=indicator_scores
        )

        # Step 5: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 6: Extract and score metadata
        metadata = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context
        )
        quality_score = self._execute_method(
            "IndustrialPolicyProcessor", "_calculate_quality_score", context,
            metadata=metadata
        )

        # Step 7: Generate traceability record
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            indicators=indicator_scores
        )

        raw_evidence = {
            "indicators_with_baseline": [i for i in indicator_scores if i.get("has_baseline")],
            "indicators_with_target": [i for i in indicator_scores if i.get("has_target")],
            "indicators_with_source": [i for i in indicator_scores if i.get("has_source")],
            "indicator_quality_scores": indicator_scores,
            "traceability": traceability_audit,
            "probative_values": probative_values,
            "evidential_strength": evidential_tests,
            "critical_links": critical_links,
            "overall_quality_score": quality_score,
            "traceability_record": traceability_record
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_indicators": len(indicator_scores or []),
                "complete_indicators": len([i for i in indicator_scores
                    if i.get("has_baseline") and i.get("has_target") and i.get("has_source")]),
                "critical_links_assessed": len(critical_links or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
```

```
        }


class D3_Q2_TargetProportionalityAnalyzer(BaseExecutor):
    """
    DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY â\200\224 Analyzes proportionality of targets to the diagnosed universe using canonical D3 notation.
    Epistemic mix: structural coverage, financial/normative feasibility, statistical Bayes tests, and semantic indicator quality.

    Methods (from D3-Q2):
    - AdvancedDAGValidator._calculate_bayesian_posterior
    - AdvancedDAGValidator._calculate_confidence_interval
    - AdaptivePriorCalculator._adjust_domain_weights
    - PDETMunicipalPlanAnalyzer._get_spanish_stopwords
    - BayesianMechanismInference._log_refactored_components
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - FinancialAuditor._calculate_sufficiency
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - PDETMunicipalPlanAnalyzer._assess_financial_sustainability
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    - IndustrialPolicyProcessor._calculate_quality_score
    - TeoriaCambio._generar_sugerencias_internas
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._indicator_to_dict
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - IndustrialPolicyProcessor._compile_pattern_registry
    - IndustrialPolicyProcessor._build_point_patterns
    - IndustrialPolicyProcessor._empty_result
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Financial feasibility snapshot and indicator quality
        financial_feasibility = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        indicator_quality = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )
        spanish_stopwords = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords", context
        )
        funding_sources = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
            financial_indicators=financial_feasibility.get("financial_indicators", []),
            tables=context.get("tables", [])
        )
        financial_component = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
            financial_analysis=financial_feasibility
        )
        pattern_registry = self._execute_method(
```

```
            "IndustrialPolicyProcessor", "_compile_pattern_registry", context
        )
        point_patterns = self._execute_method(
            "IndustrialPolicyProcessor", "_build_point_patterns", context
        )
        empty_policy_result = self._execute_method(
            "IndustrialPolicyProcessor", "_empty_result", context
        )
        dedup_tables = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=context.get("tables", [])
        )
        # Type-safe indicator extraction: explicit None, not wrong-typed {}
        first_indicator = None
        if isinstance(financial_feasibility.get("financial_indicators", []), list):
            inds = financial_feasibility.get("financial_indicators", [])
            if inds and isinstance(inds[0], dict):
                first_indicator = inds[0]

        # Pass None explicitly when no indicator exists, maintaining type contract
        indicator_dict = None
        if first_indicator is not None:
            indicator_dict = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_indicator_to_dict", context,
                ind=first_indicator
            )
        proportionality_recommendations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={
                "financial_analysis": financial_feasibility,
                "quality_score": quality_score
            }
        )

        # Step 1: Calculate sufficiency
        sufficiency_calc = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )

        # Step 2: Test sufficiency and necessity of targets
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Assess financial sustainability
        sustainability_assessment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability", context
        )
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risk=financial_feasibility.get("risk_assessment", {}).get("risk_score", 0.0)
        )

```

```
        # Step 4: Calculate adaptive likelihood
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context
        )
        domain_scores = {
            "structural": sufficiency_calc.get("coverage_ratio", 0.0),
            "financial": financial_feasibility.get("sustainability_score", 0.0),
            "semantic": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0
        }
        adjusted_weights = self._execute_method(
            "AdaptivePriorCalculator", "_adjust_domain_weights", context,
            domain_scores=domain_scores
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D3": {"dimension_confidence": domain_scores.get("structural", 0.0)}}
        )

        # Step 6: Generate internal suggestions
        internal_suggestions = self._execute_method(
            "TeoriaCambio", "_generar_sugerencias_internas", context
        )
        # Bayesian posterior diagnostics for proportionality evidence
        posterior_probability = self._execute_method(
            "AdvancedDAGValidator", "_calculate_bayesian_posterior", context,
            likelihood=sufficiency_calc.get("coverage_ratio", 0.5),
            prior=0.5
        )
        confidence_interval = self._execute_method(
            "AdvancedDAGValidator", "_calculate_confidence_interval", context,
            s=int(sufficiency_calc.get("covered_targets", 0)),
            n=max(1, int(sufficiency_calc.get("targets_total", len(context.get("product_targets", []))))),
            conf=0.95
        )
        self._execute_method(
            "BayesianMechanismInference", "_log_refactored_components", context
        )

        raw_evidence = {
            "target_population_size": context.get("diagnosed_universe", 0),
            "product_targets": context.get("product_targets", []),
            "coverage_ratio": sufficiency_calc.get("coverage_ratio", 0),
            "dosage_analysis": sufficiency_calc.get("dosage", {}),
            "sufficiency_test": sufficiency_test,
            "necessity_test": necessity_test,
            "sustainability": sustainability_assessment,
            "financial_feasibility": financial_feasibility,
            "indicator_quality": indicator_quality,
            "risk_interpretation": risk_interpretation,
            "proportionality_score": quality_score,
            "recommendations": internal_suggestions,
            "stopwords_spanish": spanish_stopwords,
            "funding_sources_analysis": funding_sources,
            "financial_component_score": financial_component,
            "pattern_registry": pattern_registry,
            "point_patterns": point_patterns,
```

```
            "empty_policy_result": empty_policy_result,
            "avg_confidence": avg_confidence,
            "deduplicated_tables": dedup_tables,
            "indicator_sample": indicator_dict,
            "proportionality_recommendations": proportionality_recommendations,
            "adjusted_domain_weights": adjusted_weights,
            "posterior_proportionality": posterior_probability,
            "coverage_interval": confidence_interval
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "targets_analyzed": len(context.get("product_targets", [])),
                "coverage_adequate": sufficiency_calc.get("is_sufficient", False),
                "canonical_question": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q3_TraceabilityValidator(BaseExecutor):
    """
    DIM03_Q03_TRACEABILITY_BUDGET_ORG â\200\224 Validates budgetary and organizational traceability of products under canonical D3 notation.
    Epistemic mix: structural budget tracing, organizational semantics, and accountability synthesis.

    Methods executed (in order):
    Step 1: Budget matching - FinancialAuditor._match_program_to_node
    Step 2: Goal-budget matching - FinancialAuditor._match_goal_to_budget
    Step 3: Responsibility extraction - PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables
    Step 4: Entity consolidation - PDETMunicipalPlanAnalyzer._consolidate_entities
    Step 5: Entity identification - PDETMunicipalPlanAnalyzer.identify_responsible_entities
    Step 6: Clarity scoring - PDETMunicipalPlanAnalyzer._score_responsibility_clarity
    Step 7: Document processing - PolicyAnalysisEmbedder.process_document
    Step 8: Query generation - PolicyAnalysisEmbedder._generate_query_from_pdq
    Step 9: Semantic search - PolicyAnalysisEmbedder.semantic_search
    Step 10: MMR diversification - PolicyAnalysisEmbedder._apply_mmr
    Step 11: Semantic cube baseline - SemanticAnalyzer._empty_semantic_cube
    Step 12: Policy domain classification - SemanticAnalyzer._classify_policy_domain
    Step 13: Cross-cutting themes - SemanticAnalyzer._classify_cross_cutting_themes
    Step 14: Value chain classification - SemanticAnalyzer._classify_value_chain_link
    Step 15: Segment vectorization - SemanticAnalyzer._vectorize_segments
    Step 16: Segment processing - SemanticAnalyzer._process_segment
    Step 17: Semantic complexity - SemanticAnalyzer._calculate_semantic_complexity
    Step 18: Evidence confidence - IndustrialPolicyProcessor._compute_evidence_confidence
    Step 19: Entity serialization - PDETMunicipalPlanAnalyzer._entity_to_dict (loop)
    Step 20: Traceability record - AdaptivePriorCalculator.generate_traceability_record
    Step 21: PDQ report - PolicyAnalysisEmbedder.generate_pdq_report
    Step 22: Accountability matrix - ReportingEngine.generate_accountability_matrix
```

```
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Match programs to budget nodes
        program_matches = self._execute_method(
            "FinancialAuditor", "_match_program_to_node", context
        )
        goal_budget_matches = self._execute_method(
            "FinancialAuditor", "_match_goal_to_budget", context,
            programs=program_matches
        )

        # Step 2: Extract responsibility assignments
        responsibility_data = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables", context
        )
        consolidated_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
            entities=responsibility_data
        )
        responsible_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
        )
        responsibility_clarity = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity", context,
            entities=consolidated_entities
        )
        # Semantic traceability via embeddings
        semantic_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "process_document", context,
            document_text=document_text,
            document_metadata=document_metadata
        )
        pdq_query = self._execute_method(
            "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
            pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
        )
        semantic_hits = self._execute_method(
            "PolicyAnalysisEmbedder", "semantic_search", context,
            query=pdq_query,
            document_chunks=semantic_chunks or []
        )
        diversified_hits = self._execute_method(
            "PolicyAnalysisEmbedder", "_apply_mmr", context,
            ranked_results=semantic_hits or []
        )
        semantic_cube_stub = self._execute_method(
            "SemanticAnalyzer", "_empty_semantic_cube", context
        )
        domain_scores = self._execute_method(
            "SemanticAnalyzer", "_classify_policy_domain", context,
```

```
            segment=document_text
        )
        cross_cutting = self._execute_method(
            "SemanticAnalyzer", "_classify_cross_cutting_themes", context,
            segment=document_text
        )
        value_chain = self._execute_method(
            "SemanticAnalyzer", "_classify_value_chain_link", context,
            segment=document_text
        )
        semantic_vectors = self._execute_method(
            "SemanticAnalyzer", "_vectorize_segments", context,
            segments=[document_text]
        )
        processed_segment = self._execute_method(
            "SemanticAnalyzer", "_process_segment", context,
            segment=document_text,
            idx=0,
            vector=semantic_vectors[0] if semantic_vectors else None
        )
        semantic_complexity = self._execute_method(
            "SemanticAnalyzer", "_calculate_semantic_complexity", context,
            semantic_cube=semantic_cube_stub
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=[m.get("bpin","") for m in program_matches if isinstance(m, dict)],
            text_length=len(document_text),
            pattern_specificity=0.5
        )

        consolidated_entities = self._safe_process_list(
            consolidated_entities, label="consolidated_entities"
        )

        entity_dicts = []
        for e in consolidated_entities[:5]:
            if not (isinstance(e, dict) or hasattr(e, "__dict__")):
                continue

            try:
                entity_dict = self._execute_method(
                    "PDETMunicipalPlanAnalyzer", "_entity_to_dict", context, entity=e
                )
                entity_dict = self._safe_process_dict(entity_dict, label=f"entity_dict_{len(entity_dicts)}")
                entity_dicts.append(entity_dict)
            except MemoryError:
                logger.error("Memory exhausted during entity conversion, stopping")
                break
            except ExecutorFailure as e:
                logger.warning(f"Entity conversion failed: {e}")
                continue

        # Step 3: Generate traceability records
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
```

```
            matches=program_matches
        )

        # Step 4: Generate PDQ report
        pdq_report = self._execute_method(
            "PolicyAnalysisEmbedder", "generate_pdq_report", context,
            traceability=traceability_record
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
            entities=consolidated_entities
        )

        raw_evidence = {
            "budgetary_traceability": {
                "bpin_codes": [m.get("bpin") for m in (program_matches or []) if m.get("bpin")],
                "project_codes": [m.get("project_code") for m in (program_matches or []) if m.get("project_code")],
                "budget_matches": goal_budget_matches
            },
            "organizational_traceability": {
                "responsible_entities": consolidated_entities,
                "office_assignments": [e for e in (consolidated_entities or []) if e.get("office")],
                "secretariat_assignments": [e for e in (consolidated_entities or []) if e.get("secretariat")]
            },
            "traceability_record": traceability_record,
            "pdq_report": pdq_report,
            "accountability_matrix": accountability_matrix,
            "responsible_entities": responsible_entities,
            "responsibility_clarity_score": responsibility_clarity,
            "semantic_traceability": {
                "query": pdq_query,
                "semantic_hits": semantic_hits,
                "diversified_hits": diversified_hits
            },
            "semantic_cube_baseline": semantic_cube_stub,
            "policy_domain_scores": domain_scores,
            "responsibility_entities_dict": entity_dicts,
            "cross_cutting_themes": cross_cutting,
            "value_chain_links": value_chain,
            "semantic_vectors": semantic_vectors,
            "semantic_complexity": semantic_complexity,
            "evidence_confidence": evidence_confidence,
            "processed_segment": processed_segment
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "products_with_bpin": len([m for m in program_matches if isinstance(m, dict) and m.get("bpin")]),
                "products_with_responsible": len(consolidated_entities) if consolidated_entities else 0,
                "total_semantic_hits": len(semantic_hits) if semantic_hits else 0,
                "has_semantic_hits": bool(semantic_hits),
```

```
                "total_responsible_entities": len(responsible_entities) if responsible_entities else 0,
                "has_responsible_entities": bool(responsible_entities),
                "total_diversified_hits": len(diversified_hits) if diversified_hits else 0,
                "total_entity_dicts": len(entity_dicts) if entity_dicts else 0,
                "has_semantic_vectors": bool(semantic_vectors),
                "total_semantic_vectors": len(semantic_vectors) if semantic_vectors else 0,
                "canonical_question": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q4_TechnicalFeasibilityEvaluator(BaseExecutor):
    """
    DIM03_Q04_TECHNICAL_FEASIBILITY â\200\224 Evaluates activity-product feasibility vs resources/deadlines (canonical D3).
    Epistemic mix: structural DAG validity, causal necessity, performance/implementation readiness, and statistical robustness.

    Methods executed (in order):
    Step 1: Acyclicity p-value - AdvancedDAGValidator.calculate_acyclicity_pvalue
    Step 2: Acyclicity check - AdvancedDAGValidator._is_acyclic
    Step 3: Graph statistics - AdvancedDAGValidator.get_graph_stats
    Step 4: Node importance - AdvancedDAGValidator._calculate_node_importance
    Step 5: Subgraph generation - AdvancedDAGValidator._generate_subgraph
    Step 6: Node addition - AdvancedDAGValidator.add_node
    Step 7: Edge addition - AdvancedDAGValidator.add_edge
    Step 8: Node export - AdvancedDAGValidator.export_nodes
    Step 9: RNG initialization - AdvancedDAGValidator._initialize_rng
    Step 10: Statistical power - AdvancedDAGValidator._calculate_statistical_power
    Step 11: Node validator - AdvancedDAGValidator._get_node_validator
    Step 12: Empty result creation - AdvancedDAGValidator._create_empty_result
    Step 13: Necessity test - BayesianMechanismInference._test_necessity
    Step 14: Validation suite - IndustrialGradeValidator.execute_suite
    Step 15: Connection matrix - IndustrialGradeValidator.validate_connection_matrix
    Step 16: Performance benchmarks - IndustrialGradeValidator.run_performance_benchmarks
    Step 17: Benchmark operation - IndustrialGradeValidator._benchmark_operation
    Step 18: Metric logging - IndustrialGradeValidator._log_metric
    Step 19: Engine readiness - IndustrialGradeValidator.validate_engine_readiness
    Step 20: Performance analysis - PerformanceAnalyzer.analyze_performance
    Step 21: Loss functions - PerformanceAnalyzer._calculate_loss_functions
    Step 22: Resource likelihood - HierarchicalGenerativeModel._calculate_likelihood
    Step 23: ESS calculation - HierarchicalGenerativeModel._calculate_ess
    Step 24: R-hat calculation - HierarchicalGenerativeModel._calculate_r_hat
    Step 25: Causal categories validation - IndustrialGradeValidator.validate_causal_categories
    Step 26: Category extraction - TeoriaCambio._extraer_categorias
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        plan_name = context.get("metadata", {}).get("title", "plan_desarrollo")

```

```
        # Step 1: Validate DAG structure
        acyclicity_pvalue = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )
        is_acyclic = self._execute_method(
            "AdvancedDAGValidator", "_is_acyclic", context
        )
        graph_stats = self._execute_method(
            "AdvancedDAGValidator", "get_graph_stats", context
        )
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
        )
        subgraph = self._execute_method(
            "AdvancedDAGValidator", "_generate_subgraph", context
        )
        added_node = self._execute_method(
            "AdvancedDAGValidator", "add_node", context,
            node_name="temp_node"
        )
        added_edge = self._execute_method(
            "AdvancedDAGValidator", "add_edge", context,
            source="temp_node",
            target="temp_target",
            weight=1.0
        )
        node_export = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context
        )
        rng_seed = self._execute_method(
            "AdvancedDAGValidator", "_initialize_rng", context,
            plan_name=plan_name,
            salt=dim_info.code
        )
        stat_power = self._execute_method(
            "AdvancedDAGValidator", "_calculate_statistical_power", context,
            s=int(graph_stats.get("edges", 0)),
            n=max(1, int(graph_stats.get("nodes", 1)))
        )
        node_validator = self._execute_method(
            "AdvancedDAGValidator", "_get_node_validator", context,
            node_type="producto"
        )
        empty_result = self._execute_method(
            "AdvancedDAGValidator", "_create_empty_result", context,
            plan_name=plan_name,
            seed=rng_seed,
            timestamp=context.get("metadata", {}).get("timestamp", "")
        )

        # Step 2: Test necessity of activities for products
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Execute industrial-grade validation
```

```
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )
        connection_validation = self._execute_method(
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )
        performance_benchmarks = self._execute_method(
            "IndustrialGradeValidator", "run_performance_benchmarks", context
        )
        benchmark_ops = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )
        metric_log = self._execute_method(
            "IndustrialGradeValidator", "_log_metric", context,
            name="custom_latency",
            value=graph_stats.get("edges", 0),
            unit="edges",
            threshold=10.0
        )
        engine_readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 4: Analyze performance
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )
        loss_functions = self._execute_method(
            "PerformanceAnalyzer", "_calculate_loss_functions", context
        )
        # Likelihood estimation for resource adequacy
        resource_likelihood = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_likelihood", context,
            mechanism_type="tecnico",
            observations={"coherence": (performance_analysis or {}).get("resource_fit", {}).get("score", 0.0)}
        )

        # Step 5: Calculate effective sample size
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context,
            chains=[]
        )
        causal_categories_valid = self._execute_method(
            "IndustrialGradeValidator", "validate_causal_categories", context
        )
        extracted_categories = self._execute_method(
            "TeoriaCambio", "_extraer_categorias", context,
            text=context.get("document_text", "")
        )

        raw_evidence = {
            "activity_product_mapping": connection_validation,
            "resource_adequacy": (performance_analysis or {}).get("resource_fit", {}),
```

```
            "timeline_feasibility": (performance_analysis or {}).get("timeline_feasibility", {}),
            "technical_validation": {
                "dag_valid": is_acyclic,
                "acyclicity_p": acyclicity_pvalue,
                "necessity_score": necessity_test,
                "graph_stats": graph_stats,
                "node_importance": node_importance,
                "subgraph_sample": subgraph,
                "added_node": added_node,
                "added_edge": added_edge,
                "node_validator": node_validator,
                "empty_result": empty_result,
                "node_export": node_export,
                "rng_seed": rng_seed,
                "statistical_power": stat_power
            },
            "performance_metrics": {
                "benchmarks": performance_benchmarks,
                "loss_functions": loss_functions,
                "ess": ess,
                "r_hat": r_hat,
                "resource_likelihood": resource_likelihood
            },
            "engine_readiness": engine_readiness,
            "feasibility_score": validation_suite.get("overall_score", 0),
            "causal_categories_valid": causal_categories_valid,
            "extracted_categories": extracted_categories,
            "metric_log": metric_log
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "dag_is_valid": is_acyclic,
                "feasibility_score": (validation_suite or {}).get("overall_score", 0) if validation_suite else 0,
                "total_graph_nodes": (graph_stats or {}).get("nodes", 0) if graph_stats else 0,
                "total_graph_edges": (graph_stats or {}).get("edges", 0) if graph_stats else 0,
                "has_node_export": bool(node_export),
                "total_exported_nodes": len(node_export) if node_export else 0,
                "has_subgraph": bool(subgraph),
                "total_extracted_categories": len(extracted_categories) if extracted_categories else 0,
                "has_extracted_categories": bool(extracted_categories),
                "statistical_power": stat_power if isinstance(stat_power, (int, float)) else 0.0,
                "has_engine_readiness": bool(engine_readiness),
                "canonical_question": "DIM03_Q04_TECHNICAL_FEASIBILITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }

```

```

class D3_Q5_OutputOutcomeLinkageAnalyzer(BaseExecutor):
    """
    DIM03_Q05_OUTPUT_OUTCOME_LINKAGE â\200\224 Analyzes mechanisms linking outputs to outcomes with canonical D3 labeling.
    Epistemic mix: semantic hierarchy checks, causal order validation, DAG/effect estimation, and Bayesian mechanism inference.

Methods (from D3-Q5):
    - PDETMunicipalPlanAnalyzer._identify_confounders
    - PDETMunicipalPlanAnalyzer._effect_to_dict
    - PDETMunicipalPlanAnalyzer._scenario_to_dict
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - PDETMunicipalPlanAnalyzer._identify_causal_nodes
    - BayesianCounterfactualAuditor._evaluate_factual
    - BayesianCounterfactualAuditor._evaluate_counterfactual
    - CausalExtractor._assess_financial_consistency
    - BayesianMechanismInference._infer_activity_sequence
    - BayesianMechanismInference._generate_necessity_remediation
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - IndustrialPolicyProcessor._load_questionnaire
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - PDETMunicipalPlanAnalyzer.estimate_causal_effects
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - CausalExtractor._build_type_hierarchy
    - CausalExtractor._check_structural_violation
    - CausalExtractor._calculate_type_transition_prior
    - CausalExtractor._calculate_textual_proximity
    - TeoriaCambio._validar_orden_causal
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - PolicyAnalysisEmbedder.compare_policy_interventions
    - BayesianMechanismInference.infer_mechanisms
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Build causal backbone and effects
        financial_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        causal_dag = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
            financial_analysis=financial_analysis
        )
        causal_effects = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "estimate_causal_effects", context,
            dag=causal_dag,
            financial_analysis=financial_analysis
        )
        counterfactuals = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context,
            dag=causal_dag,
            causal_effects=causal_effects,
            financial_analysis=financial_analysis
```

```
        )
        simulated_intervention = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            intervention={},
            dag=causal_dag,
            causal_effects=causal_effects,
            label="baseline"
        )
        causal_nodes = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_nodes", context,
            text=context.get("document_text", ""),
            tables=context.get("tables", []),
            financial_analysis=financial_analysis
        )
        confounders = {}
        for effect in causal_effects:
            treatment = effect.treatment if hasattr(effect, "treatment") else None
            outcome = effect.outcome if hasattr(effect, "outcome") else None
            if treatment and outcome:
                confounders[(treatment, outcome)] = self._execute_method(
                    "PDETMunicipalPlanAnalyzer", "_identify_confounders", context,
                    treatment=treatment,
                    outcome=outcome,
                    dag=causal_dag
                )
        effect_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_effect_to_dict", context, effect=effect)
            for effect in causal_effects
        ]
        scenario_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_scenario_to_dict", context, scenario=scenario)
            for scenario in counterfactuals
        ]
        causal_recommendations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={"financial_analysis": financial_analysis, "quality_score": getattr(causal_dag, 'graph', {})}
        )
        factual_eval = None
        counterfactual_eval = None
        if causal_effects:
            first_effect = causal_effects[0]
            target = getattr(first_effect, "outcome", None) or ""
            evidence = {"p_effect": getattr(first_effect, "probability_significant", 0.0)}
            factual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_factual", context,
                target=target,
                evidence=evidence
            )
            counterfactual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_counterfactual", context,
                target=target,
                intervention={"shift": 0.1}
            )
        # Only catch specific expected exceptions, let system exceptions propagate
        matched_node = None
        try:
```

```
            matched_node = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_match_text_to_node", context,
                text=context.get("document_text", "")[:200],
                nodes=causal_nodes if isinstance(causal_nodes, dict) else {}
            )
        except (KeyError, ValueError, TypeError, AttributeError, ExecutorFailure) as e:
            logger.warning(f"Node matching failed: {type(e).__name__}: {e}")
            matched_node = None
        # Let critical system exceptions (KeyboardInterrupt, SystemExit, MemoryError) propagate

        # Step 1: Build type hierarchy
        type_hierarchy = self._execute_method(
            "CausalExtractor", "_build_type_hierarchy", context
        )

        # Step 2: Check structural violations
        structural_violations = self._execute_method(
            "CausalExtractor", "_check_structural_violation", context,
            hierarchy=type_hierarchy
        )

        # Step 3: Calculate transition priors and proximity
        transition_priors = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            hierarchy=type_hierarchy
        )
        textual_proximity = self._execute_method(
            "CausalExtractor", "_calculate_textual_proximity", context
        )

        # Step 4: Validate causal order
        causal_order_validation = self._execute_method(
            "TeoriaCambio", "_validar_orden_causal", context,
            hierarchy=type_hierarchy
        )

        # Step 5: Refine edge probabilities
        refined_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            priors=transition_priors
        )
        financial_consistency = None
        if refined_edges:
            first_edge = refined_edges[0] if isinstance(refined_edges, list) else {}
            source = first_edge.get("source") if isinstance(first_edge, dict) else ""
            target = first_edge.get("target") if isinstance(first_edge, dict) else ""
            financial_consistency = self._execute_method(
                "CausalExtractor", "_assess_financial_consistency", context,
                source=source or "",
                target=target or ""
            )

        # Step 6: Compare policy interventions
        intervention_comparison = self._execute_method(
            "PolicyAnalysisEmbedder", "compare_policy_interventions", context
        )
```

```

        # Step 7: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context,
            edges=refined_edges
        )
        mechanism_sample = next(iter(mechanisms.values()), {})
        activity_sequence = self._execute_method(
            "BayesianMechanismInference", "_infer_activity_sequence", context,
            observations=mechanism_sample.get("observations", {}),
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0})
        )
        quantified_uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0}),
            sequence_posterior=mechanism_sample.get("activity_sequence", {}),
            coherence_score=mechanism_sample.get("coherence_score", 0.0)
        )
        mechanism_observations = self._execute_method(
            "BayesianMechanismInference", "_extract_observations", context,
            node={"id": next(iter(mechanisms.keys()), "")},
            text=context.get("document_text", "")
        )
        necessity_remediation = self._execute_method(
            "BayesianMechanismInference", "_generate_necessity_remediation", context,
            node_id=next(iter(mechanisms.keys()), ""),
            missing_components=structural_violations
        )
        questionnaire_stub = self._execute_method(
            "IndustrialPolicyProcessor", "_load_questionnaire", context
        )
        # Only catch specific expected exceptions, let system exceptions propagate
        refutation_checks = None
        try:
            confounder_keys = list(confounders.keys())
            first_pair = confounder_keys[0] if confounder_keys else ("", "")
            refutation_checks = self._execute_method(
                "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
                dag=getattr(causal_dag, "graph", None),
                target=first_pair[1],
                treatment=first_pair[0],
                confounders=list(confounders.values())[0] if confounders else []
            )
        except (KeyError, ValueError, TypeError, AttributeError, IndexError, ExecutorFailure) as e:
            logger.warning(f"Refutation checks failed: {type(e).__name__}: {e}")
            refutation_checks = None
        # Let critical system exceptions (KeyboardInterrupt, SystemExit, MemoryError) propagate

        raw_evidence = {
            "output_outcome_links": refined_edges,
            "mechanism_explanation": mechanisms,
            "type_hierarchy": type_hierarchy,
            "causal_dag": causal_dag,
            "causal_effects": causal_effects,
            "counterfactuals": counterfactuals,
            "simulated_intervention": simulated_intervention,
```

```
            "causal_nodes": causal_nodes,
            "financial_analysis": financial_analysis,
            "causal_validity": {
                "structural_violations": structural_violations,
                "order_valid": causal_order_validation
            },
            "transition_probabilities": transition_priors,
            "textual_proximity": textual_proximity,
            "intervention_comparison": intervention_comparison,
            "confounders": confounders,
            "effect_dicts": effect_dicts,
            "scenario_dicts": scenario_dicts,
            "activity_sequence_sample": activity_sequence,
            "uncertainty_quantified": quantified_uncertainty,
            "mechanism_observations": mechanism_observations,
            "refutation_checks": refutation_checks,
            "necessity_remediation": necessity_remediation,
            "questionnaire_stub": questionnaire_stub,
            "causal_recommendations": causal_recommendations,
            "financial_consistency": financial_consistency,
            "factual_eval": factual_eval,
            "counterfactual_eval": counterfactual_eval,
            "matched_node": matched_node
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(mechanisms or {}),
                "violations_found": len(structural_violations or []),
                "canonical_question": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 4: RESULTS & OUTCOMES
# ============================================================================

class D4_Q1_OutcomeMetricsValidator(BaseExecutor):
    """
    DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS â\200\224 Validates outcome indicators (baseline, target, horizon) with canonical D4 notation.
    Epistemic mix: semantic goal extraction, temporal/consistency checks, statistical performance signals, and indicator quality scoring.

    Methods (from D4-Q1):
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - CausalExtractor._calculate_language_specificity
```

```
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._calculate_semantic_distance
    - TemporalLogicVerifier._classify_temporal_type
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._find_outcome_mentions
    - PDETMunicipalPlanAnalyzer._score_temporal_consistency
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
    - CausalExtractor._classify_goal_type
    - TemporalLogicVerifier._parse_temporal_marker
    - TemporalLogicVerifier._extract_resources
    - TemporalLogicVerifier._should_precede
    - PerformanceAnalyzer.analyze_performance
    - PerformanceAnalyzer._generate_recommendations
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D4.value)

        # Step 1: Find outcome mentions
        outcome_mentions = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_outcome_mentions", context
        )
        entities_syntax = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context,
            text=context.get("document_text", "")
        )
        entities_syntax = self._safe_process_list(entities_syntax, label="entities_syntax")

        entities_ner = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context,
            text=context.get("document_text", "")
        )
        entities_ner = self._safe_process_list(entities_ner, label="entities_ner")

        # Step 2: Score temporal consistency
        temporal_consistency = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_temporal_consistency", context,
            outcomes=outcome_mentions
        )

        # Step 3: Extract and classify goals
        goals = self._execute_method(
            "CausalExtractor", "_extract_goals", context
        )
        goal_contexts = self._execute_method(
            "CausalExtractor", "_parse_goal_context", context,
            goals=goals
        )
        goal_types = self._execute_method(
            "CausalExtractor", "_classify_goal_type", context,
            goals=goals
        )
        semantic_distance = 0.0
        if goal_types and outcome_mentions:
```

```
                semantic_distance = self._execute_method(
                    "CausalExtractor", "_calculate_semantic_distance", context,
                    source=str(goal_types[0]),
                    target=str(outcome_mentions[0])
                )

            # Step 4: Parse temporal markers
            temporal_markers = self._execute_method(
                "TemporalLogicVerifier", "_parse_temporal_marker", context,
                contexts=goal_contexts
            )
            temporal_type = self._execute_method(
                "TemporalLogicVerifier", "_classify_temporal_type", context,
                marker=temporal_markers[0] if temporal_markers else ""
            )
            resources_mentioned = self._execute_method(
                "TemporalLogicVerifier", "_extract_resources", context,
                text=context.get("document_text", "")
            )
            precedence_check = self._execute_method(
                "TemporalLogicVerifier", "_should_precede", context,
                marker_a=temporal_markers[0] if temporal_markers else "",
                marker_b=temporal_markers[1] if len(temporal_markers) > 1 else ""
            )

            # Step 5: Analyze performance
            performance_analysis = self._execute_method(
                "PerformanceAnalyzer", "analyze_performance", context,
                outcomes=outcome_mentions
            )
            indicator_quality = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_score_indicators", context
            )
            performance_recommendations = self._execute_method(
                "PerformanceAnalyzer", "_generate_recommendations", context,
                performance_analysis=performance_analysis
            )
            # Semantic certainty for goals
            language_specificity = self._execute_method(
                "CausalExtractor", "_calculate_language_specificity", context,
                keyword=goal_contexts[0] if goal_contexts else "",
                policy_area=context.get("policy_area")
            )
            composite_likelihood = self._execute_method(
                "CausalExtractor", "_calculate_composite_likelihood", context,
                evidence={
                    "semantic_distance": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0,
                    "textual_proximity": performance_analysis.get("coherence_score", 0.0) if isinstance(performance_analysis, dict) else 0.0,
                    "language_specificity": language_specificity,
                    "temporal_coherence": temporal_consistency if isinstance(temporal_consistency, (int, float)) else 0.0
                }
            )

            raw_evidence = {
                "outcome_indicators": outcome_mentions,
                "indicators_with_baseline": [o for o in outcome_mentions if o.get("has_baseline")],
```

```
            "indicators_with_target": [o for o in outcome_mentions if o.get("has_target")],
            "indicators_with_horizon": [o for o in outcome_mentions if o.get("time_horizon")],
            "temporal_consistency_score": temporal_consistency,
            "goal_classifications": goal_types,
            "temporal_markers": temporal_markers,
            "performance_metrics": performance_analysis,
            "indicator_quality": indicator_quality,
            "performance_recommendations": performance_recommendations,
            "entities_syntax": entities_syntax,
            "entities_ner": entities_ner,
            "temporal_type": temporal_type,
            "language_specificity": language_specificity,
            "composite_likelihood": composite_likelihood,
            "goal_outcome_semantic_distance": semantic_distance,
            "resources_mentioned": resources_mentioned,
            "precedence_check": precedence_check
        }

        memory_metrics = self._get_memory_metrics_summary()

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_outcomes": len(outcome_mentions or []),
                "complete_indicators": len([o for o in outcome_mentions or []
                    if o.get("has_baseline") and o.get("has_target") and o.get("time_horizon")]),
                "canonical_question": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log),
                "memory_safety": memory_metrics
            }
        }


class D4_Q2_CausalChainValidator(BaseExecutor):
    """
    Validates explicit causal chain with assumptions and enabling conditions.

    Methods (from D4-Q2):
    - TeoriaCambio._encontrar_caminos_completos
    - TeoriaCambio.validacion_completa
    - CausalExtractor.extract_causal_hierarchy
    - HierarchicalGenerativeModel.verify_conditional_independence
    - HierarchicalGenerativeModel._generate_independence_tests
    - BayesianCounterfactualAuditor.construct_scm
    - AdvancedDAGValidator._perform_sensitivity_analysis_internal
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
```

```
        raw_evidence = {}

        # Step 1: Find complete causal paths
        complete_paths = self._execute_method(
            "TeoriaCambio", "_encontrar_caminos_completos", context
        )

        # Step 2: Complete validation
        validation_results = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            paths=complete_paths
        )

        # Step 3: Extract causal hierarchy
        causal_hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        # Step 4: Verify conditional independence
        independence_verification = self._execute_method(
            "HierarchicalGenerativeModel", "verify_conditional_independence", context,
            hierarchy=causal_hierarchy
        )
        independence_tests = self._execute_method(
            "HierarchicalGenerativeModel", "_generate_independence_tests", context,
            verification=independence_verification
        )

        # Step 5: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            hierarchy=causal_hierarchy
        )

        # Step 6: Perform sensitivity analysis
        sensitivity_analysis = self._execute_method(
            "AdvancedDAGValidator", "_perform_sensitivity_analysis_internal", context,
            scm=scm
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            analysis=sensitivity_analysis
        )

        raw_evidence = {
            "causal_chain": complete_paths,
            "key_assumptions": validation_results.get("assumptions", []),
            "enabling_conditions": validation_results.get("conditions", []),
            "external_factors": validation_results.get("external_factors", []),
            "causal_hierarchy": causal_hierarchy,
            "independence_tests": independence_tests,
            "structural_model": scm,
            "sensitivity": sensitivity_analysis,
            "evidential_strength": bayes_factor
```

```
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "complete_paths_found": len(complete_paths),
                "assumptions_identified": len(validation_results.get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q3_AmbitionJustificationAnalyzer(BaseExecutor):
    """
    Analyzes justification of result ambition based on investment/capacity/benchmarks.

    Methods (from D4-Q3):
    - PDETMunicipalPlanAnalyzer._get_prior_effect
    - PDETMunicipalPlanAnalyzer._estimate_effect_bayesian
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - AdaptivePriorCalculator.sensitivity_analysis
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator._calculate_statistical_power
    - BayesianMechanismInference._aggregate_bayesian_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Get prior effect estimates
        prior_effects = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_prior_effect", context
        )

        # Step 2: Estimate effect using Bayesian methods
        effect_estimate = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_estimate_effect_bayesian", context,
            priors=prior_effects
        )

        # Step 3: Compute robustness
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            estimate=effect_estimate
        )

        # Step 4: Sensitivity analysis
        sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            estimate=effect_estimate
```

```
        )

        # Step 5: Calculate convergence diagnostics
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context
        )
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )

        # Step 6: Calculate statistical power
        statistical_power = self._execute_method(
            "AdvancedDAGValidator", "_calculate_statistical_power", context,
            effect=effect_estimate
        )

        # Step 7: Aggregate confidence
        confidence_aggregate = self._execute_method(
            "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
            estimates=[effect_estimate, robustness, statistical_power]
        )

        raw_evidence = {
            "ambition_level": context.get("target_ambition", {}),
            "financial_investment": context.get("total_investment", 0),
            "institutional_capacity": context.get("capacity_score", 0),
            "comparative_benchmarks": prior_effects,
            "justification_analysis": {
                "effect_estimate": effect_estimate,
                "robustness": robustness,
                "sensitivity": sensitivity,
                "statistical_power": statistical_power
            },
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess
            },
            "overall_confidence": confidence_aggregate
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "ambition_justified": confidence_aggregate > 0.7,
                "statistical_power": statistical_power
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q4_ProblemSolvencyEvaluator(BaseExecutor):
```

```
    """
    Evaluates whether results address/resolve prioritized problems from diagnosis.

    Methods (from D4-Q4):
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._identify_affected_sections
    - PolicyContradictionDetector._generate_resolution_recommendations
    - OperationalizationAuditor._generate_optimal_remediations
    - OperationalizationAuditor._get_remediation_text
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - FinancialAuditor._detect_allocation_gaps
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate objective alignment
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )

        # Step 2: Identify affected sections
        affected_sections = self._execute_method(
            "PolicyContradictionDetector", "_identify_affected_sections", context,
            alignment=objective_alignment
        )

        # Step 3: Generate resolution recommendations
        resolutions = self._execute_method(
            "PolicyContradictionDetector", "_generate_resolution_recommendations", context,
            sections=affected_sections
        )

        # Step 4: Generate optimal remediations
        remediations = self._execute_method(
            "OperationalizationAuditor", "_generate_optimal_remediations", context
        )
        remediation_text = self._execute_method(
            "OperationalizationAuditor", "_get_remediation_text", context,
            remediations=remediations
        )

        # Step 5: Aggregate risk and prioritize
        risk_priorities = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context
        )

        # Step 6: Detect allocation gaps
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )

        raw_evidence = {
            "prioritized_problems": context.get("diagnosis_problems", []),
            "proposed_results": context.get("outcome_indicators", []),
            "problem_result_mapping": objective_alignment,
```

```
            "unaddressed_problems": [p for p in affected_sections if not p.get("addressed")],
            "solvency_score": objective_alignment.get("score", 0),
            "resolution_recommendations": resolutions,
            "remediations": remediation_text,
            "risk_priorities": risk_priorities,
            "allocation_gaps": allocation_gaps
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "problems_addressed": len([p for p in affected_sections if p.get("addressed")]),
                "problems_unaddressed": len([p for p in affected_sections if not p.get("addressed")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q5_VerticalAlignmentValidator(BaseExecutor):
    """
    Validates alignment with superior frameworks (PND, SDGs).

    Methods (from D4-Q5):
    - PDETMunicipalPlanAnalyzer._score_pdet_alignment
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - CDAFFramework._validate_dnp_compliance
    - CDAFFramework._generate_dnp_report
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - AdaptivePriorCalculator.validate_quality_criteria
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score PDET alignment
        pdet_alignment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_pdet_alignment", context
        )

        # Step 2: Score causal coherence
        causal_coherence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 3: Validate DNP compliance
        dnp_compliance = self._execute_method(
            "CDAFFramework", "_validate_dnp_compliance", context
        )
        dnp_report = self._execute_method(
            "CDAFFramework", "_generate_dnp_report", context,
            compliance=dnp_compliance
```

```
        )

        # Step 4: Analyze causal dimensions
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )

        # Step 5: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context,
            alignment=pdet_alignment
        )

        raw_evidence = {
            "pnd_alignment": dnp_compliance,
            "sdg_alignment": context.get("sdg_mappings", []),
            "pdet_alignment": pdet_alignment,
            "alignment_declarations": (dnp_report or {}).get("declarations", []),
            "causal_coherence": causal_coherence,
            "causal_dimensions": causal_dimensions,
            "quality_validation": quality_validation,
            "alignment_score": (pdet_alignment.get("score", 0) +
                                (dnp_compliance or {}).get("score", 0)) / 2
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "pnd_aligned": (dnp_compliance or {}).get("is_compliant", False),
                "sdgs_referenced": len(context.get("sdg_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 5: IMPACTS
# ============================================================================

class D5_Q1_LongTermVisionAnalyzer(BaseExecutor):
    """
    Analyzes long-term impacts, transmission routes, and time lags.

    Methods (from D5-Q1):
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_scenario_narrative
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - TeoriaCambio._validar_orden_causal
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._generate_interventions
```

```
    - BayesianCounterfactualAuditor.construct_scm
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Generate counterfactuals
        counterfactuals = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context
        )

        # Step 2: Simulate interventions
        simulation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            counterfactuals=counterfactuals
        )

        # Step 3: Generate scenario narratives
        scenarios = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_scenario_narrative", context,
            simulation=simulation
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Validate causal order
        causal_order = self._execute_method(
            "TeoriaCambio", "_validar_orden_causal", context,
            mediators=mediators
        )

        # Step 6: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 7: Generate interventions
        interventions = self._execute_method(
            "TextMiningEngine", "_generate_interventions", context
        )

        # Step 8: Construct SCM
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            order=causal_order
        )

        raw_evidence = {
            "long_term_impacts": context.get("impact_indicators", []),
            "structural_transformations": scenarios,
            "transmission_routes": mediators,
            "expected_time_lags": temporal_coherence.get("time_lags", []),
            "counterfactual_analysis": counterfactuals,
```

```
            "simulation_results": simulation,
            "causal_order_validation": causal_order,
            "causal_pathways": scm,
            "intervention_scenarios": interventions
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "impacts_defined": len(context.get("impact_indicators", [])),
                "mediators_identified": len(mediators or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q2_CompositeMeasurementValidator(BaseExecutor):
    """
    DIM05_Q02_COMPOSITE_PROXY_VALIDITY â\200\224 Validates composite indices/proxies for complex impacts (canonical D5).
    Epistemic mix: statistical robustness (E-value), Bayesian confidence, normative reporting quality, and semantic consistency.

    Methods executed (in order):
    Step 1: Quality score calculation - PDETMunicipalPlanAnalyzer.calculate_quality_score
    Step 2: Score confidence estimation - PDETMunicipalPlanAnalyzer._estimate_score_confidence
    Step 3: E-value computation - PDETMunicipalPlanAnalyzer._compute_e_value
    Step 4: Robustness computation - PDETMunicipalPlanAnalyzer._compute_robustness_value
    Step 5: Sensitivity interpretation - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    Step 6: Reporting quality - ReportingEngine._calculate_quality_score
    Step 7: Bayesian confidence aggregation - BayesianMechanismInference._aggregate_bayesian_confidence
    Step 8: Numerical consistency evaluation - PolicyAnalysisEmbedder.evaluate_policy_numerical_consistency
    Step 9: Embedder diagnostics - PolicyAnalysisEmbedder.get_diagnostics
    Step 10: Document processing - PolicyAnalysisEmbedder.process_document
    Step 11: PDQ query generation - PolicyAnalysisEmbedder._generate_query_from_pdq
    Step 12: PDQ filtering - PolicyAnalysisEmbedder._filter_by_pdq
    Step 13: Numerical value extraction - PolicyAnalysisEmbedder._extract_numerical_values
    Step 14: Text embedding - PolicyAnalysisEmbedder._embed_texts
    Step 15: Overall confidence computation - PolicyAnalysisEmbedder._compute_overall_confidence
    Step 16: Sufficiency calculation - FinancialAuditor._calculate_sufficiency
    Step 17: Overall quality interpretation - PDETMunicipalPlanAnalyzer._interpret_overall_quality
    Step 18: Risk prioritization - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    Step 19: Unicode normalization - PolicyTextProcessor.normalize_unicode
    Step 20: Sentence segmentation - PolicyTextProcessor.segment_into_sentences
    Step 21: Evidence confidence - IndustrialPolicyProcessor._compute_evidence_confidence
    Step 22: Average confidence - IndustrialPolicyProcessor._compute_avg_confidence
    Step 23: Quality serialization - PDETMunicipalPlanAnalyzer._quality_to_dict
    Step 24: Evidence bundle - IndustrialPolicyProcessor._construct_evidence_bundle
    Step 25: Pattern compilation - PolicyTextProcessor.compile_pattern
    Step 26: Contextual window extraction - PolicyTextProcessor.extract_contextual_window
    Step 27: Executive report generation - PDETMunicipalPlanAnalyzer.generate_executive_report
    Step 28: Results export - IndustrialPolicyProcessor.export_results
    """
```

```

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D5.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Calculate quality scores
        quality_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "calculate_quality_score", context
        )
        score_confidence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_estimate_score_confidence", context,
            score=quality_score
        )

        # Step 2: Compute robustness metrics
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            score=quality_score
        )
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            score=quality_score
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            e_value=e_value,
            robustness=robustness
        )

        # Step 3: Calculate reporting quality score
        reporting_quality = self._execute_method(
            "ReportingEngine", "_calculate_quality_score", context
        )

        # Step 4: Aggregate Bayesian confidence
        bayesian_confidence = self._execute_method(
            "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
            scores=[quality_score, reporting_quality]
        )

        # Step 5: Evaluate numerical consistency
        numerical_consistency = self._execute_method(
            "PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency", context
        )
        embedder_diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context
        )
        processed_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "process_document", context,
            document_text=document_text,
            document_metadata=document_metadata
        )
        processed_chunks = self._safe_process_list(processed_chunks, label="processed_chunks")

```

```
        pdq_filter = self._execute_method(
            "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
            pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
        )
        filtered_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "_filter_by_pdq", context,
            chunks=processed_chunks,
            pdq_filter=pdq_filter
        )
        filtered_chunks = self._safe_process_list(filtered_chunks, label="filtered_chunks")
        numerical_values = self._execute_method(
            "PolicyAnalysisEmbedder", "_extract_numerical_values", context,
            chunks=processed_chunks
        )
        embedded_texts = self._execute_method(
            "PolicyAnalysisEmbedder", "_embed_texts", context,
            texts=[c.get("content", "") for c in processed_chunks] if isinstance(processed_chunks, list) else []
        )
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            relevant_chunks=filtered_chunks[:5] if isinstance(filtered_chunks, list) else [],
            numerical_eval=bayesian_confidence if isinstance(bayesian_confidence, dict) else {"evidence_strength": "weak", "numerical_coherence": 0.0}
        )

        # Step 6: Calculate sufficiency
        sufficiency = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )
        overall_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_overall_quality", context,
            score=getattr(quality_score, "overall_score", quality_score)
        )
        risk_prioritization = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            omission_score=1 - quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.2,
            insufficiency_score=1 - (sufficiency or {}).get("coverage_ratio", 0.0),
            unnecessity_score=1 - (robustness if isinstance(robustness, (int, float)) else 0.0),
            causal_effect=e_value,
            feasibility=quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.8,
            cost=1.0
        )
        normalized_text = self._execute_method(
            "PolicyTextProcessor", "normalize_unicode", context,
            text=document_text
        )
        segmented_sentences = self._execute_method(
            "PolicyTextProcessor", "segment_into_sentences", context,
            text=document_text
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=context.get("proxy_indicators", []),
            text_length=len(document_text),
            pattern_specificity=0.5
        )
        avg_confidence = self._execute_method(
```

```
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D5": {"dimension_confidence": (bayesian_confidence or {}).get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0}}
        )
        quality_dict = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_quality_to_dict", context,
            quality=quality_score
        )
        evidence_bundle = self._execute_method(
            "IndustrialPolicyProcessor", "_construct_evidence_bundle", context,
            dimension=None,
            category="composite",
            matches=context.get("proxy_indicators", []),
            positions=[],
            confidence=bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0
        )
        compiled_pattern = self._execute_method(
            "PolicyTextProcessor", "compile_pattern", context,
            pattern_str=r"[A-Z]{2,}\\s+\\d+"
        )
        contextual_window = self._execute_method(
            "PolicyTextProcessor", "extract_contextual_window", context,
            text=document_text,
            match_position=0,
            window_size=200
        )
        exec_report = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_executive_report", context,
            analysis_results={"quality_score": quality_dict, "financial_analysis": context.get("financial_analysis", {}) or {"total_budget": 0, "funding_sources": {}, "confidence": (0, 0)}}
        )
        export_result = self._execute_method(
            "IndustrialPolicyProcessor", "export_results", context,
            results={"quality": quality_dict, "robustness": robustness},
            output_path="output/composite_results.json"
        )

        raw_evidence = {
            "composite_indices": context.get("composite_indicators", []),
            "proxy_indicators": context.get("proxy_indicators", []),
            "validity_justification": score_confidence,
            "robustness_metrics": {
                "e_value": e_value,
                "robustness": robustness,
                "interpretation": sensitivity_interpretation
            },
            "quality_scores": {
                "overall": quality_score,
                "reporting": reporting_quality
            },
            "bayesian_confidence": bayesian_confidence,
            "numerical_consistency": numerical_consistency,
            "measurement_sufficiency": sufficiency,
            "embedder_diagnostics": embedder_diagnostics,
            "quality_interpretation": overall_interpretation,
            "pdq_filter": pdq_filter,
```

```
            "filtered_chunks": filtered_chunks,
            "numerical_values": numerical_values,
            "embedded_texts": embedded_texts,
            "overall_confidence": overall_confidence,
            "risk_prioritization": risk_prioritization,
            "normalized_text": normalized_text,
            "segmented_sentences": segmented_sentences,
            "evidence_confidence": evidence_confidence,
            "avg_confidence": avg_confidence,
            "quality_dict": quality_dict,
            "compiled_pattern": compiled_pattern,
            "contextual_window": contextual_window,
            "evidence_bundle": evidence_bundle,
            "executive_report": exec_report,
            "export_result": export_result
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "composite_indices_count": len(context.get("composite_indicators", [])),
                "total_proxy_indicators": len(context.get("proxy_indicators", [])),
                "has_proxy_indicators": bool(context.get("proxy_indicators")),
                "total_numerical_values": len(numerical_values) if numerical_values else 0,
                "has_numerical_values": bool(numerical_values),
                "total_filtered_chunks": len(filtered_chunks) if filtered_chunks else 0,
                "has_filtered_chunks": bool(filtered_chunks),
                "total_segmented_sentences": len(segmented_sentences) if segmented_sentences else 0,
                "has_segmented_sentences": bool(segmented_sentences),
                "total_embedded_texts": len(embedded_texts) if embedded_texts else 0,
                "has_embedded_texts": bool(embedded_texts),
                "validity_score": score_confidence,
                "canonical_question": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log),
                "memory_safety": self._get_memory_metrics_summary()
            }
        }


class D5_Q3_IntangibleMeasurementAnalyzer(BaseExecutor):
    """
    Analyzes proxy indicators for intangible impacts with validity documentation.

    Methods (from D5-Q3):
    - CausalExtractor._calculate_semantic_distance
    - SemanticAnalyzer.extract_semantic_cube
    - BayesianMechanismInference._quantify_uncertainty
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - PolicyAnalysisEmbedder.get_diagnostics
```

```
    - AdaptivePriorCalculator._perturb_evidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate semantic distance
        semantic_distance = self._execute_method(
            "CausalExtractor", "_calculate_semantic_distance", context
        )

        # Step 2: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        # Step 3: Quantify uncertainty
        uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            semantic_data=semantic_cube
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Get diagnostics
        diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context,
            mediators=mediators
        )

        # Step 6: Perturb evidence for sensitivity
        perturbed_evidence = self._execute_method(
            "AdaptivePriorCalculator", "_perturb_evidence", context,
            diagnostics=diagnostics
        )

        raw_evidence = {
            "intangible_impacts": context.get("intangible_indicators", []),
            "proxy_indicators": context.get("proxy_mappings", []),
            "validity_documentation": diagnostics,
            "limitations_acknowledged": (diagnostics or {}).get("limitations", []),
            "semantic_relationships": semantic_cube,
            "semantic_distance": semantic_distance,
            "uncertainty_quantification": uncertainty,
            "sensitivity_to_proxies": perturbed_evidence
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "intangibles_count": len(context.get("intangible_indicators", [])),
```

```
                "proxies_defined": len(context.get("proxy_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q4_SystemicRiskEvaluator(BaseExecutor):
    """
    Evaluates systemic risks that could rupture causal mechanisms.

    Methods (from D5-Q4):
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - BayesianCounterfactualAuditor._test_effect_stability
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - PDETMunicipalPlanAnalyzer._break_cycles
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Audit systemic risks
        systemic_risks = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 2: Refutation and sanity checks
        refutation = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            risks=systemic_risks
        )

        # Step 3: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            refutation=refutation
        )

        # Step 4: Interpret risks
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risks=systemic_risks
        )

        # Step 5: Interpret sensitivity
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            stability=effect_stability
        )

        # Step 6: Break cycles if present
```

```
        cycle_breaks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_break_cycles", context
        )

        # Step 7: Sensitivity analysis
        sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=systemic_risks
        )

        raw_evidence = {
            "macroeconomic_risks": [r for r in systemic_risks or [] if r.get("type") == "macroeconomic"],
            "environmental_risks": [r for r in systemic_risks or [] if r.get("type") == "environmental"],
            "political_risks": [r for r in systemic_risks or [] if r.get("type") == "political"],
            "mechanism_rupture_potential": (risk_interpretation or {}).get("rupture_probability", 0),
            "effect_stability": effect_stability,
            "refutation_results": refutation,
            "sensitivity_analysis": sensitivity,
            "sensitivity_interpretation": sensitivity_interpretation,
            "cycle_vulnerabilities": cycle_breaks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "systemic_risks_identified": len(systemic_risks or []),
                "high_risk_count": len([r for r in systemic_risks or [] if r.get("severity") == "high"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q5_RealismAndSideEffectsAnalyzer(BaseExecutor):
    """
    Analyzes realism of impact ambition and potential unintended effects.

    Methods (from D5-Q5):
    - HierarchicalGenerativeModel.posterior_predictive_check
    - HierarchicalGenerativeModel._ablation_analysis
    - HierarchicalGenerativeModel._calculate_waic_difference
    - AdaptivePriorCalculator._add_ood_noise
    - AdaptivePriorCalculator.validate_quality_criteria
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - BayesianMechanismInference._calculate_coherence_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Posterior predictive check
```

```
        predictive_check = self._execute_method(
            "HierarchicalGenerativeModel", "posterior_predictive_check", context
        )

        # Step 2: Ablation analysis
        ablation = self._execute_method(
            "HierarchicalGenerativeModel", "_ablation_analysis", context,
            check=predictive_check
        )

        # Step 3: Calculate WAIC difference
        waic_diff = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_waic_difference", context,
            ablation=ablation
        )

        # Step 4: Add out-of-distribution noise
        ood_analysis = self._execute_method(
            "AdaptivePriorCalculator", "_add_ood_noise", context
        )

        # Step 5: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context,
            ood=ood_analysis
        )

        # Step 6: Compute robustness metrics
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context
        )
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context
        )

        # Step 7: Calculate coherence factor
        coherence = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context
        )

        raw_evidence = {
            "impact_ambition_level": context.get("declared_ambition", 0),
            "realism_assessment": (predictive_check or {}).get("realism_score", 0),
            "negative_side_effects": (ablation or {}).get("negative_effects", []),
            "limit_hypotheses": (quality_validation or {}).get("limits", []),
            "robustness_metrics": {
                "e_value": e_value,
                "robustness": robustness,
                "coherence": coherence
            },
            "predictive_validity": predictive_check,
            "ablation_results": ablation,
            "model_comparison": waic_diff,
            "ood_sensitivity": ood_analysis
        }

```

```
        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "realism_score": (predictive_check or {}).get("realism_score", 0),
                "side_effects_identified": len((ablation or {}).get("negative_effects", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 6: CAUSALITY & THEORY OF CHANGE
# ============================================================================

class D6_Q1_ExplicitTheoryBuilder(BaseExecutor):
    """
    Builds/validates explicit Theory of Change with diagram and assumptions.

    Methods (from D6-Q1):
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio.validacion_completa
    - TeoriaCambio.export_nodes
    - ReportingEngine.generate_causal_diagram
    - ReportingEngine.generate_causal_model_json
    - AdvancedDAGValidator.export_nodes
    - PDETMunicipalPlanAnalyzer.export_causal_network
    - CausalExtractor.extract_causal_hierarchy
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Build causal graph
        causal_graph = self._execute_method(
            "TeoriaCambio", "construir_grafo_causal", context
        )

        # Step 2: Complete validation
        validation = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            graph=causal_graph
        )

        causal_graph = self._safe_process_dict(causal_graph, label="causal_graph")

        # Step 3: Export nodes from Theory of Change
        toc_nodes = self._execute_method(
            "TeoriaCambio", "export_nodes", context,
            graph=causal_graph
        )
        toc_nodes = self._safe_process_list(toc_nodes, label="toc_nodes")
```

```
        # Step 4: Generate causal diagram
        diagram = self._execute_method(
            "ReportingEngine", "generate_causal_diagram", context,
            graph=causal_graph
        )

        # Step 5: Generate causal model JSON
        model_json = self._execute_method(
            "ReportingEngine", "generate_causal_model_json", context,
            graph=causal_graph
        )
        model_json = self._safe_process_dict(model_json, label="causal_model_json")

        # Step 6: Export nodes from DAG validator
        dag_nodes = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context,
            graph=causal_graph
        )
        dag_nodes = self._safe_process_list(dag_nodes, label="dag_nodes")

        # Step 7: Export causal network
        network_export = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "export_causal_network", context,
            graph=causal_graph
        )
        network_export = self._safe_process_dict(network_export, label="network_export")

        # Step 8: Extract causal hierarchy
        hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )
        hierarchy = self._safe_process_dict(hierarchy, label="causal_hierarchy")

        raw_evidence = {
            "toc_exists": bool(causal_graph),
            "toc_diagram": diagram,
            "toc_json": model_json,
            "causal_graph": causal_graph,
            "nodes": toc_nodes,
            "dag_nodes": dag_nodes,
            "causes_identified": (hierarchy or {}).get("causes", []),
            "mediators_identified": (hierarchy or {}).get("mediators", []),
            "assumptions": (validation or {}).get("assumptions", []),
            "network_structure": network_export,
            "validation_results": validation
        }

        memory_metrics = self._get_memory_metrics_summary()

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "nodes_count": len(toc_nodes or []),
```

```
                "assumptions_count": len((validation or {}).get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log),
                "memory_safety": memory_metrics
            }
        }


class D6_Q2_LogicalProportionalityValidator(BaseExecutor):
    """
    Validates logical proportionality: no leaps, intervention matches result scale.

    Methods (from D6-Q2):
    - BeachEvidentialTest.apply_test_logic
    - BayesianMechanismInference._test_necessity
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._calculate_coherence_factor
    - BayesianCounterfactualAuditor._test_effect_stability
    - IndustrialGradeValidator.validate_connection_matrix
    - PolicyAnalysisEmbedder._compute_overall_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context
        )

        # Step 2: Test necessity
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Test sufficiency
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )

        # Step 4: Calculate coherence factor
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            necessity=necessity_test,
            sufficiency=sufficiency_test
        )

        # Step 5: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context
        )

        # Step 6: Validate connection matrix
        connection_validation = self._execute_method(
```

```
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )

        # Step 7: Compute overall confidence
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            tests=[necessity_test, sufficiency_test, effect_stability]
        )

        raw_evidence = {
            "logical_leaps_detected": (evidential_tests or {}).get("leaps", []),
            "intervention_scale": context.get("intervention_magnitude", 0),
            "result_scale": context.get("result_magnitude", 0),
            "proportionality_ratio": context.get("intervention_magnitude", 0) / max(context.get("result_magnitude", 1), 1),
            "necessity_score": necessity_test,
            "sufficiency_score": sufficiency_test,
            "coherence_factor": coherence_factor,
            "effect_stability": effect_stability,
            "connection_validation": connection_validation,
            "overall_confidence": overall_confidence,
            "implementation_miracles": [leap for leap in (evidential_tests or {}).get("leaps", [])
                                       if isinstance(leap, dict) and leap.get("type") == "miracle"]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "leaps_detected": len((evidential_tests or {}).get("leaps", [])),
                "proportionality_adequate": abs(raw_evidence["proportionality_ratio"] - 1.0) < 0.5
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q3_ValidationTestingAnalyzer(BaseExecutor):
    """
    Analyzes validation/testing proposals for weak assumptions before scaling.

    Methods (from D6-Q3):
    - IndustrialGradeValidator.execute_suite
    - IndustrialGradeValidator.validate_engine_readiness
    - IndustrialGradeValidator._benchmark_operation
    - AdaptivePriorCalculator.validate_quality_criteria
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator.calculate_acyclicity_pvalue
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
```

```
        # Step 1: Execute validation suite
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )

        # Step 2: Validate engine readiness
        readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 3: Benchmark operations
        benchmarks = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )

        # Step 4: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context
        )

        # Step 5: Calculate convergence diagnostics
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context
        )
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )

        # Step 6: Calculate acyclicity p-value
        acyclicity_p = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )

        # Step 7: Analyze performance
        performance = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "inconsistencies_recognized": (validation_suite or {}).get("inconsistencies", []),
            "weak_assumptions": (quality_validation or {}).get("weak_assumptions", []),
            "pilot_proposals": context.get("pilot_programs", []),
            "testing_proposals": context.get("testing_plans", []),
            "validation_before_scaling": (readiness or {}).get("ready_to_scale", False),
            "validation_results": validation_suite,
            "quality_criteria": quality_validation,
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess,
                "acyclicity_p": acyclicity_p
            },
            "performance_analysis": performance,
            "benchmarks": benchmarks
        }

```

```
        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "inconsistencies_count": len((validation_suite or {}).get("inconsistencies", [])),
                "pilots_proposed": len(context.get("pilot_programs", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q4_FeedbackLoopAnalyzer(BaseExecutor):
    """
    Analyzes monitoring system with correction mechanisms and learning processes.

    Methods (from D6-Q4):
    - ConfigLoader.update_priors_from_feedback
    - ConfigLoader.check_uncertainty_reduction_criterion
    - ConfigLoader._save_prior_history
    - ConfigLoader._load_uncertainty_history
    - CDAFFramework._extract_feedback_from_audit
    - AdvancedDAGValidator._calculate_node_importance
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Update priors from feedback
        prior_updates = self._execute_method(
            "ConfigLoader", "update_priors_from_feedback", context
        )

        # Step 2: Check uncertainty reduction
        uncertainty_reduction = self._execute_method(
            "ConfigLoader", "check_uncertainty_reduction_criterion", context,
            updates=prior_updates
        )

        # Step 3: Save prior history
        history_saved = self._execute_method(
            "ConfigLoader", "_save_prior_history", context,
            updates=prior_updates
        )

        # Step 4: Load uncertainty history
        uncertainty_history = self._execute_method(
            "ConfigLoader", "_load_uncertainty_history", context
        )

        # Step 5: Extract feedback from audit
        feedback_extracted = self._execute_method(
```

```
            "CDAFFramework", "_extract_feedback_from_audit", context
        )

        # Step 6: Calculate node importance
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            updates=prior_updates
        )

        raw_evidence = {
            "monitoring_system_described": len(context.get("monitoring_indicators", [])) > 0,
            "correction_mechanisms": (feedback_extracted or {}).get("mechanisms", []),
            "feedback_loops": (feedback_extracted or {}).get("loops", []),
            "learning_processes": (feedback_extracted or {}).get("learning", []),
            "prior_updates": prior_updates,
            "uncertainty_reduction": uncertainty_reduction,
            "history_saved": history_saved,
            "uncertainty_history": uncertainty_history,
            "node_importance": node_importance,
            "learning_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "feedback_mechanisms": len((feedback_extracted or {}).get("mechanisms", [])),
                "learning_processes": len((feedback_extracted or {}).get("learning", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q5_ContextualAdaptabilityEvaluator(BaseExecutor):
    """
    Evaluates contextual adaptation: differential impacts and territorial constraints.

    Methods executed (in order):
    Step 1: Language specificity - CausalExtractor._calculate_language_specificity
    Step 2: Temporal coherence - CausalExtractor._assess_temporal_coherence
    Step 3: Critical links diagnosis - TextMiningEngine.diagnose_critical_links
    Step 4: Failure points identification - CausalInferenceSetup.identify_failure_points
    Step 5: Dynamics pattern - CausalInferenceSetup._get_dynamics_pattern
    Step 6: Text chunking - SemanticProcessor.chunk_text
    Step 7: PDM structure detection - SemanticProcessor._detect_pdm_structure
    Step 8: Table detection - SemanticProcessor._detect_table
    Step 9: Traceability record - AdaptivePriorCalculator.generate_traceability_record
```

```
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate language specificity
        language_specificity = self._execute_method(
            "CausalExtractor", "_calculate_language_specificity", context
        )

        # Step 2: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 3: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 4: Identify failure points
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 5: Get dynamics pattern
        dynamics_pattern = self._execute_method(
            "CausalInferenceSetup", "_get_dynamics_pattern", context
        )

        # Step 6: Process text structure
        text_chunks = self._execute_method(
            "SemanticProcessor", "chunk_text", context
        )
        pdm_structure = self._execute_method(
            "SemanticProcessor", "_detect_pdm_structure", context,
            chunks=text_chunks
        )
        table_detection = self._execute_method(
            "SemanticProcessor", "_detect_table", context,
            chunks=text_chunks
        )

        # Step 7: Generate traceability record
        traceability = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            specificity=language_specificity
        )

        raw_evidence = {
            "context_adaptation": (language_specificity or {}).get("adaptation_level", 0),
            "differential_impacts_recognized": (critical_links or {}).get("differential_groups", []),
            "specific_groups_mentioned": (critical_links or {}).get("target_groups", []),
            "territorial_constraints": (failure_points or {}).get("territorial", []),
            "local_context_integration": (pdm_structure or {}).get("local_sections", []),
            "language_specificity": language_specificity,
```

```
            "temporal_coherence": temporal_coherence,
            "critical_links": critical_links,
            "failure_points": failure_points,
            "dynamics_pattern": dynamics_pattern,
            "structure_analysis": pdm_structure,
            "table_detection": table_detection,
            "text_chunks": text_chunks,
            "traceability": traceability
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "groups_identified": len((critical_links or {}).get("target_groups", [])) if critical_links else 0,
                "territorial_constraints": len((failure_points or {}).get("territorial", [])) if failure_points else 0,
                "total_text_chunks": len(text_chunks) if text_chunks else 0,
                "has_text_chunks": bool(text_chunks),
                "total_table_detections": len(table_detection) if isinstance(table_detection, list) else (1 if table_detection else 0),
                "has_table_detection": bool(table_detection),
                "has_pdm_structure": bool(pdm_structure),
                "has_dynamics_pattern": bool(dynamics_pattern),
                "has_traceability": bool(traceability)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# EXECUTOR REGISTRY
# ============================================================================

EXECUTOR_REGISTRY = {
    "D1-Q1": D1_Q1_QuantitativeBaselineExtractor,
    "D1-Q2": D1_Q2_ProblemDimensioningAnalyzer,
    "D1-Q3": D1_Q3_BudgetAllocationTracer,
    "D1-Q4": D1_Q4_InstitutionalCapacityIdentifier,
    "D1-Q5": D1_Q5_ScopeJustificationValidator,

    "D2-Q1": D2_Q1_StructuredPlanningValidator,
    "D2-Q2": D2_Q2_InterventionLogicInferencer,
    "D2-Q3": D2_Q3_RootCauseLinkageAnalyzer,
    "D2-Q4": D2_Q4_RiskManagementAnalyzer,
    "D2-Q5": D2_Q5_StrategicCoherenceEvaluator,

    "D3-Q1": D3_Q1_IndicatorQualityValidator,
    "D3-Q2": D3_Q2_TargetProportionalityAnalyzer,
    "D3-Q3": D3_Q3_TraceabilityValidator,
    "D3-Q4": D3_Q4_TechnicalFeasibilityEvaluator,
    "D3-Q5": D3_Q5_OutputOutcomeLinkageAnalyzer,

    "D4-Q1": D4_Q1_OutcomeMetricsValidator,
```

```
    "D4-Q2": D4_Q2_CausalChainValidator,
    "D4-Q3": D4_Q3_AmbitionJustificationAnalyzer,
    "D4-Q4": D4_Q4_ProblemSolvencyEvaluator,
    "D4-Q5": D4_Q5_VerticalAlignmentValidator,

    "D5-Q1": D5_Q1_LongTermVisionAnalyzer,
    "D5-Q2": D5_Q2_CompositeMeasurementValidator,
    "D5-Q3": D5_Q3_IntangibleMeasurementAnalyzer,
    "D5-Q4": D5_Q4_SystemicRiskEvaluator,
    "D5-Q5": D5_Q5_RealismAndSideEffectsAnalyzer,

    "D6-Q1": D6_Q1_ExplicitTheoryBuilder,
    "D6-Q2": D6_Q2_LogicalProportionalityValidator,
    "D6-Q3": D6_Q3_ValidationTestingAnalyzer,
    "D6-Q4": D6_Q4_FeedbackLoopAnalyzer,
    "D6-Q5": D6_Q5_ContextualAdaptabilityEvaluator,
}


# ============================================================================
# PHASE 2 ORCHESTRATION
# ============================================================================


def _build_method_executor() -> MethodExecutor:
    """Construct a canonical MethodExecutor via the factory wiring."""
    bundle = build_processor()
    method_executor = getattr(bundle, "method_executor", None)
    if not isinstance(method_executor, MethodExecutor):
        raise RuntimeError("ProcessorBundle did not provide a valid MethodExecutor instance.")
    return method_executor


def _canonical_metadata(executor_id: str) -> Dict[str, Any]:
    """Build canonical metadata block using canonical_notation."""
    metadata: Dict[str, Any] = {}
    try:
        dim_key = executor_id.split("-")[0]
        dim_info = get_dimension_info(dim_key)
        metadata["dimension_code"] = dim_info.code
        metadata["dimension_label"] = dim_info.label
    except (KeyError, ValueError, IndexError, AttributeError) as e:
        logger.warning(f"Failed to load canonical metadata for {executor_id}: {e}")
        # Continue with empty metadata rather than failing
    # Let critical system exceptions (KeyboardInterrupt, SystemExit) propagate

    if executor_id in CANONICAL_QUESTION_LABELS:
        metadata["canonical_question"] = CANONICAL_QUESTION_LABELS[executor_id]
    return metadata


def run_phase2_executors(context_package: Dict[str, Any],
                         policy_areas: List[str]) -> Dict[str, Any]:
    """
    Phase 2 Entry Point: Runs all 30 executors for each policy area.

```

```
    Args:
        context_package: Canonical package with document data from Phase 1
        policy_areas: List of policy area identifiers to analyze

    Returns:
        Dict mapping policy_area -> executor_id -> raw_evidence
    """
    results = {}
    method_executor = _build_method_executor()

    for policy_area in policy_areas:
        print(f"\n{'='*80}")
        print(f"Processing Policy Area: {policy_area}")
        print(f"{'='*80}\n")

        # Prepare context for this policy area
        area_context = {
            **context_package,
            "policy_area": policy_area
        }

        # Execute all 30 executors
        area_results = {}
        for executor_id, executor_class in EXECUTOR_REGISTRY.items():
            print(f"Running {executor_id}: {executor_class.__name__}...")

            try:
                # Instantiate executor with config
                config = load_executor_config(executor_id)
                executor = executor_class(executor_id, config, method_executor=method_executor)

                # Execute and collect results
                result = executor.execute(area_context)
                # Append canonical metadata consistently
                result_metadata = result.get("metadata", {})
                result_metadata.update(_canonical_metadata(executor_id))
                result["metadata"] = result_metadata
                area_results[executor_id] = result

                print(f"  â\234\223 Success: {len(result['metadata']['methods_executed'])} methods executed")

            except ExecutorFailure as e:
                print(f"  â\234\227 FAILED: {str(e)}")
                raise  # Re-raise to stop execution as per requirement

        results[policy_area] = area_results

    return results


def load_executor_config(executor_id: str) -> Dict[str, Any]:
    """
    Load executor configuration from JSON contract.

    Args:
        executor_id: Executor identifier (e.g., "D1-Q1")
```

```

    Returns:
        Configuration dictionary from JSON contract
    """
    import json
    from pathlib import Path

    config_path = Path(f"config/executor_contracts/{executor_id}.json")

    if not config_path.exists():
        raise FileNotFoundError(f"Executor config not found: {config_path}")

    with open(config_path, 'r', encoding='utf-8') as f:
        return json.load(f)


# ============================================================================
# EXAMPLE USAGE
# ============================================================================

if __name__ == "__main__":
    # Example context package from Phase 1
    context_package = {
        "document_path": "data/pdm_municipality_xyz.pdf",
        "document_text": "...",  # Full document text
        "tables": [],  # Extracted tables from Phase 1
        "embeddings": {},  # Precomputed embeddings
        "entities": [],  # Pre-extracted entities
        "metadata": {
            "municipality": "Municipality XYZ",
            "year": 2024,
            "pages": 150
        }
    }

    # Policy areas to analyze
    policy_areas = [
        "PA01",  # Education
        "PA02",  # Health
        "PA03",  # Infrastructure
        # ... up to 10+ policy areas
    ]

    # Run Phase 2
    try:
        results = run_phase2_executors(context_package, policy_areas)
        print("\n" + "="*80)
        print("PHASE 2 COMPLETED SUCCESSFULLY")
        print("="*80)
        print(f"Processed {len(policy_areas)} policy areas")
        print(f"Executed {len(EXECUTOR_REGISTRY)} executors per area")
        print(f"Total executions: {len(policy_areas) * len(EXECUTOR_REGISTRY)}")

    except ExecutorFailure as e:
        print("\n" + "="*80)
        print("PHASE 2 FAILED")
```

```
        print("="*80)
        print(f"Error: {str(e)}")
        print("Execution halted as per requirement: any method failure = executor failure")



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/executors_contract.py
================================================================================

from __future__ import annotations

from farfan_pipeline.core.orchestrator.base_executor_with_contract import BaseExecutorWithContract


class D1Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q1"


class D1Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q2"


class D1Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q3"


class D1Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q4"


class D1Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q5"


class D2Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q1"


class D2Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q2"

```

```

class D2Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q3"


class D2Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q4"


class D2Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q5"


class D3Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q1"


class D3Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q2"


class D3Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q3"


class D3Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q4"


class D3Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q5"


class D4Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q1"


class D4Q2_Executor_Contract(BaseExecutorWithContract):
```

```
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q2"


class D4Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q3"


class D4Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q4"


class D4Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q5"


class D5Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q1"


class D5Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q2"


class D5Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q3"


class D5Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q4"


class D5Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q5"


class D6Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
```

```
        return "D6-Q1"


class D6Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q2"


class D6Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q3"


class D6Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q4"


class D6Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q5"


# Aliases expected by core orchestrator
D1Q1_Executor = D1Q1_Executor_Contract
D1Q2_Executor = D1Q2_Executor_Contract
D1Q3_Executor = D1Q3_Executor_Contract
D1Q4_Executor = D1Q4_Executor_Contract
D1Q5_Executor = D1Q5_Executor_Contract
D2Q1_Executor = D2Q1_Executor_Contract
D2Q2_Executor = D2Q2_Executor_Contract
D2Q3_Executor = D2Q3_Executor_Contract
D2Q4_Executor = D2Q4_Executor_Contract
D2Q5_Executor = D2Q5_Executor_Contract
D3Q1_Executor = D3Q1_Executor_Contract
D3Q2_Executor = D3Q2_Executor_Contract
D3Q3_Executor = D3Q3_Executor_Contract
D3Q4_Executor = D3Q4_Executor_Contract
D3Q5_Executor = D3Q5_Executor_Contract
D4Q1_Executor = D4Q1_Executor_Contract
D4Q2_Executor = D4Q2_Executor_Contract
D4Q3_Executor = D4Q3_Executor_Contract
D4Q4_Executor = D4Q4_Executor_Contract
D4Q5_Executor = D4Q5_Executor_Contract
D5Q1_Executor = D5Q1_Executor_Contract
D5Q2_Executor = D5Q2_Executor_Contract
D5Q3_Executor = D5Q3_Executor_Contract
D5Q4_Executor = D5Q4_Executor_Contract
D5Q5_Executor = D5Q5_Executor_Contract
D6Q1_Executor = D6Q1_Executor_Contract
D6Q2_Executor = D6Q2_Executor_Contract
D6Q3_Executor = D6Q3_Executor_Contract
```

```
D6Q4_Executor = D6Q4_Executor_Contract
D6Q5_Executor = D6Q5_Executor_Contract



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/executors_snapshot/executors.py
================================================================================

"""
executors.py - Phase 2: Executor Orchestration for Policy Document Analysis

This module defines 30 executors (one per D{n}-Q{m} question) that orchestrate
methods from the core module to extract raw evidence from Colombian municipal
development plans (PDET/PDM documents).

Architecture:
- Each executor is independent and receives a canonical context package
- Methods execute in configured order; any failure causes executor failure
- Outputs are Python dicts/lists matching JSON contract specifications
- Executors are injected via MethodExecutor factory pattern

Usage:
    from factory import run_executor
    result = run_executor("D1-Q1", context_package)
"""

from typing import Dict, List, Any, Optional
from abc import ABC, abstractmethod

from farfan_pipeline.core.canonical_notation import CanonicalDimension, get_dimension_info
from farfan_pipeline.core.orchestrator.core import MethodExecutor
from farfan_pipeline.core.orchestrator.factory import build_processor

# Canonical question labels (only defined when verified in repo)
CANONICAL_QUESTION_LABELS = {
    "D3-Q2": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
    "D3-Q3": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
    "D3-Q4": "DIM03_Q04_TECHNICAL_FEASIBILITY",
    "D3-Q5": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
    "D4-Q1": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
    "D5-Q2": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
}

# Epistemic taxonomy per method (focused on executors expanded in this iteration)
EPISTEMIC_TAGS = {
    ("FinancialAuditor", "_calculate_sufficiency"): ["statistical", "normative"],
    ("FinancialAuditor", "_match_program_to_node"): ["structural"],
    ("FinancialAuditor", "_match_goal_to_budget"): ["structural", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability"): ["financial", "normative"],
    ("PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility"): ["financial", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_score_indicators"): ["normative", "semantic"],
    ("PDETMunicipalPlanAnalyzer", "_interpret_risk"): ["normative", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_consolidate_entities"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_syntax"): ["semantic"],
```

```
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_ner"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "identify_responsible_entities"): ["semantic", "structural"],
    ("PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities"): ["statistical", "causal"],
    ("PDETMunicipalPlanAnalyzer", "construct_causal_dag"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "estimate_causal_effects"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "generate_counterfactuals"): ["causal"],
    ("PDETMunicipalPlanAnalyzer", "_identify_confounders"): ["causal", "consistency"],
    ("PDETMunicipalPlanAnalyzer", "_effect_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_scenario_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords"): ["semantic"],
    ("AdaptivePriorCalculator", "calculate_likelihood_adaptativo"): ["statistical", "bayesian"],
    ("AdaptivePriorCalculator", "_adjust_domain_weights"): ["statistical"],
    ("BayesianMechanismInference", "_test_sufficiency"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_test_necessity"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_log_refactored_components"): ["implementation"],
    ("BayesianMechanismInference", "_infer_activity_sequence"): ["causal"],
    ("BayesianMechanismInference", "infer_mechanisms"): ["causal", "bayesian"],
    ("AdvancedDAGValidator", "calculate_acyclicity_pvalue"): ["statistical", "consistency"],
    ("AdvancedDAGValidator", "_is_acyclic"): ["structural", "consistency"],
    ("AdvancedDAGValidator", "_calculate_bayesian_posterior"): ["statistical", "bayesian"],
    ("AdvancedDAGValidator", "_calculate_confidence_interval"): ["statistical"],
    ("AdvancedDAGValidator", "_calculate_statistical_power"): ["statistical"],
    ("AdvancedDAGValidator", "_generate_subgraph"): ["structural"],
    ("AdvancedDAGValidator", "_get_node_validator"): ["implementation"],
    ("AdvancedDAGValidator", "_create_empty_result"): ["descriptive"],
    ("AdvancedDAGValidator", "_initialize_rng"): ["implementation"],
    ("AdvancedDAGValidator", "get_graph_stats"): ["structural"],
    ("AdvancedDAGValidator", "_calculate_node_importance"): ["structural"],
    ("AdvancedDAGValidator", "export_nodes"): ["structural", "descriptive"],
    ("AdvancedDAGValidator", "add_node"): ["structural"],
    ("AdvancedDAGValidator", "add_edge"): ["structural"],
    ("IndustrialGradeValidator", "execute_suite"): ["implementation", "normative"],
    ("IndustrialGradeValidator", "validate_connection_matrix"): ["consistency"],
    ("IndustrialGradeValidator", "run_performance_benchmarks"): ["implementation"],
    ("IndustrialGradeValidator", "_benchmark_operation"): ["implementation"],
    ("IndustrialGradeValidator", "validate_causal_categories"): ["consistency"],
    ("IndustrialGradeValidator", "_log_metric"): ["implementation"],
    ("PerformanceAnalyzer", "analyze_performance"): ["implementation", "normative"],
    ("PerformanceAnalyzer", "_calculate_loss_functions"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_ess"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_likelihood"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_r_hat"): ["statistical"],
    ("ReportingEngine", "generate_accountability_matrix"): ["normative", "structural"],
    ("ReportingEngine", "_calculate_quality_score"): ["normative", "statistical"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "descriptive"],
    ("PolicyAnalysisEmbedder", "compare_policy_interventions"): ["normative"],
    ("PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency"): ["consistency", "statistical"],
    ("PolicyAnalysisEmbedder", "process_document"): ["semantic", "structural"],
    ("PolicyAnalysisEmbedder", "semantic_search"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_apply_mmr"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_generate_query_from_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_filter_by_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_extract_numerical_values"): ["statistical"],
    ("PolicyAnalysisEmbedder", "_compute_overall_confidence"): ["statistical", "normative"],
    ("PolicyAnalysisEmbedder", "_embed_texts"): ["semantic"],
```

```
    ("SemanticAnalyzer", "_classify_policy_domain"): ["semantic"],
    ("SemanticAnalyzer", "_empty_semantic_cube"): ["descriptive"],
    ("SemanticAnalyzer", "_classify_cross_cutting_themes"): ["semantic"],
    ("SemanticAnalyzer", "_classify_value_chain_link"): ["semantic"],
    ("SemanticAnalyzer", "_vectorize_segments"): ["semantic"],
    ("SemanticAnalyzer", "_calculate_semantic_complexity"): ["semantic"],
    ("SemanticAnalyzer", "_process_segment"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_entity_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_quality_to_dict"): ["descriptive", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_deduplicate_tables"): ["structural", "implementation"],
    ("PDETMunicipalPlanAnalyzer", "_indicator_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_generate_recommendations"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_simulate_intervention"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_identify_causal_nodes"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "_match_text_to_node"): ["semantic", "structural"],
    ("TeoriaCambio", "_validar_orden_causal"): ["causal", "consistency"],
    ("TeoriaCambio", "_generar_sugerencias_internas"): ["normative"],
    ("TeoriaCambio", "_extraer_categorias"): ["semantic"],
    ("BayesianMechanismInference", "_extract_observations"): ["semantic", "causal"],
    ("BayesianMechanismInference", "_generate_necessity_remediation"): ["normative", "causal"],
    ("BayesianMechanismInference", "_quantify_uncertainty"): ["statistical", "bayesian"],
    ("CausalExtractor", "_build_type_hierarchy"): ["structural"],
    ("CausalExtractor", "_check_structural_violation"): ["structural", "consistency"],
    ("CausalExtractor", "_calculate_type_transition_prior"): ["statistical", "bayesian"],
    ("CausalExtractor", "_calculate_textual_proximity"): ["semantic"],
    ("CausalExtractor", "_calculate_language_specificity"): ["semantic"],
    ("CausalExtractor", "_calculate_composite_likelihood"): ["statistical", "semantic"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("CausalExtractor", "_calculate_semantic_distance"): ["semantic"],
    ("CausalExtractor", "_extract_goals"): ["semantic"],
    ("CausalExtractor", "_parse_goal_context"): ["semantic"],
    ("CausalExtractor", "_classify_goal_type"): ["semantic"],
    ("TemporalLogicVerifier", "_parse_temporal_marker"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_classify_temporal_type"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_extract_resources"): ["structural"],
    ("TemporalLogicVerifier", "_should_precede"): ["temporal", "consistency"],
    ("AdaptivePriorCalculator", "generate_traceability_record"): ["structural", "semantic"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "normative"],
    ("ReportingEngine", "generate_confidence_report"): ["normative", "descriptive"],
    ("PolicyTextProcessor", "segment_into_sentences"): ["semantic", "structural"],
    ("PolicyTextProcessor", "normalize_unicode"): ["implementation"],
    ("PolicyTextProcessor", "compile_pattern"): ["implementation"],
    ("PolicyTextProcessor", "extract_contextual_window"): ["semantic"],
    ("BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize"): ["causal", "normative"],
    ("BayesianCounterfactualAuditor", "refutation_and_sanity_checks"): ["causal", "consistency"],
    ("BayesianCounterfactualAuditor", "_evaluate_factual"): ["causal", "statistical"],
    ("BayesianCounterfactualAuditor", "_evaluate_counterfactual"): ["causal", "statistical"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("IndustrialPolicyProcessor", "_load_questionnaire"): ["descriptive", "implementation"],
    ("IndustrialPolicyProcessor", "_compile_pattern_registry"): ["structural", "semantic"],
    ("IndustrialPolicyProcessor", "_build_point_patterns"): ["semantic"],
    ("IndustrialPolicyProcessor", "_empty_result"): ["implementation"],
    ("IndustrialPolicyProcessor", "_compute_evidence_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_compute_avg_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_construct_evidence_bundle"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "generate_executive_report"): ["normative"],
```

```
    ("IndustrialPolicyProcessor", "export_results"): ["implementation"],
}


class BaseExecutor(ABC):
    """
    Base class for all executors with standardized execution template.
    All executors must implement execute() and return structured evidence.
    """

    def __init__(self, executor_id: str, config: Dict[str, Any], method_executor: MethodExecutor):
        self.executor_id = executor_id
        self.config = config
        if not isinstance(method_executor, MethodExecutor):
            raise RuntimeError("A valid MethodExecutor instance is required for executor injection.")
        self.method_executor = method_executor
        self.execution_log = []
        self.dimension_info = None
        try:
            dim_key = executor_id.split("-")[0].replace("D", "D")
            self.dimension_info = get_dimension_info(dim_key)
        except Exception:
            self.dimension_info = None

    @abstractmethod
    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute configured methods and return raw evidence.

        Args:
            context: Canonical package with document, tables, metadata

        Returns:
            Dict with raw_evidence, metadata, execution_metrics

        Raises:
            ExecutorFailure: If any method fails
        """
        pass

    def _log_method_execution(self, class_name: str, method_name: str,
                              success: bool, result: Any = None, error: str = None):
        """Track method execution for debugging and traceability."""
        self.execution_log.append({
            "class": class_name,
            "method": method_name,
            "success": success,
            "result_type": type(result).__name__ if result else None,
            "error": error
        })

    def _execute_method(self, class_name: str, method_name: str,
                        context: Dict[str, Any], **kwargs) -> Any:
        """
        Execute a single method with error handling.

```

```
        Raises:
            ExecutorFailure: If method execution fails
        """
        try:
            # Method injection happens via factory - placeholder for actual execution
            method = self._get_method(class_name, method_name)
            result = method(context, **kwargs)
            self._log_method_execution(class_name, method_name, True, result)
            return result
        except Exception as e:
            self._log_method_execution(class_name, method_name, False, error=str(e))
            raise ExecutorFailure(
                f"Executor {self.executor_id} failed: {class_name}.{method_name} - {str(e)}"
            )

    def _get_method(self, class_name: str, method_name: str):
        """Retrieve method using MethodExecutor to enforce routed execution."""
        if not isinstance(self.method_executor, MethodExecutor):
            raise RuntimeError(f"Invalid method executor provided: {type(self.method_executor).__name__}")

        def _wrapped(context: Dict[str, Any], **kwargs: Any) -> Any:
            payload: Dict[str, Any] = {}
            if context:
                payload.update(context)
            if kwargs:
                payload.update(kwargs)
            return self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

        return _wrapped


class ExecutorFailure(Exception):
    """Raised when any method in an executor fails."""
    pass


# ============================================================================
# DIMENSION 1: DIAGNOSTICS & INPUTS
# ============================================================================

class D1_Q1_QuantitativeBaselineExtractor(BaseExecutor):
    """
    Extracts numeric data, reference years, and official sources as baseline.

    Methods (from D1-Q1):
    - TextMiningEngine.diagnose_critical_links
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor.process
    - IndustrialPolicyProcessor._match_patterns_in_sentences
    - IndustrialPolicyProcessor._extract_point_evidence
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
```

```
    - FinancialAuditor._parse_amount
    - PDETMunicipalPlanAnalyzer._extract_financial_amounts
    - PDETMunicipalPlanAnalyzer._extract_from_budget_table
    - PolicyContradictionDetector._extract_quantitative_claims
    - PolicyContradictionDetector._parse_number
    - PolicyContradictionDetector._statistical_significance_test
    - BayesianNumericalAnalyzer.evaluate_policy_metric
    - BayesianNumericalAnalyzer.compare_policies
    - SemanticProcessor.chunk_text
    - SemanticProcessor.embed_single
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Identify critical data-bearing sections
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context,
            links=critical_links
        )

        # Step 2: Extract structured quantitative claims
        processed_sections = self._execute_method(
            "IndustrialPolicyProcessor", "process", context
        )
        pattern_matches = self._execute_method(
            "IndustrialPolicyProcessor", "_match_patterns_in_sentences", context,
            sections=processed_sections
        )
        point_evidence = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_point_evidence", context,
            matches=pattern_matches
        )

        # Step 3: Parse numerical amounts and baseline data
        parsed_amounts = self._execute_method(
            "FinancialAuditor", "_parse_amount", context,
            evidence=point_evidence
        )
        financial_amounts = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_financial_amounts", context
        )
        budget_table_data = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_from_budget_table", context
        )

        # Step 4: Extract temporal context (reference years)
        goals = self._execute_method(
            "CausalExtractor", "_extract_goals", context
        )
        goal_contexts = self._execute_method(
            "CausalExtractor", "_parse_goal_context", context,
            goals=goals
```

```
        )

        # Step 5: Validate quantitative claims
        quant_claims = self._execute_method(
            "PolicyContradictionDetector", "_extract_quantitative_claims", context
        )
        parsed_numbers = self._execute_method(
            "PolicyContradictionDetector", "_parse_number", context,
            claims=quant_claims
        )
        significance_test = self._execute_method(
            "PolicyContradictionDetector", "_statistical_significance_test", context,
            numbers=parsed_numbers
        )

        # Step 6: Evaluate baseline quality and compare
        metric_evaluation = self._execute_method(
            "BayesianNumericalAnalyzer", "evaluate_policy_metric", context,
            metrics=parsed_numbers
        )
        policy_comparison = self._execute_method(
            "BayesianNumericalAnalyzer", "compare_policies", context,
            evaluations=metric_evaluation
        )

        # Step 7: Semantic validation of sources
        text_chunks = self._execute_method(
            "SemanticProcessor", "chunk_text", context
        )
        embeddings = self._execute_method(
            "SemanticProcessor", "embed_single", context,
            chunks=text_chunks
        )

        # Assemble raw evidence
        raw_evidence = {
            "numeric_data": parsed_numbers,
            "reference_years": [gc.get("year") for gc in goal_contexts if gc.get("year")],
            "official_sources": point_evidence.get("sources", []),
            "financial_baseline": financial_amounts,
            "budget_tables": budget_table_data,
            "significance_results": significance_test,
            "metric_evaluation": metric_evaluation,
            "source_embeddings": embeddings
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_numeric_claims": len(parsed_numbers),
                "sources_identified": len(point_evidence.get("sources", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
```

```
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q2_ProblemDimensioningAnalyzer(BaseExecutor):
    """
    Quantifies problem magnitude, gaps, and identifies data limitations.

    Methods (from D1-Q2):
    - OperationalizationAuditor._audit_direct_evidence
    - OperationalizationAuditor._audit_systemic_risk
    - FinancialAuditor._detect_allocation_gaps
    - BayesianMechanismInference._detect_gaps
    - PDETMunicipalPlanAnalyzer._generate_optimal_remediations
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - BayesianCounterfactualAuditor.counterfactual_query
    - BayesianCounterfactualAuditor._test_effect_stability
    - PolicyContradictionDetector._detect_numerical_inconsistencies
    - PolicyContradictionDetector._calculate_numerical_divergence
    - BayesianConfidenceCalculator.calculate_posterior
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Audit evidence completeness
        direct_evidence_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_direct_evidence", context
        )
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 2: Detect gaps in resource allocation and mechanisms
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )
        mechanism_gaps = self._execute_method(
            "BayesianMechanismInference", "_detect_gaps", context
        )

        # Step 3: Generate optimal remediations and simulate interventions
        remediations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_optimal_remediations", context,
            gaps=allocation_gaps
        )
        simulation_results = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            remediations=remediations
        )

        # Step 4: Counterfactual analysis for problem dimensioning
        counterfactual = self._execute_method(
            "BayesianCounterfactualAuditor", "counterfactual_query", context
```

```
        )
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            counterfactual=counterfactual
        )

        # Step 5: Detect numerical inconsistencies
        numerical_inconsistencies = self._execute_method(
            "PolicyContradictionDetector", "_detect_numerical_inconsistencies", context
        )
        divergence_calc = self._execute_method(
            "PolicyContradictionDetector", "_calculate_numerical_divergence", context,
            inconsistencies=numerical_inconsistencies
        )

        # Step 6: Calculate confidence and analyze performance
        posterior_confidence = self._execute_method(
            "BayesianConfidenceCalculator", "calculate_posterior", context,
            evidence=direct_evidence_audit
        )
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "magnitude_indicators": {
                "allocation_gaps": allocation_gaps,
                "mechanism_gaps": mechanism_gaps,
                "numerical_inconsistencies": numerical_inconsistencies
            },
            "deficit_quantification": divergence_calc,
            "data_limitations": {
                "evidence_gaps": direct_evidence_audit.get("gaps", []),
                "systemic_risks": systemic_risk_audit
            },
            "simulation_results": simulation_results,
            "confidence_scores": posterior_confidence,
            "performance_metrics": performance_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "gaps_identified": len(allocation_gaps) + len(mechanism_gaps),
                "inconsistencies_found": len(numerical_inconsistencies)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q3_BudgetAllocationTracer(BaseExecutor):
```

```
    """
    Traces monetary resources assigned to programs in Investment Plan (PPI).

    Methods (from D1-Q3):
    - FinancialAuditor.trace_financial_allocation
    - FinancialAuditor._process_financial_table
    - FinancialAuditor._match_program_to_node
    - FinancialAuditor._match_goal_to_budget
    - FinancialAuditor._perform_counterfactual_budget_check
    - FinancialAuditor._calculate_sufficiency
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer._extract_budget_for_pillar
    - PDETMunicipalPlanAnalyzer._identify_funding_source
    - PDETMunicipalPlanAnalyzer._classify_tables
    - PDETMunicipalPlanAnalyzer._analyze_funding_sources
    - PDETMunicipalPlanAnalyzer._score_financial_component
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Trace complete financial allocation chain
        allocation_trace = self._execute_method(
            "FinancialAuditor", "trace_financial_allocation", context
        )
        processed_tables = self._execute_method(
            "FinancialAuditor", "_process_financial_table", context
        )

        # Step 2: Match programs to budget nodes
        program_matches = self._execute_method(
            "FinancialAuditor", "_match_program_to_node", context,
            tables=processed_tables
        )
        goal_budget_matches = self._execute_method(
            "FinancialAuditor", "_match_goal_to_budget", context,
            programs=program_matches
        )

        # Step 3: Counterfactual checks and sufficiency calculation
        counterfactual_check = self._execute_method(
            "FinancialAuditor", "_perform_counterfactual_budget_check", context,
            matches=goal_budget_matches
        )
        sufficiency_calc = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context,
            allocation=allocation_trace
        )

        # Step 4: Analyze financial feasibility
        feasibility_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        pillar_budgets = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_budget_for_pillar", context
```

```
        )
        funding_sources = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_funding_source", context
        )

        # Step 5: Classify and analyze tables
        table_classification = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
            tables=processed_tables
        )
        funding_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
            sources=funding_sources
        )
        financial_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
            analysis=funding_analysis
        )

        # Step 6: Aggregate risk and prioritize
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            sufficiency=sufficiency_calc
        )

        raw_evidence = {
            "budget_allocations": allocation_trace,
            "program_mappings": program_matches,
            "goal_budget_links": goal_budget_matches,
            "sufficiency_analysis": sufficiency_calc,
            "pillar_budgets": pillar_budgets,
            "funding_sources": funding_sources,
            "financial_feasibility": feasibility_analysis,
            "financial_score": financial_score,
            "risk_priorities": risk_aggregation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "programs_traced": len(program_matches),
                "funding_sources_identified": len(funding_sources)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q4_InstitutionalCapacityIdentifier(BaseExecutor):
    """
    Identifies installed capacity (entities, staff, equipment) and limitations.

```

```
    Methods (from D1-Q4):
    - PDETMunicipalPlanAnalyzer.identify_responsible_entities
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._classify_entity_type
    - PDETMunicipalPlanAnalyzer._score_entity_specificity
    - PDETMunicipalPlanAnalyzer._consolidate_entities
    - MechanismPartExtractor.extract_entity_activity
    - MechanismPartExtractor._normalize_entity
    - MechanismPartExtractor._validate_entity_activity
    - MechanismPartExtractor._calculate_ea_confidence
    - OperationalizationAuditor.audit_evidence_traceability
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Identify all responsible entities
        entities_identified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
        )

        # Step 2: Extract entities using NER and syntax
        ner_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context
        )
        syntax_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context
        )

        # Step 3: Classify and score entities
        entity_types = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_entity_type", context,
            entities=ner_entities + syntax_entities
        )
        specificity_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_entity_specificity", context,
            entities=entity_types
        )
        consolidated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
            entities=entity_types
        )

        # Step 4: Extract entity-activity relationships
        entity_activities = self._execute_method(
            "MechanismPartExtractor", "extract_entity_activity", context,
            entities=consolidated
        )
        normalized = self._execute_method(
            "MechanismPartExtractor", "_normalize_entity", context,
            activities=entity_activities
        )
        validated = self._execute_method(
            "MechanismPartExtractor", "_validate_entity_activity", context,
            normalized=normalized
```

```
        )
        ea_confidence = self._execute_method(
            "MechanismPartExtractor", "_calculate_ea_confidence", context,
            validated=validated
        )

        # Step 5: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            entity_activities=validated
        )

        raw_evidence = {
            "entities_identified": consolidated,
            "entity_types": entity_types,
            "specificity_scores": specificity_scores,
            "entity_activities": validated,
            "activity_confidence": ea_confidence,
            "capacity_indicators": {
                "staff_mentions": [e for e in consolidated if e.get("type") == "staff"],
                "equipment_mentions": [e for e in consolidated if e.get("type") == "equipment"],
                "organizational_units": [e for e in consolidated if e.get("type") == "organization"]
            },
            "limitations_identified": traceability_audit.get("gaps", []),
            "traceability_audit": traceability_audit
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "entities_count": len(consolidated),
                "activities_extracted": len(validated)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q5_ScopeJustificationValidator(BaseExecutor):
    """
    Validates scope justification via legal framework and constraint recognition.

    Methods (from D1-Q5):
    - TemporalLogicVerifier._check_deadline_constraints
    - TemporalLogicVerifier.verify_temporal_consistency
    - CausalInferenceSetup.identify_failure_points
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - IndustrialPolicyProcessor._extract_metadata
    """

```

```
    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Verify temporal constraints
        deadline_constraints = self._execute_method(
            "TemporalLogicVerifier", "_check_deadline_constraints", context
        )
        temporal_consistency = self._execute_method(
            "TemporalLogicVerifier", "verify_temporal_consistency", context
        )

        # Step 2: Identify failure points in scope
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 3: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 4: Analyze link text for justifications
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context
        )

        # Step 5: Analyze causal dimensions and extract metadata
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )
        metadata_extracted = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context,
            dimensions=causal_dimensions
        )

        raw_evidence = {
            "legal_framework_citations": metadata_extracted.get("legal_refs", []),
            "temporal_constraints": {
                "deadline_checks": deadline_constraints,
                "consistency": temporal_consistency,
                "coherence": temporal_coherence
            },
            "budgetary_constraints": metadata_extracted.get("budget_limits", []),
            "competence_constraints": metadata_extracted.get("competence_refs", []),
            "failure_points": failure_points,
            "scope_justifications": link_analysis.get("justifications", []),
            "causal_dimensions": causal_dimensions
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "constraints_identified": len(deadline_constraints),
                "legal_citations": len(metadata_extracted.get("legal_refs", []))
```

```
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 2: ACTIVITY DESIGN
# ============================================================================

class D2_Q1_StructuredPlanningValidator(BaseExecutor):
    """
    Validates structured format of activities (table/matrix with required columns).

    Methods (from D2-Q1):
    - PDFProcessor.extract_tables
    - FinancialAuditor._process_financial_table
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._classify_tables
    - PDETMunicipalPlanAnalyzer._is_likely_header
    - PDETMunicipalPlanAnalyzer._clean_dataframe
    - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract all tables
        extracted_tables = self._execute_method(
            "PDFProcessor", "extract_tables", context
        )

        # Step 2: Process financial tables
        processed_tables = self._execute_method(
            "FinancialAuditor", "_process_financial_table", context,
            tables=extracted_tables
        )

        # Step 3: Deduplicate and classify tables
        deduplicated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=processed_tables
        )
        classified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
            tables=deduplicated
        )

        # Step 4: Identify headers and clean dataframes
        header_checks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_is_likely_header", context,
            tables=classified
        )
        cleaned = self._execute_method(
```

```
            "PDETMunicipalPlanAnalyzer", "_clean_dataframe", context,
            tables=classified
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
            tables=cleaned
        )

        raw_evidence = {
            "tables_extracted": len(extracted_tables),
            "activity_tables": [t for t in classified if t.get("type") == "activity"],
            "matrix_structure": accountability_matrix,
            "required_columns_present": {
                "responsible_entity": any("responsible" in str(t.get("columns", [])).lower()
                                         for t in cleaned),
                "deliverable": any("deliverable" in str(t.get("columns", [])).lower()
                                  for t in cleaned),
                "timeline": any("timeline" in str(t.get("columns", [])).lower()
                               for t in cleaned),
                "cost": any("cost" in str(t.get("columns", [])).lower()
                           for t in cleaned)
            },
            "table_quality": {
                "clean_tables": len(cleaned),
                "with_headers": sum(1 for h in header_checks if h)
            }
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_tables": len(extracted_tables),
                "activity_tables": len([t for t in classified if t.get("type") == "activity"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q2_InterventionLogicInferencer(BaseExecutor):
    """
    Infers intervention logic: instrument (how), target (who), causality (why).

    Methods (from D2-Q2):
    - BayesianMechanismInference.infer_mechanisms
    - BayesianMechanismInference._infer_single_mechanism
    - BayesianMechanismInference._infer_mechanism_type
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - CausalExtractor.extract_causal_hierarchy
```

```
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio._es_conexion_valida
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - BeachEvidentialTest.classify_test
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context
        )
        single_mechanisms = []
        for mech in mechanisms:
            single = self._execute_method(
                "BayesianMechanismInference", "_infer_single_mechanism", context,
                mechanism=mech
            )
            single_mechanisms.append(single)

        mechanism_types = self._execute_method(
            "BayesianMechanismInference", "_infer_mechanism_type", context,
            mechanisms=single_mechanisms
        )

        # Step 2: Test sufficiency and necessity
        sufficiency_tests = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context,
            mechanisms=single_mechanisms
        )
        necessity_tests = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context,
            mechanisms=single_mechanisms
        )

        # Step 3: Extract causal hierarchy
        causal_hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        # Step 4: Build causal graph
        causal_graph = self._execute_method(
            "TeoriaCambio", "construir_grafo_causal", context,
            hierarchy=causal_hierarchy
        )
        connection_validation = self._execute_method(
            "TeoriaCambio", "_es_conexion_valida", context,
            graph=causal_graph
        )

        # Step 5: Construct DAG
        causal_dag = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
            graph=causal_graph
```

```
        )

        # Step 6: Classify evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "classify_test", context,
            mechanisms=single_mechanisms
        )

        # Step 7: Analyze causal dimensions
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )

        raw_evidence = {
            "intervention_instruments": [m.get("instrument") for m in single_mechanisms],
            "target_populations": [m.get("target") for m in single_mechanisms],
            "causal_logic": {
                "mechanisms": single_mechanisms,
                "mechanism_types": mechanism_types,
                "sufficiency": sufficiency_tests,
                "necessity": necessity_tests
            },
            "causal_hierarchy": causal_hierarchy,
            "causal_graph": causal_graph,
            "causal_dag": causal_dag,
            "evidential_strength": evidential_tests,
            "dimensions": causal_dimensions
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(single_mechanisms),
                "instruments_found": len([m for m in single_mechanisms if m.get("instrument")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q3_RootCauseLinkageAnalyzer(BaseExecutor):
    """
    Analyzes linkage between activities and root causes/structural determinants.

    Methods (from D2-Q3):
    - CausalExtractor._extract_causal_links
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._initialize_prior
    - CausalExtractor._calculate_type_transition_prior
    - PDETMunicipalPlanAnalyzer._identify_causal_edges
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - BayesianCounterfactualAuditor.construct_scm
```

```
    - BayesianCounterfactualAuditor._create_default_equations
    - SemanticAnalyzer.extract_semantic_cube
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract causal links
        causal_links = self._execute_method(
            "CausalExtractor", "_extract_causal_links", context
        )

        # Step 2: Calculate likelihoods
        composite_likelihood = self._execute_method(
            "CausalExtractor", "_calculate_composite_likelihood", context,
            links=causal_links
        )
        prior_init = self._execute_method(
            "CausalExtractor", "_initialize_prior", context
        )
        type_transition_prior = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            links=causal_links
        )

        # Step 3: Identify and refine causal edges
        causal_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_edges", context,
            links=causal_links
        )
        refined_probabilities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            edges=causal_edges
        )

        # Step 4: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            edges=refined_probabilities
        )
        default_equations = self._execute_method(
            "BayesianCounterfactualAuditor", "_create_default_equations", context,
            scm=scm
        )

        # Step 5: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        raw_evidence = {
            "root_causes_identified": [link.get("root_cause") for link in causal_links],
            "activity_linkages": causal_links,
            "link_probabilities": refined_probabilities,
            "composite_likelihood": composite_likelihood,
            "structural_model": scm,
```

```
            "model_equations": default_equations,
            "semantic_relationships": semantic_cube,
            "determinants_addressed": [link for link in causal_links if link.get("addresses_determinant")]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "causal_links_found": len(causal_links),
                "root_causes_count": len(set(link.get("root_cause") for link in causal_links))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q4_RiskManagementAnalyzer(BaseExecutor):
    """
    Identifies implementation risks and mitigation measures.

    Methods (from D2-Q4):
    - PDETMunicipalPlanAnalyzer._bayesian_risk_inference
    - PDETMunicipalPlanAnalyzer.sensitivity_analysis
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Bayesian risk inference
        risk_inference = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_bayesian_risk_inference", context
        )

        # Step 2: Sensitivity analysis
        sensitivity = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "sensitivity_analysis", context,
            risks=risk_inference
        )

        # Step 3: Risk interpretation
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            inference=risk_inference
        )
```

```

        # Step 4: Compute robustness metrics
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            sensitivity=sensitivity
        )
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            robustness=robustness
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            sensitivity=sensitivity
        )

        # Step 5: Audit systemic risks
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 6: Aggregate and prioritize risks
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            risks=risk_inference
        )

        # Step 7: Refutation and sanity checks
        refutation_checks = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            aggregation=risk_aggregation
        )

        # Step 8: Additional sensitivity analysis
        adaptive_sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=risk_inference
        )

        raw_evidence = {
            "operational_risks": [r for r in risk_inference if r.get("type") == "operational"],
            "social_risks": [r for r in risk_inference if r.get("type") == "social"],
            "security_risks": [r for r in risk_inference if r.get("type") == "security"],
            "mitigation_measures": risk_interpretation.get("mitigations", []),
            "risk_priorities": risk_aggregation,
            "robustness_metrics": {
                "robustness_value": robustness,
                "e_value": e_value
            },
            "sensitivity_analysis": sensitivity,
            "systemic_risks": systemic_risk_audit,
            "validation_checks": refutation_checks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
```

```
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "risks_identified": len(risk_inference),
                "mitigations_proposed": len(risk_interpretation.get("mitigations", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q5_StrategicCoherenceEvaluator(BaseExecutor):
    """
    Evaluates strategic coherence: complementarity and logical sequence.

    Methods (from D2-Q5):
    - PolicyContradictionDetector._detect_logical_incompatibilities
    - PolicyContradictionDetector._calculate_coherence_metrics
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._calculate_graph_fragmentation
    - OperationalizationAuditor.audit_sequence_logic
    - BayesianMechanismInference._calculate_coherence_factor
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Detect logical incompatibilities
        incompatibilities = self._execute_method(
            "PolicyContradictionDetector", "_detect_logical_incompatibilities", context
        )

        # Step 2: Calculate coherence metrics
        coherence_metrics = self._execute_method(
            "PolicyContradictionDetector", "_calculate_coherence_metrics", context
        )
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )
        graph_fragmentation = self._execute_method(
            "PolicyContradictionDetector", "_calculate_graph_fragmentation", context
        )

        # Step 3: Audit sequence logic
        sequence_audit = self._execute_method(
            "OperationalizationAuditor", "audit_sequence_logic", context
        )

        # Step 4: Calculate coherence factors
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            metrics=coherence_metrics
        )
```

```
        causal_coherence_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 5: Adaptive likelihood calculation
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context,
            coherence=causal_coherence_score
        )

        raw_evidence = {
            "complementarity_evidence": coherence_metrics.get("complementarity", []),
            "sequential_logic": sequence_audit,
            "logical_incompatibilities": incompatibilities,
            "coherence_scores": {
                "overall_coherence": coherence_metrics,
                "objective_alignment": objective_alignment,
                "causal_coherence": causal_coherence_score,
                "coherence_factor": coherence_factor
            },
            "graph_metrics": {
                "fragmentation": graph_fragmentation
            },
            "adaptive_likelihood": adaptive_likelihood
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "incompatibilities_found": len(incompatibilities),
                "coherence_score": coherence_metrics.get("score", 0)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 3: PRODUCTS & OUTPUTS
# ============================================================================

class D3_Q1_IndicatorQualityValidator(BaseExecutor):
    """
    Validates indicator quality: baseline, target, source of verification.

    Methods (from D3-Q1):
    - PDETMunicipalPlanAnalyzer._score_indicators
    - OperationalizationAuditor.audit_evidence_traceability
    - CausalInferenceSetup.assign_probative_value
    - BeachEvidentialTest.apply_test_logic
    - TextMiningEngine.diagnose_critical_links
    - IndustrialPolicyProcessor._extract_metadata
```

```
    - IndustrialPolicyProcessor._calculate_quality_score
    - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score indicators
        indicator_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )

        # Step 2: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            indicators=indicator_scores
        )

        # Step 3: Assign probative value
        probative_values = self._execute_method(
            "CausalInferenceSetup", "assign_probative_value", context,
            indicators=indicator_scores
        )

        # Step 4: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context,
            indicators=indicator_scores
        )

        # Step 5: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 6: Extract and score metadata
        metadata = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context
        )
        quality_score = self._execute_method(
            "IndustrialPolicyProcessor", "_calculate_quality_score", context,
            metadata=metadata
        )

        # Step 7: Generate traceability record
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            indicators=indicator_scores
        )

        raw_evidence = {
            "indicators_with_baseline": [i for i in indicator_scores if i.get("has_baseline")],
            "indicators_with_target": [i for i in indicator_scores if i.get("has_target")],
            "indicators_with_source": [i for i in indicator_scores if i.get("has_source")],
            "indicator_quality_scores": indicator_scores,
            "traceability": traceability_audit,
```

```
            "probative_values": probative_values,
            "evidential_strength": evidential_tests,
            "overall_quality_score": quality_score,
            "traceability_record": traceability_record
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_indicators": len(indicator_scores),
                "complete_indicators": len([i for i in indicator_scores
                    if i.get("has_baseline") and i.get("has_target") and i.get("has_source")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q2_TargetProportionalityAnalyzer(BaseExecutor):
    """
    DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY â\200\224 Analyzes proportionality of targets to the diagnosed universe using canonical D3 notation.
    Epistemic mix: structural coverage, financial/normative feasibility, statistical Bayes tests, and semantic indicator quality.

    Methods (from D3-Q2):
    - AdvancedDAGValidator._calculate_bayesian_posterior
    - AdvancedDAGValidator._calculate_confidence_interval
    - AdaptivePriorCalculator._adjust_domain_weights
    - PDETMunicipalPlanAnalyzer._get_spanish_stopwords
    - BayesianMechanismInference._log_refactored_components
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - FinancialAuditor._calculate_sufficiency
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - PDETMunicipalPlanAnalyzer._assess_financial_sustainability
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    - IndustrialPolicyProcessor._calculate_quality_score
    - TeoriaCambio._generar_sugerencias_internas
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._indicator_to_dict
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - IndustrialPolicyProcessor._compile_pattern_registry
    - IndustrialPolicyProcessor._build_point_patterns
    - IndustrialPolicyProcessor._empty_result
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Financial feasibility snapshot and indicator quality
```

```
        financial_feasibility = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        indicator_quality = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )
        spanish_stopwords = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords", context
        )
        funding_sources = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
            financial_indicators=financial_feasibility.get("financial_indicators", []),
            tables=context.get("tables", [])
        )
        financial_component = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
            financial_analysis=financial_feasibility
        )
        pattern_registry = self._execute_method(
            "IndustrialPolicyProcessor", "_compile_pattern_registry", context
        )
        point_patterns = self._execute_method(
            "IndustrialPolicyProcessor", "_build_point_patterns", context
        )
        empty_policy_result = self._execute_method(
            "IndustrialPolicyProcessor", "_empty_result", context
        )
        dedup_tables = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=context.get("tables", [])
        )
        first_indicator = None
        if isinstance(financial_feasibility.get("financial_indicators", []), list):
            inds = financial_feasibility.get("financial_indicators", [])
            first_indicator = inds[0] if inds else None
        indicator_dict = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_indicator_to_dict", context,
            ind=first_indicator if first_indicator else {}
        )
        proportionality_recommendations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={"financial_analysis": financial_feasibility, "quality_score": quality_score} if 'quality_score' in locals() else {}
        )

        # Step 1: Calculate sufficiency
        sufficiency_calc = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )

        # Step 2: Test sufficiency and necessity of targets
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )
```

```

        # Step 3: Assess financial sustainability
        sustainability_assessment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability", context
        )
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risk=financial_feasibility.get("risk_assessment", {}).get("risk_score", 0.0)
        )

        # Step 4: Calculate adaptive likelihood
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context
        )
        domain_scores = {
            "structural": sufficiency_calc.get("coverage_ratio", 0.0),
            "financial": financial_feasibility.get("sustainability_score", 0.0),
            "semantic": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0
        }
        adjusted_weights = self._execute_method(
            "AdaptivePriorCalculator", "_adjust_domain_weights", context,
            domain_scores=domain_scores
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D3": {"dimension_confidence": domain_scores.get("structural", 0.0)}}
        )

        # Step 5: Calculate quality score
        quality_score = self._execute_method(
            "IndustrialPolicyProcessor", "_calculate_quality_score", context
        )

        # Step 6: Generate internal suggestions
        internal_suggestions = self._execute_method(
            "TeoriaCambio", "_generar_sugerencias_internas", context
        )
        # Bayesian posterior diagnostics for proportionality evidence
        posterior_probability = self._execute_method(
            "AdvancedDAGValidator", "_calculate_bayesian_posterior", context,
            likelihood=sufficiency_calc.get("coverage_ratio", 0.5),
            prior=0.5
        )
        confidence_interval = self._execute_method(
            "AdvancedDAGValidator", "_calculate_confidence_interval", context,
            s=int(sufficiency_calc.get("covered_targets", 0)),
            n=max(1, int(sufficiency_calc.get("targets_total", len(context.get("product_targets", []))))),
            conf=0.95
        )
        self._execute_method(
            "BayesianMechanismInference", "_log_refactored_components", context
        )

        raw_evidence = {
            "target_population_size": context.get("diagnosed_universe", 0),
            "product_targets": context.get("product_targets", []),
```

```
            "coverage_ratio": sufficiency_calc.get("coverage_ratio", 0),
            "dosage_analysis": sufficiency_calc.get("dosage", {}),
            "sufficiency_test": sufficiency_test,
            "necessity_test": necessity_test,
            "sustainability": sustainability_assessment,
            "financial_feasibility": financial_feasibility,
            "indicator_quality": indicator_quality,
            "risk_interpretation": risk_interpretation,
            "proportionality_score": quality_score,
            "recommendations": internal_suggestions,
            "stopwords_spanish": spanish_stopwords,
            "funding_sources_analysis": funding_sources,
            "financial_component_score": financial_component,
            "pattern_registry": pattern_registry,
            "point_patterns": point_patterns,
            "empty_policy_result": empty_policy_result,
            "avg_confidence": avg_confidence,
            "deduplicated_tables": dedup_tables,
            "indicator_sample": indicator_dict,
            "proportionality_recommendations": proportionality_recommendations,
            "adjusted_domain_weights": adjusted_weights,
            "posterior_proportionality": posterior_probability,
            "coverage_interval": confidence_interval
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "targets_analyzed": len(context.get("product_targets", [])),
                "coverage_adequate": sufficiency_calc.get("is_sufficient", False),
                "canonical_question": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q3_TraceabilityValidator(BaseExecutor):
    """
    DIM03_Q03_TRACEABILITY_BUDGET_ORG â\200\224 Validates budgetary and organizational traceability of products under canonical D3 notation.
    Epistemic mix: structural budget tracing, organizational semantics, and accountability synthesis.

    Methods (from D3-Q3):
    - PolicyAnalysisEmbedder.process_document
    - PolicyAnalysisEmbedder.semantic_search
    - PolicyAnalysisEmbedder._apply_mmr
    - PolicyAnalysisEmbedder._generate_query_from_pdq
    - SemanticAnalyzer._empty_semantic_cube
    - FinancialAuditor._match_program_to_node
    - FinancialAuditor._match_goal_to_budget
```

```
    - PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables
    - PDETMunicipalPlanAnalyzer._consolidate_entities
    - AdaptivePriorCalculator.generate_traceability_record
    - PolicyAnalysisEmbedder.generate_pdq_report
    - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Match programs to budget nodes
        program_matches = self._execute_method(
            "FinancialAuditor", "_match_program_to_node", context
        )
        goal_budget_matches = self._execute_method(
            "FinancialAuditor", "_match_goal_to_budget", context,
            programs=program_matches
        )

        # Step 2: Extract responsibility assignments
        responsibility_data = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables", context
        )
        consolidated_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
            entities=responsibility_data
        )
        responsible_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
        )
        responsibility_clarity = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity", context,
            entities=consolidated_entities
        )
        # Semantic traceability via embeddings
        semantic_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "process_document", context,
            document_text=document_text,
            document_metadata=document_metadata
        )
        pdq_query = self._execute_method(
            "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
            pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
        )
        semantic_hits = self._execute_method(
            "PolicyAnalysisEmbedder", "semantic_search", context,
            query=pdq_query,
            document_chunks=semantic_chunks or []
        )
        diversified_hits = self._execute_method(
            "PolicyAnalysisEmbedder", "_apply_mmr", context,
            ranked_results=semantic_hits or []
        )
```

```
        semantic_cube_stub = self._execute_method(
            "SemanticAnalyzer", "_empty_semantic_cube", context
        )
        domain_scores = self._execute_method(
            "SemanticAnalyzer", "_classify_policy_domain", context,
            segment=document_text
        )
        cross_cutting = self._execute_method(
            "SemanticAnalyzer", "_classify_cross_cutting_themes", context,
            segment=document_text
        )
        value_chain = self._execute_method(
            "SemanticAnalyzer", "_classify_value_chain_link", context,
            segment=document_text
        )
        semantic_vectors = self._execute_method(
            "SemanticAnalyzer", "_vectorize_segments", context,
            segments=[document_text]
        )
        processed_segment = self._execute_method(
            "SemanticAnalyzer", "_process_segment", context,
            segment=document_text,
            idx=0,
            vector=semantic_vectors[0] if semantic_vectors else None
        )
        semantic_complexity = self._execute_method(
            "SemanticAnalyzer", "_calculate_semantic_complexity", context,
            semantic_cube=semantic_cube_stub
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=[m.get("bpin","") for m in program_matches if isinstance(m, dict)],
            text_length=len(document_text),
            pattern_specificity=0.5
        )
        entity_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_entity_to_dict", context, entity=e)
            for e in consolidated_entities[:5]
            if isinstance(e, dict) or hasattr(e, "__dict__")
        ]

        # Step 3: Generate traceability records
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            matches=program_matches
        )

        # Step 4: Generate PDQ report
        pdq_report = self._execute_method(
            "PolicyAnalysisEmbedder", "generate_pdq_report", context,
            traceability=traceability_record
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
```

```
            entities=consolidated_entities
        )

        raw_evidence = {
            "budgetary_traceability": {
                "bpin_codes": [m.get("bpin") for m in program_matches if m.get("bpin")],
                "project_codes": [m.get("project_code") for m in program_matches if m.get("project_code")],
                "budget_matches": goal_budget_matches
            },
            "organizational_traceability": {
                "responsible_entities": consolidated_entities,
                "office_assignments": [e for e in consolidated_entities if e.get("office")],
                "secretariat_assignments": [e for e in consolidated_entities if e.get("secretariat")]
            },
            "traceability_record": traceability_record,
            "pdq_report": pdq_report,
            "accountability_matrix": accountability_matrix,
            "responsible_entities": responsible_entities,
            "responsibility_clarity_score": responsibility_clarity,
            "semantic_traceability": {
                "query": pdq_query,
                "semantic_hits": semantic_hits,
                "diversified_hits": diversified_hits
            },
            "semantic_cube_baseline": semantic_cube_stub,
            "policy_domain_scores": domain_scores,
            "responsibility_entities_dict": entity_dicts,
            "cross_cutting_themes": cross_cutting,
            "value_chain_links": value_chain,
            "semantic_vectors": semantic_vectors,
            "semantic_complexity": semantic_complexity,
            "evidence_confidence": evidence_confidence,
            "processed_segment": processed_segment
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "products_with_bpin": len([m for m in program_matches if m.get("bpin")]),
                "products_with_responsible": len(consolidated_entities),
                "canonical_question": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q4_TechnicalFeasibilityEvaluator(BaseExecutor):
    """
    DIM03_Q04_TECHNICAL_FEASIBILITY â\200\224 Evaluates activity-product feasibility vs resources/deadlines (canonical D3).
```

```
    Epistemic mix: structural DAG validity, causal necessity, performance/implementation readiness, and statistical robustness.

    Methods (from D3-Q4):
    - AdvancedDAGValidator._calculate_statistical_power
    - AdvancedDAGValidator._initialize_rng
    - AdvancedDAGValidator.export_nodes
    - AdvancedDAGValidator._generate_subgraph
    - AdvancedDAGValidator._get_node_validator
    - AdvancedDAGValidator._create_empty_result
    - HierarchicalGenerativeModel._calculate_likelihood
    - IndustrialGradeValidator.validate_causal_categories
    - TeoriaCambio._extraer_categorias
    - AdvancedDAGValidator.get_graph_stats
    - AdvancedDAGValidator._calculate_node_importance
    - AdvancedDAGValidator.calculate_acyclicity_pvalue
    - AdvancedDAGValidator._is_acyclic
    - BayesianMechanismInference._test_necessity
    - IndustrialGradeValidator.execute_suite
    - IndustrialGradeValidator.validate_connection_matrix
    - IndustrialGradeValidator.run_performance_benchmarks
    - IndustrialGradeValidator._benchmark_operation
    - PerformanceAnalyzer.analyze_performance
    - PerformanceAnalyzer._calculate_loss_functions
    - HierarchicalGenerativeModel._calculate_ess
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        plan_name = context.get("metadata", {}).get("title", "plan_desarrollo")

        # Step 1: Validate DAG structure
        acyclicity_pvalue = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )
        is_acyclic = self._execute_method(
            "AdvancedDAGValidator", "_is_acyclic", context
        )
        graph_stats = self._execute_method(
            "AdvancedDAGValidator", "get_graph_stats", context
        )
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
        )
        subgraph = self._execute_method(
            "AdvancedDAGValidator", "_generate_subgraph", context
        )
        added_node = self._execute_method(
            "AdvancedDAGValidator", "add_node", context,
            node_name="temp_node"
        )
        added_edge = self._execute_method(
            "AdvancedDAGValidator", "add_edge", context,
            source="temp_node",
            target="temp_target",
            weight=1.0
```

```
        )
        node_export = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context
        )
        rng_seed = self._execute_method(
            "AdvancedDAGValidator", "_initialize_rng", context,
            plan_name=plan_name,
            salt=dim_info.code
        )
        stat_power = self._execute_method(
            "AdvancedDAGValidator", "_calculate_statistical_power", context,
            s=int(graph_stats.get("edges", 0)),
            n=max(1, int(graph_stats.get("nodes", 1)))
        )
        node_validator = self._execute_method(
            "AdvancedDAGValidator", "_get_node_validator", context,
            node_type="producto"
        )
        empty_result = self._execute_method(
            "AdvancedDAGValidator", "_create_empty_result", context,
            plan_name=plan_name,
            seed=rng_seed,
            timestamp=context.get("metadata", {}).get("timestamp", "")
        )

        # Step 2: Test necessity of activities for products
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Execute industrial-grade validation
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )
        connection_validation = self._execute_method(
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )
        performance_benchmarks = self._execute_method(
            "IndustrialGradeValidator", "run_performance_benchmarks", context
        )
        benchmark_ops = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )
        metric_log = self._execute_method(
            "IndustrialGradeValidator", "_log_metric", context,
            name="custom_latency",
            value=graph_stats.get("edges", 0),
            unit="edges",
            threshold=10.0
        )
        engine_readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 4: Analyze performance
        performance_analysis = self._execute_method(
```

```
                "PerformanceAnalyzer", "analyze_performance", context
            )
            loss_functions = self._execute_method(
                "PerformanceAnalyzer", "_calculate_loss_functions", context
            )
            # Likelihood estimation for resource adequacy
            resource_likelihood = self._execute_method(
                "HierarchicalGenerativeModel", "_calculate_likelihood", context,
                mechanism_type="tecnico",
                observations={"coherence": performance_analysis.get("resource_fit", {}).get("score", 0.0)}
            )

            # Step 5: Calculate effective sample size
            ess = self._execute_method(
                "HierarchicalGenerativeModel", "_calculate_ess", context
            )
            r_hat = self._execute_method(
                "HierarchicalGenerativeModel", "_calculate_r_hat", context,
                chains=[]
            )
            causal_categories_valid = self._execute_method(
                "IndustrialGradeValidator", "validate_causal_categories", context
            )
            extracted_categories = self._execute_method(
                "TeoriaCambio", "_extraer_categorias", context,
                text=context.get("document_text", "")
            )

            raw_evidence = {
                "activity_product_mapping": connection_validation,
                "resource_adequacy": performance_analysis.get("resource_fit", {}),
                "timeline_feasibility": performance_analysis.get("timeline_feasibility", {}),
                "technical_validation": {
                    "dag_valid": is_acyclic,
                    "acyclicity_p": acyclicity_pvalue,
                    "necessity_score": necessity_test,
                    "graph_stats": graph_stats,
                    "node_importance": node_importance,
                    "subgraph_sample": subgraph,
                    "added_node": added_node,
                    "added_edge": added_edge,
                    "node_validator": node_validator,
                    "empty_result": empty_result,
                    "node_export": node_export,
                    "rng_seed": rng_seed,
                    "statistical_power": stat_power
                },
                "performance_metrics": {
                    "benchmarks": performance_benchmarks,
                    "loss_functions": loss_functions,
                    "ess": ess,
                    "r_hat": r_hat,
                    "resource_likelihood": resource_likelihood
                },
                "engine_readiness": engine_readiness,
                "feasibility_score": validation_suite.get("overall_score", 0),
```

```
            "causal_categories_valid": causal_categories_valid,
            "extracted_categories": extracted_categories,
            "metric_log": metric_log
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "dag_is_valid": is_acyclic,
                "feasibility_score": validation_suite.get("overall_score", 0),
                "canonical_question": "DIM03_Q04_TECHNICAL_FEASIBILITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q5_OutputOutcomeLinkageAnalyzer(BaseExecutor):
    """
    DIM03_Q05_OUTPUT_OUTCOME_LINKAGE â\200\224 Analyzes mechanisms linking outputs to outcomes with canonical D3 labeling.
    Epistemic mix: semantic hierarchy checks, causal order validation, DAG/effect estimation, and Bayesian mechanism inference.

Methods (from D3-Q5):
    - PDETMunicipalPlanAnalyzer._identify_confounders
    - PDETMunicipalPlanAnalyzer._effect_to_dict
    - PDETMunicipalPlanAnalyzer._scenario_to_dict
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - PDETMunicipalPlanAnalyzer._identify_causal_nodes
    - BayesianCounterfactualAuditor._evaluate_factual
    - BayesianCounterfactualAuditor._evaluate_counterfactual
    - CausalExtractor._assess_financial_consistency
    - BayesianMechanismInference._infer_activity_sequence
    - BayesianMechanismInference._generate_necessity_remediation
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - IndustrialPolicyProcessor._load_questionnaire
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - PDETMunicipalPlanAnalyzer.estimate_causal_effects
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - CausalExtractor._build_type_hierarchy
    - CausalExtractor._check_structural_violation
    - CausalExtractor._calculate_type_transition_prior
    - CausalExtractor._calculate_textual_proximity
    - TeoriaCambio._validar_orden_causal
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - PolicyAnalysisEmbedder.compare_policy_interventions
    - BayesianMechanismInference.infer_mechanisms
    """

```

```
    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Build causal backbone and effects
        financial_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        causal_dag = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
            financial_analysis=financial_analysis
        )
        causal_effects = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "estimate_causal_effects", context,
            dag=causal_dag,
            financial_analysis=financial_analysis
        )
        counterfactuals = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context,
            dag=causal_dag,
            causal_effects=causal_effects,
            financial_analysis=financial_analysis
        )
        simulated_intervention = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            intervention={},
            dag=causal_dag,
            causal_effects=causal_effects,
            label="baseline"
        )
        causal_nodes = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_nodes", context,
            text=context.get("document_text", ""),
            tables=context.get("tables", []),
            financial_analysis=financial_analysis
        )
        confounders = {}
        for effect in causal_effects:
            treatment = effect.treatment if hasattr(effect, "treatment") else None
            outcome = effect.outcome if hasattr(effect, "outcome") else None
            if treatment and outcome:
                confounders[(treatment, outcome)] = self._execute_method(
                    "PDETMunicipalPlanAnalyzer", "_identify_confounders", context,
                    treatment=treatment,
                    outcome=outcome,
                    dag=causal_dag
                )
        effect_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_effect_to_dict", context, effect=effect)
            for effect in causal_effects
        ]
        scenario_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_scenario_to_dict", context, scenario=scenario)
            for scenario in counterfactuals
        ]
        causal_recommendations = self._execute_method(
```

```
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={"financial_analysis": financial_analysis, "quality_score": getattr(causal_dag, 'graph', {})}
        )
        financial_consistency = None
        if refined_edges:
            first_edge = refined_edges[0] if isinstance(refined_edges, list) else {}
            source = first_edge.get("source") if isinstance(first_edge, dict) else ""
            target = first_edge.get("target") if isinstance(first_edge, dict) else ""
            financial_consistency = self._execute_method(
                "CausalExtractor", "_assess_financial_consistency", context,
                source=source or "",
                target=target or ""
            )
        factual_eval = None
        counterfactual_eval = None
        if causal_effects:
            first_effect = causal_effects[0]
            target = getattr(first_effect, "outcome", None) or ""
            evidence = {"p_effect": getattr(first_effect, "probability_significant", 0.0)}
            factual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_factual", context,
                target=target,
                evidence=evidence
            )
            counterfactual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_counterfactual", context,
                target=target,
                intervention={"shift": 0.1}
            )
        matched_node = None
        try:
            matched_node = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_match_text_to_node", context,
                text=context.get("document_text", "")[:200],
                nodes=causal_nodes if isinstance(causal_nodes, dict) else {}
            )
        except Exception:
            matched_node = None

        # Step 1: Build type hierarchy
        type_hierarchy = self._execute_method(
            "CausalExtractor", "_build_type_hierarchy", context
        )

        # Step 2: Check structural violations
        structural_violations = self._execute_method(
            "CausalExtractor", "_check_structural_violation", context,
            hierarchy=type_hierarchy
        )

        # Step 3: Calculate transition priors and proximity
        transition_priors = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            hierarchy=type_hierarchy
        )
        textual_proximity = self._execute_method(
```

```
            "CausalExtractor", "_calculate_textual_proximity", context
        )

        # Step 4: Validate causal order
        causal_order_validation = self._execute_method(
            "TeoriaCambio", "_validar_orden_causal", context,
            hierarchy=type_hierarchy
        )

        # Step 5: Refine edge probabilities
        refined_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            priors=transition_priors
        )

        # Step 6: Compare policy interventions
        intervention_comparison = self._execute_method(
            "PolicyAnalysisEmbedder", "compare_policy_interventions", context
        )

        # Step 7: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context,
            edges=refined_edges
        )
        mechanism_sample = next(iter(mechanisms.values()), {})
        activity_sequence = self._execute_method(
            "BayesianMechanismInference", "_infer_activity_sequence", context,
            observations=mechanism_sample.get("observations", {}),
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0})
        )
        quantified_uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0}),
            sequence_posterior=mechanism_sample.get("activity_sequence", {}),
            coherence_score=mechanism_sample.get("coherence_score", 0.0)
        )
        mechanism_observations = self._execute_method(
            "BayesianMechanismInference", "_extract_observations", context,
            node={"id": next(iter(mechanisms.keys()), "")},
            text=context.get("document_text", "")
        )
        necessity_remediation = self._execute_method(
            "BayesianMechanismInference", "_generate_necessity_remediation", context,
            node_id=next(iter(mechanisms.keys()), ""),
            missing_components=structural_violations
        )
        questionnaire_stub = self._execute_method(
            "IndustrialPolicyProcessor", "_load_questionnaire", context
        )
        refutation_checks = None
        try:
            confounder_keys = list(confounders.keys())
            first_pair = confounder_keys[0] if confounder_keys else ("", "")
            refutation_checks = self._execute_method(
                "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
```

```
                dag=getattr(causal_dag, "graph", None),
                target=first_pair[1],
                treatment=first_pair[0],
                confounders=list(confounders.values())[0] if confounders else []
            )
        except Exception:
            refutation_checks = None

        raw_evidence = {
            "output_outcome_links": refined_edges,
            "mechanism_explanation": mechanisms,
            "type_hierarchy": type_hierarchy,
            "causal_dag": causal_dag,
            "causal_effects": causal_effects,
            "counterfactuals": counterfactuals,
            "simulated_intervention": simulated_intervention,
            "causal_nodes": causal_nodes,
            "financial_analysis": financial_analysis,
            "causal_validity": {
                "structural_violations": structural_violations,
                "order_valid": causal_order_validation
            },
            "transition_probabilities": transition_priors,
            "textual_proximity": textual_proximity,
            "intervention_comparison": intervention_comparison,
            "confounders": confounders,
            "effect_dicts": effect_dicts,
            "scenario_dicts": scenario_dicts,
            "activity_sequence_sample": activity_sequence,
            "uncertainty_quantified": quantified_uncertainty,
            "mechanism_observations": mechanism_observations,
            "refutation_checks": refutation_checks,
            "necessity_remediation": necessity_remediation,
            "questionnaire_stub": questionnaire_stub,
            "causal_recommendations": causal_recommendations,
            "financial_consistency": financial_consistency,
            "factual_eval": factual_eval,
            "counterfactual_eval": counterfactual_eval,
            "matched_node": matched_node
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(mechanisms),
                "violations_found": len(structural_violations),
                "canonical_question": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
```

```
        }


# ============================================================================
# DIMENSION 4: RESULTS & OUTCOMES
# ============================================================================

class D4_Q1_OutcomeMetricsValidator(BaseExecutor):
    """
    DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS â\200\224 Validates outcome indicators (baseline, target, horizon) with canonical D4 notation.
    Epistemic mix: semantic goal extraction, temporal/consistency checks, statistical performance signals, and indicator quality scoring.

    Methods (from D4-Q1):
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - CausalExtractor._calculate_language_specificity
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._calculate_semantic_distance
    - TemporalLogicVerifier._classify_temporal_type
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._find_outcome_mentions
    - PDETMunicipalPlanAnalyzer._score_temporal_consistency
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
    - CausalExtractor._classify_goal_type
    - TemporalLogicVerifier._parse_temporal_marker
    - TemporalLogicVerifier._extract_resources
    - PerformanceAnalyzer.analyze_performance
    - PerformanceAnalyzer._generate_recommendations
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D4.value)

        # Step 1: Find outcome mentions
        outcome_mentions = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_outcome_mentions", context
        )
        entities_syntax = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context,
            text=context.get("document_text", "")
        )
        entities_ner = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context,
            text=context.get("document_text", "")
        )

        # Step 2: Score temporal consistency
        temporal_consistency = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_temporal_consistency", context,
            outcomes=outcome_mentions
        )

        # Step 3: Extract and classify goals
        goals = self._execute_method(
```

```
                "CausalExtractor", "_extract_goals", context
            )
            goal_contexts = self._execute_method(
                "CausalExtractor", "_parse_goal_context", context,
                goals=goals
            )
            goal_types = self._execute_method(
                "CausalExtractor", "_classify_goal_type", context,
                goals=goals
            )
            semantic_distance = 0.0
            if goal_types and outcome_mentions:
                semantic_distance = self._execute_method(
                    "CausalExtractor", "_calculate_semantic_distance", context,
                    source=str(goal_types[0]),
                    target=str(outcome_mentions[0])
                )

            # Step 4: Parse temporal markers
            temporal_markers = self._execute_method(
                "TemporalLogicVerifier", "_parse_temporal_marker", context,
                contexts=goal_contexts
            )
            temporal_type = self._execute_method(
                "TemporalLogicVerifier", "_classify_temporal_type", context,
                marker=temporal_markers[0] if temporal_markers else ""
            )
            resources_mentioned = self._execute_method(
                "TemporalLogicVerifier", "_extract_resources", context,
                text=context.get("document_text", "")
            )
            precedence_check = self._execute_method(
                "TemporalLogicVerifier", "_should_precede", context,
                marker_a=temporal_markers[0] if temporal_markers else "",
                marker_b=temporal_markers[1] if len(temporal_markers) > 1 else ""
            )

            # Step 5: Analyze performance
            performance_analysis = self._execute_method(
                "PerformanceAnalyzer", "analyze_performance", context,
                outcomes=outcome_mentions
            )
            indicator_quality = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_score_indicators", context
            )
            performance_recommendations = self._execute_method(
                "PerformanceAnalyzer", "_generate_recommendations", context,
                performance_analysis=performance_analysis
            )
            # Semantic certainty for goals
            language_specificity = self._execute_method(
                "CausalExtractor", "_calculate_language_specificity", context,
                keyword=goal_contexts[0] if goal_contexts else "",
                policy_area=context.get("policy_area")
            )
            composite_likelihood = self._execute_method(
```

```
            "CausalExtractor", "_calculate_composite_likelihood", context,
            evidence={
                "semantic_distance": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0,
                "textual_proximity": performance_analysis.get("coherence_score", 0.0) if isinstance(performance_analysis, dict) else 0.0,
                "language_specificity": language_specificity,
                "temporal_coherence": temporal_consistency if isinstance(temporal_consistency, (int, float)) else 0.0
            }
        )

        raw_evidence = {
            "outcome_indicators": outcome_mentions,
            "indicators_with_baseline": [o for o in outcome_mentions if o.get("has_baseline")],
            "indicators_with_target": [o for o in outcome_mentions if o.get("has_target")],
            "indicators_with_horizon": [o for o in outcome_mentions if o.get("time_horizon")],
            "temporal_consistency_score": temporal_consistency,
            "goal_classifications": goal_types,
            "temporal_markers": temporal_markers,
            "performance_metrics": performance_analysis,
            "indicator_quality": indicator_quality,
            "performance_recommendations": performance_recommendations,
            "entities_syntax": entities_syntax,
            "entities_ner": entities_ner,
            "temporal_type": temporal_type,
            "language_specificity": language_specificity,
            "composite_likelihood": composite_likelihood,
            "goal_outcome_semantic_distance": semantic_distance,
            "resources_mentioned": resources_mentioned,
            "precedence_check": precedence_check
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_outcomes": len(outcome_mentions),
                "complete_indicators": len([o for o in outcome_mentions
                    if o.get("has_baseline") and o.get("has_target") and o.get("time_horizon")]),
                "canonical_question": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q2_CausalChainValidator(BaseExecutor):
    """
    Validates explicit causal chain with assumptions and enabling conditions.

    Methods (from D4-Q2):
    - TeoriaCambio._encontrar_caminos_completos
    - TeoriaCambio.validacion_completa
```

```
    - CausalExtractor.extract_causal_hierarchy
    - HierarchicalGenerativeModel.verify_conditional_independence
    - HierarchicalGenerativeModel._generate_independence_tests
    - BayesianCounterfactualAuditor.construct_scm
    - AdvancedDAGValidator._perform_sensitivity_analysis_internal
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Find complete causal paths
        complete_paths = self._execute_method(
            "TeoriaCambio", "_encontrar_caminos_completos", context
        )

        # Step 2: Complete validation
        validation_results = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            paths=complete_paths
        )

        # Step 3: Extract causal hierarchy
        causal_hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        # Step 4: Verify conditional independence
        independence_verification = self._execute_method(
            "HierarchicalGenerativeModel", "verify_conditional_independence", context,
            hierarchy=causal_hierarchy
        )
        independence_tests = self._execute_method(
            "HierarchicalGenerativeModel", "_generate_independence_tests", context,
            verification=independence_verification
        )

        # Step 5: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            hierarchy=causal_hierarchy
        )

        # Step 6: Perform sensitivity analysis
        sensitivity_analysis = self._execute_method(
            "AdvancedDAGValidator", "_perform_sensitivity_analysis_internal", context,
            scm=scm
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            analysis=sensitivity_analysis
        )

        raw_evidence = {
```

```
            "causal_chain": complete_paths,
            "key_assumptions": validation_results.get("assumptions", []),
            "enabling_conditions": validation_results.get("conditions", []),
            "external_factors": validation_results.get("external_factors", []),
            "causal_hierarchy": causal_hierarchy,
            "independence_tests": independence_tests,
            "structural_model": scm,
            "sensitivity": sensitivity_analysis,
            "evidential_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "complete_paths_found": len(complete_paths),
                "assumptions_identified": len(validation_results.get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q3_AmbitionJustificationAnalyzer(BaseExecutor):
    """
    Analyzes justification of result ambition based on investment/capacity/benchmarks.

    Methods (from D4-Q3):
    - PDETMunicipalPlanAnalyzer._get_prior_effect
    - PDETMunicipalPlanAnalyzer._estimate_effect_bayesian
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - AdaptivePriorCalculator.sensitivity_analysis
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator._calculate_statistical_power
    - BayesianMechanismInference._aggregate_bayesian_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Get prior effect estimates
        prior_effects = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_prior_effect", context
        )

        # Step 2: Estimate effect using Bayesian methods
        effect_estimate = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_estimate_effect_bayesian", context,
            priors=prior_effects
        )

        # Step 3: Compute robustness
```

```
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            estimate=effect_estimate
        )

        # Step 4: Sensitivity analysis
        sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            estimate=effect_estimate
        )

        # Step 5: Calculate convergence diagnostics
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context
        )
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )

        # Step 6: Calculate statistical power
        statistical_power = self._execute_method(
            "AdvancedDAGValidator", "_calculate_statistical_power", context,
            effect=effect_estimate
        )

        # Step 7: Aggregate confidence
        confidence_aggregate = self._execute_method(
            "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
            estimates=[effect_estimate, robustness, statistical_power]
        )

        raw_evidence = {
            "ambition_level": context.get("target_ambition", {}),
            "financial_investment": context.get("total_investment", 0),
            "institutional_capacity": context.get("capacity_score", 0),
            "comparative_benchmarks": prior_effects,
            "justification_analysis": {
                "effect_estimate": effect_estimate,
                "robustness": robustness,
                "sensitivity": sensitivity,
                "statistical_power": statistical_power
            },
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess
            },
            "overall_confidence": confidence_aggregate
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "ambition_justified": confidence_aggregate > 0.7,
                "statistical_power": statistical_power
```

```
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q4_ProblemSolvencyEvaluator(BaseExecutor):
    """
    Evaluates whether results address/resolve prioritized problems from diagnosis.

    Methods (from D4-Q4):
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._identify_affected_sections
    - PolicyContradictionDetector._generate_resolution_recommendations
    - OperationalizationAuditor._generate_optimal_remediations
    - OperationalizationAuditor._get_remediation_text
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - FinancialAuditor._detect_allocation_gaps
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate objective alignment
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )

        # Step 2: Identify affected sections
        affected_sections = self._execute_method(
            "PolicyContradictionDetector", "_identify_affected_sections", context,
            alignment=objective_alignment
        )

        # Step 3: Generate resolution recommendations
        resolutions = self._execute_method(
            "PolicyContradictionDetector", "_generate_resolution_recommendations", context,
            sections=affected_sections
        )

        # Step 4: Generate optimal remediations
        remediations = self._execute_method(
            "OperationalizationAuditor", "_generate_optimal_remediations", context
        )
        remediation_text = self._execute_method(
            "OperationalizationAuditor", "_get_remediation_text", context,
            remediations=remediations
        )

        # Step 5: Aggregate risk and prioritize
        risk_priorities = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context
        )

```

```
        # Step 6: Detect allocation gaps
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )

        raw_evidence = {
            "prioritized_problems": context.get("diagnosis_problems", []),
            "proposed_results": context.get("outcome_indicators", []),
            "problem_result_mapping": objective_alignment,
            "unaddressed_problems": [p for p in affected_sections if not p.get("addressed")],
            "solvency_score": objective_alignment.get("score", 0),
            "resolution_recommendations": resolutions,
            "remediations": remediation_text,
            "risk_priorities": risk_priorities,
            "allocation_gaps": allocation_gaps
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "problems_addressed": len([p for p in affected_sections if p.get("addressed")]),
                "problems_unaddressed": len([p for p in affected_sections if not p.get("addressed")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q5_VerticalAlignmentValidator(BaseExecutor):
    """
    Validates alignment with superior frameworks (PND, SDGs).

    Methods (from D4-Q5):
    - PDETMunicipalPlanAnalyzer._score_pdet_alignment
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - CDAFFramework._validate_dnp_compliance
    - CDAFFramework._generate_dnp_report
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - AdaptivePriorCalculator.validate_quality_criteria
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score PDET alignment
        pdet_alignment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_pdet_alignment", context
        )

        # Step 2: Score causal coherence
        causal_coherence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
```

```
        )

        # Step 3: Validate DNP compliance
        dnp_compliance = self._execute_method(
            "CDAFFramework", "_validate_dnp_compliance", context
        )
        dnp_report = self._execute_method(
            "CDAFFramework", "_generate_dnp_report", context,
            compliance=dnp_compliance
        )

        # Step 4: Analyze causal dimensions
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )

        # Step 5: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context,
            alignment=pdet_alignment
        )

        raw_evidence = {
            "pnd_alignment": dnp_compliance,
            "sdg_alignment": context.get("sdg_mappings", []),
            "pdet_alignment": pdet_alignment,
            "alignment_declarations": dnp_report.get("declarations", []),
            "causal_coherence": causal_coherence,
            "causal_dimensions": causal_dimensions,
            "quality_validation": quality_validation,
            "alignment_score": (pdet_alignment.get("score", 0) +
                                dnp_compliance.get("score", 0)) / 2
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "pnd_aligned": dnp_compliance.get("is_compliant", False),
                "sdgs_referenced": len(context.get("sdg_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 5: IMPACTS
# ============================================================================

class D5_Q1_LongTermVisionAnalyzer(BaseExecutor):
    """
    Analyzes long-term impacts, transmission routes, and time lags.
```

```

    Methods (from D5-Q1):
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_scenario_narrative
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - TeoriaCambio._validar_orden_causal
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._generate_interventions
    - BayesianCounterfactualAuditor.construct_scm
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Generate counterfactuals
        counterfactuals = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context
        )

        # Step 2: Simulate interventions
        simulation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            counterfactuals=counterfactuals
        )

        # Step 3: Generate scenario narratives
        scenarios = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_scenario_narrative", context,
            simulation=simulation
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Validate causal order
        causal_order = self._execute_method(
            "TeoriaCambio", "_validar_orden_causal", context,
            mediators=mediators
        )

        # Step 6: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 7: Generate interventions
        interventions = self._execute_method(
            "TextMiningEngine", "_generate_interventions", context
        )

        # Step 8: Construct SCM
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
```

```
            order=causal_order
        )

        raw_evidence = {
            "long_term_impacts": context.get("impact_indicators", []),
            "structural_transformations": scenarios,
            "transmission_routes": mediators,
            "expected_time_lags": temporal_coherence.get("time_lags", []),
            "counterfactual_analysis": counterfactuals,
            "simulation_results": simulation,
            "causal_pathways": scm,
            "intervention_scenarios": interventions
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "impacts_defined": len(context.get("impact_indicators", [])),
                "mediators_identified": len(mediators)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q2_CompositeMeasurementValidator(BaseExecutor):
    """
    DIM05_Q02_COMPOSITE_PROXY_VALIDITY â\200\224 Validates composite indices/proxies for complex impacts (canonical D5).
    Epistemic mix: statistical robustness (E-value), Bayesian confidence, normative reporting quality, and semantic consistency.

    Methods (from D5-Q2):
    - PDETMunicipalPlanAnalyzer._quality_to_dict
    - PolicyAnalysisEmbedder.process_document
    - PolicyAnalysisEmbedder._filter_by_pdq
    - PolicyAnalysisEmbedder._extract_numerical_values
    - PolicyAnalysisEmbedder._compute_overall_confidence
    - PolicyAnalysisEmbedder._embed_texts
    - PolicyTextProcessor.normalize_unicode
    - PolicyTextProcessor.segment_into_sentences
    - PolicyTextProcessor.compile_pattern
    - PolicyTextProcessor.extract_contextual_window
    - IndustrialPolicyProcessor._compute_evidence_confidence
    - IndustrialPolicyProcessor._compute_avg_confidence
    - IndustrialPolicyProcessor._construct_evidence_bundle
    - PDETMunicipalPlanAnalyzer.generate_executive_report
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - PDETMunicipalPlanAnalyzer._interpret_overall_quality
    - PolicyAnalysisEmbedder.get_diagnostics
    - PDETMunicipalPlanAnalyzer.calculate_quality_score
    - PDETMunicipalPlanAnalyzer._estimate_score_confidence
    - PDETMunicipalPlanAnalyzer._compute_e_value
```

```
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - ReportingEngine._calculate_quality_score
    - BayesianMechanismInference._aggregate_bayesian_confidence
    - PolicyAnalysisEmbedder.evaluate_policy_numerical_consistency
    - FinancialAuditor._calculate_sufficiency
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D5.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Calculate quality scores
        quality_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "calculate_quality_score", context
        )
        score_confidence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_estimate_score_confidence", context,
            score=quality_score
        )

        # Step 2: Compute robustness metrics
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            score=quality_score
        )
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            score=quality_score
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            e_value=e_value,
            robustness=robustness
        )

        # Step 3: Calculate reporting quality score
        reporting_quality = self._execute_method(
            "ReportingEngine", "_calculate_quality_score", context
        )

        # Step 4: Aggregate Bayesian confidence
        bayesian_confidence = self._execute_method(
            "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
            scores=[quality_score, reporting_quality]
        )

        # Step 5: Evaluate numerical consistency
        numerical_consistency = self._execute_method(
            "PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency", context
        )
        embedder_diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context
        )
        processed_chunks = self._execute_method(
```

```
            "PolicyAnalysisEmbedder", "process_document", context,
            document_text=document_text,
            document_metadata=document_metadata
        )
        pdq_filter = self._execute_method(
            "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
            pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
        )
        filtered_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "_filter_by_pdq", context,
            chunks=processed_chunks,
            pdq_filter=pdq_filter
        )
        numerical_values = self._execute_method(
            "PolicyAnalysisEmbedder", "_extract_numerical_values", context,
            chunks=processed_chunks
        )
        embedded_texts = self._execute_method(
            "PolicyAnalysisEmbedder", "_embed_texts", context,
            texts=[c.get("content", "") for c in processed_chunks] if isinstance(processed_chunks, list) else []
        )
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            relevant_chunks=filtered_chunks[:5] if isinstance(filtered_chunks, list) else [],
            numerical_eval=bayesian_confidence if isinstance(bayesian_confidence, dict) else {"evidence_strength": "weak", "numerical_coherence": 0.0}
        )

        # Step 6: Calculate sufficiency
        sufficiency = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )
        overall_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_overall_quality", context,
            score=getattr(quality_score, "overall_score", quality_score)
        )
        risk_prioritization = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            omission_score=1 - quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.2,
            insufficiency_score=1 - sufficiency.get("coverage_ratio", 0.0),
            unnecessity_score=1 - (robustness if isinstance(robustness, (int, float)) else 0.0),
            causal_effect=e_value,
            feasibility=quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.8,
            cost=1.0
        )
        normalized_text = self._execute_method(
            "PolicyTextProcessor", "normalize_unicode", context,
            text=document_text
        )
        segmented_sentences = self._execute_method(
            "PolicyTextProcessor", "segment_into_sentences", context,
            text=document_text
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=context.get("proxy_indicators", []),
            text_length=len(document_text),
```

```
            pattern_specificity=0.5
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D5": {"dimension_confidence": bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0}}
        )
        quality_dict = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_quality_to_dict", context,
            quality=quality_score
        )
        evidence_bundle = self._execute_method(
            "IndustrialPolicyProcessor", "_construct_evidence_bundle", context,
            dimension=None,
            category="composite",
            matches=context.get("proxy_indicators", []),
            positions=[],
            confidence=bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0
        )
        compiled_pattern = self._execute_method(
            "PolicyTextProcessor", "compile_pattern", context,
            pattern_str=r"[A-Z]{2,}\\s+\\d+"
        )
        contextual_window = self._execute_method(
            "PolicyTextProcessor", "extract_contextual_window", context,
            text=document_text,
            match_position=0,
            window_size=200
        )
        exec_report = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_executive_report", context,
            analysis_results={"quality_score": quality_dict, "financial_analysis": context.get("financial_analysis", {}) or {"total_budget": 0, "funding_sources": {}, "confidence": (0, 0)}}
        )
        export_result = self._execute_method(
            "IndustrialPolicyProcessor", "export_results", context,
            results={"quality": quality_dict, "robustness": robustness},
            output_path="output/composite_results.json"
        )

        raw_evidence = {
            "composite_indices": context.get("composite_indicators", []),
            "proxy_indicators": context.get("proxy_indicators", []),
            "validity_justification": score_confidence,
            "robustness_metrics": {
                "e_value": e_value,
                "robustness": robustness,
                "interpretation": sensitivity_interpretation
            },
            "quality_scores": {
                "overall": quality_score,
                "reporting": reporting_quality
            },
            "bayesian_confidence": bayesian_confidence,
            "numerical_consistency": numerical_consistency,
            "measurement_sufficiency": sufficiency,
```

```
            "embedder_diagnostics": embedder_diagnostics,
            "quality_interpretation": overall_interpretation,
            "pdq_filter": pdq_filter,
            "filtered_chunks": filtered_chunks,
            "numerical_values": numerical_values,
            "embedded_texts": embedded_texts,
            "overall_confidence": overall_confidence,
            "risk_prioritization": risk_prioritization,
            "normalized_text": normalized_text,
            "segmented_sentences": segmented_sentences,
            "evidence_confidence": evidence_confidence,
            "avg_confidence": avg_confidence,
            "quality_dict": quality_dict,
            "compiled_pattern": compiled_pattern,
            "contextual_window": contextual_window,
            "evidence_bundle": evidence_bundle,
            "executive_report": exec_report,
            "export_result": export_result
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "composite_indices_count": len(context.get("composite_indicators", [])),
                "validity_score": score_confidence,
                "canonical_question": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q3_IntangibleMeasurementAnalyzer(BaseExecutor):
    """
    Analyzes proxy indicators for intangible impacts with validity documentation.

    Methods (from D5-Q3):
    - CausalExtractor._calculate_semantic_distance
    - SemanticAnalyzer.extract_semantic_cube
    - BayesianMechanismInference._quantify_uncertainty
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - PolicyAnalysisEmbedder.get_diagnostics
    - AdaptivePriorCalculator._perturb_evidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate semantic distance
        semantic_distance = self._execute_method(
```

```
            "CausalExtractor", "_calculate_semantic_distance", context
        )

        # Step 2: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        # Step 3: Quantify uncertainty
        uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            semantic_data=semantic_cube
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Get diagnostics
        diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context,
            mediators=mediators
        )

        # Step 6: Perturb evidence for sensitivity
        perturbed_evidence = self._execute_method(
            "AdaptivePriorCalculator", "_perturb_evidence", context,
            diagnostics=diagnostics
        )

        raw_evidence = {
            "intangible_impacts": context.get("intangible_indicators", []),
            "proxy_indicators": context.get("proxy_mappings", []),
            "validity_documentation": diagnostics,
            "limitations_acknowledged": diagnostics.get("limitations", []),
            "semantic_relationships": semantic_cube,
            "semantic_distance": semantic_distance,
            "uncertainty_quantification": uncertainty,
            "sensitivity_to_proxies": perturbed_evidence
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "intangibles_count": len(context.get("intangible_indicators", [])),
                "proxies_defined": len(context.get("proxy_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }

```

```

class D5_Q4_SystemicRiskEvaluator(BaseExecutor):
    """
    Evaluates systemic risks that could rupture causal mechanisms.

    Methods (from D5-Q4):
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - BayesianCounterfactualAuditor._test_effect_stability
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - PDETMunicipalPlanAnalyzer._break_cycles
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Audit systemic risks
        systemic_risks = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 2: Refutation and sanity checks
        refutation = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            risks=systemic_risks
        )

        # Step 3: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            refutation=refutation
        )

        # Step 4: Interpret risks
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risks=systemic_risks
        )

        # Step 5: Interpret sensitivity
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            stability=effect_stability
        )

        # Step 6: Break cycles if present
        cycle_breaks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_break_cycles", context
        )

        # Step 7: Sensitivity analysis
        sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=systemic_risks
```

```
        )

        raw_evidence = {
            "macroeconomic_risks": [r for r in systemic_risks if r.get("type") == "macroeconomic"],
            "environmental_risks": [r for r in systemic_risks if r.get("type") == "environmental"],
            "political_risks": [r for r in systemic_risks if r.get("type") == "political"],
            "mechanism_rupture_potential": risk_interpretation.get("rupture_probability", 0),
            "effect_stability": effect_stability,
            "refutation_results": refutation,
            "sensitivity_analysis": sensitivity,
            "cycle_vulnerabilities": cycle_breaks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "systemic_risks_identified": len(systemic_risks),
                "high_risk_count": len([r for r in systemic_risks if r.get("severity") == "high"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q5_RealismAndSideEffectsAnalyzer(BaseExecutor):
    """
    Analyzes realism of impact ambition and potential unintended effects.

    Methods (from D5-Q5):
    - HierarchicalGenerativeModel.posterior_predictive_check
    - HierarchicalGenerativeModel._ablation_analysis
    - HierarchicalGenerativeModel._calculate_waic_difference
    - AdaptivePriorCalculator._add_ood_noise
    - AdaptivePriorCalculator.validate_quality_criteria
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - BayesianMechanismInference._calculate_coherence_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Posterior predictive check
        predictive_check = self._execute_method(
            "HierarchicalGenerativeModel", "posterior_predictive_check", context
        )

        # Step 2: Ablation analysis
        ablation = self._execute_method(
            "HierarchicalGenerativeModel", "_ablation_analysis", context,
            check=predictive_check
        )
```

```
        # Step 3: Calculate WAIC difference
        waic_diff = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_waic_difference", context,
            ablation=ablation
        )

        # Step 4: Add out-of-distribution noise
        ood_analysis = self._execute_method(
            "AdaptivePriorCalculator", "_add_ood_noise", context
        )

        # Step 5: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context,
            ood=ood_analysis
        )

        # Step 6: Compute robustness metrics
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context
        )
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context
        )

        # Step 7: Calculate coherence factor
        coherence = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context
        )

        raw_evidence = {
            "impact_ambition_level": context.get("declared_ambition", 0),
            "realism_assessment": predictive_check.get("realism_score", 0),
            "negative_side_effects": ablation.get("negative_effects", []),
            "limit_hypotheses": quality_validation.get("limits", []),
            "robustness_metrics": {
                "e_value": e_value,
                "robustness": robustness,
                "coherence": coherence
            },
            "predictive_validity": predictive_check,
            "ablation_results": ablation,
            "model_comparison": waic_diff,
            "ood_sensitivity": ood_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "realism_score": predictive_check.get("realism_score", 0),
                "side_effects_identified": len(ablation.get("negative_effects", []))
            },
            "execution_metrics": {
```

```
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 6: CAUSALITY & THEORY OF CHANGE
# ============================================================================

class D6_Q1_ExplicitTheoryBuilder(BaseExecutor):
    """
    Builds/validates explicit Theory of Change with diagram and assumptions.

    Methods (from D6-Q1):
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio.validacion_completa
    - TeoriaCambio.export_nodes
    - ReportingEngine.generate_causal_diagram
    - ReportingEngine.generate_causal_model_json
    - AdvancedDAGValidator.export_nodes
    - PDETMunicipalPlanAnalyzer.export_causal_network
    - CausalExtractor.extract_causal_hierarchy
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Build causal graph
        causal_graph = self._execute_method(
            "TeoriaCambio", "construir_grafo_causal", context
        )

        # Step 2: Complete validation
        validation = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            graph=causal_graph
        )

        # Step 3: Export nodes from Theory of Change
        toc_nodes = self._execute_method(
            "TeoriaCambio", "export_nodes", context,
            graph=causal_graph
        )

        # Step 4: Generate causal diagram
        diagram = self._execute_method(
            "ReportingEngine", "generate_causal_diagram", context,
            graph=causal_graph
        )

        # Step 5: Generate causal model JSON
        model_json = self._execute_method(
            "ReportingEngine", "generate_causal_model_json", context,
            graph=causal_graph
        )
```

```

        # Step 6: Export nodes from DAG validator
        dag_nodes = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context,
            graph=causal_graph
        )

        # Step 7: Export causal network
        network_export = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "export_causal_network", context,
            graph=causal_graph
        )

        # Step 8: Extract causal hierarchy
        hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        raw_evidence = {
            "toc_exists": len(causal_graph) > 0,
            "toc_diagram": diagram,
            "toc_json": model_json,
            "causal_graph": causal_graph,
            "nodes": toc_nodes,
            "causes_identified": hierarchy.get("causes", []),
            "mediators_identified": hierarchy.get("mediators", []),
            "assumptions": validation.get("assumptions", []),
            "network_structure": network_export,
            "validation_results": validation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "nodes_count": len(toc_nodes),
                "assumptions_count": len(validation.get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q2_LogicalProportionalityValidator(BaseExecutor):
    """
    Validates logical proportionality: no leaps, intervention matches result scale.

    Methods (from D6-Q2):
    - BeachEvidentialTest.apply_test_logic
    - BayesianMechanismInference._test_necessity
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._calculate_coherence_factor
    - BayesianCounterfactualAuditor._test_effect_stability
```

```
    - IndustrialGradeValidator.validate_connection_matrix
    - PolicyAnalysisEmbedder._compute_overall_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context
        )

        # Step 2: Test necessity
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Test sufficiency
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )

        # Step 4: Calculate coherence factor
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            necessity=necessity_test,
            sufficiency=sufficiency_test
        )

        # Step 5: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context
        )

        # Step 6: Validate connection matrix
        connection_validation = self._execute_method(
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )

        # Step 7: Compute overall confidence
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            tests=[necessity_test, sufficiency_test, effect_stability]
        )

        raw_evidence = {
            "logical_leaps_detected": evidential_tests.get("leaps", []),
            "intervention_scale": context.get("intervention_magnitude", 0),
            "result_scale": context.get("result_magnitude", 0),
            "proportionality_ratio": context.get("intervention_magnitude", 0) / max(context.get("result_magnitude", 1), 1),
            "necessity_score": necessity_test,
            "sufficiency_score": sufficiency_test,
            "coherence_factor": coherence_factor,
            "effect_stability": effect_stability,
            "connection_validation": connection_validation,
            "overall_confidence": overall_confidence,
```

```
            "implementation_miracles": [leap for leap in evidential_tests.get("leaps", [])
                                        if leap.get("type") == "miracle"]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "leaps_detected": len(evidential_tests.get("leaps", [])),
                "proportionality_adequate": abs(raw_evidence["proportionality_ratio"] - 1.0) < 0.5
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q3_ValidationTestingAnalyzer(BaseExecutor):
    """
    Analyzes validation/testing proposals for weak assumptions before scaling.

    Methods (from D6-Q3):
    - IndustrialGradeValidator.execute_suite
    - IndustrialGradeValidator.validate_engine_readiness
    - IndustrialGradeValidator._benchmark_operation
    - AdaptivePriorCalculator.validate_quality_criteria
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator.calculate_acyclicity_pvalue
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Execute validation suite
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )

        # Step 2: Validate engine readiness
        readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 3: Benchmark operations
        benchmarks = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )

        # Step 4: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context
        )
```

```

        # Step 5: Calculate convergence diagnostics
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context
        )
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )

        # Step 6: Calculate acyclicity p-value
        acyclicity_p = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )

        # Step 7: Analyze performance
        performance = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "inconsistencies_recognized": validation_suite.get("inconsistencies", []),
            "weak_assumptions": quality_validation.get("weak_assumptions", []),
            "pilot_proposals": context.get("pilot_programs", []),
            "testing_proposals": context.get("testing_plans", []),
            "validation_before_scaling": readiness.get("ready_to_scale", False),
            "validation_results": validation_suite,
            "quality_criteria": quality_validation,
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess,
                "acyclicity_p": acyclicity_p
            },
            "performance_analysis": performance,
            "benchmarks": benchmarks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "inconsistencies_count": len(validation_suite.get("inconsistencies", [])),
                "pilots_proposed": len(context.get("pilot_programs", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q4_FeedbackLoopAnalyzer(BaseExecutor):
    """
    Analyzes monitoring system with correction mechanisms and learning processes.

    Methods (from D6-Q4):
```

```
    - ConfigLoader.update_priors_from_feedback
    - ConfigLoader.check_uncertainty_reduction_criterion
    - ConfigLoader._save_prior_history
    - ConfigLoader._load_uncertainty_history
    - CDAFFramework._extract_feedback_from_audit
    - AdvancedDAGValidator._calculate_node_importance
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Update priors from feedback
        prior_updates = self._execute_method(
            "ConfigLoader", "update_priors_from_feedback", context
        )

        # Step 2: Check uncertainty reduction
        uncertainty_reduction = self._execute_method(
            "ConfigLoader", "check_uncertainty_reduction_criterion", context,
            updates=prior_updates
        )

        # Step 3: Save prior history
        history_saved = self._execute_method(
            "ConfigLoader", "_save_prior_history", context,
            updates=prior_updates
        )

        # Step 4: Load uncertainty history
        uncertainty_history = self._execute_method(
            "ConfigLoader", "_load_uncertainty_history", context
        )

        # Step 5: Extract feedback from audit
        feedback_extracted = self._execute_method(
            "CDAFFramework", "_extract_feedback_from_audit", context
        )

        # Step 6: Calculate node importance
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            updates=prior_updates
        )

        raw_evidence = {
            "monitoring_system_described": len(context.get("monitoring_indicators", [])) > 0,
            "correction_mechanisms": feedback_extracted.get("mechanisms", []),
            "feedback_loops": feedback_extracted.get("loops", []),
            "learning_processes": feedback_extracted.get("learning", []),
            "prior_updates": prior_updates,
```

```
            "uncertainty_reduction": uncertainty_reduction,
            "uncertainty_history": uncertainty_history,
            "node_importance": node_importance,
            "learning_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "feedback_mechanisms": len(feedback_extracted.get("mechanisms", [])),
                "learning_processes": len(feedback_extracted.get("learning", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q5_ContextualAdaptabilityEvaluator(BaseExecutor):
    """
    Evaluates contextual adaptation: differential impacts and territorial constraints.

    Methods (from D6-Q5):
    - CausalExtractor._calculate_language_specificity
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine.diagnose_critical_links
    - CausalInferenceSetup.identify_failure_points
    - CausalInferenceSetup._get_dynamics_pattern
    - SemanticProcessor.chunk_text
    - SemanticProcessor._detect_pdm_structure
    - SemanticProcessor._detect_table
    - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate language specificity
        language_specificity = self._execute_method(
            "CausalExtractor", "_calculate_language_specificity", context
        )

        # Step 2: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 3: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 4: Identify failure points
```

```
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 5: Get dynamics pattern
        dynamics_pattern = self._execute_method(
            "CausalInferenceSetup", "_get_dynamics_pattern", context
        )

        # Step 6: Process text structure
        text_chunks = self._execute_method(
            "SemanticProcessor", "chunk_text", context
        )
        pdm_structure = self._execute_method(
            "SemanticProcessor", "_detect_pdm_structure", context,
            chunks=text_chunks
        )
        table_detection = self._execute_method(
            "SemanticProcessor", "_detect_table", context,
            chunks=text_chunks
        )

        # Step 7: Generate traceability record
        traceability = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            specificity=language_specificity
        )

        raw_evidence = {
            "context_adaptation": language_specificity.get("adaptation_level", 0),
            "differential_impacts_recognized": critical_links.get("differential_groups", []),
            "specific_groups_mentioned": critical_links.get("target_groups", []),
            "territorial_constraints": failure_points.get("territorial", []),
            "local_context_integration": pdm_structure.get("local_sections", []),
            "language_specificity": language_specificity,
            "temporal_coherence": temporal_coherence,
            "dynamics_pattern": dynamics_pattern,
            "structure_analysis": pdm_structure,
            "traceability": traceability
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "groups_identified": len(critical_links.get("target_groups", [])),
                "territorial_constraints": len(failure_points.get("territorial", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


```

```
# ============================================================================
# EXECUTOR REGISTRY
# ============================================================================

EXECUTOR_REGISTRY = {
    "D1-Q1": D1_Q1_QuantitativeBaselineExtractor,
    "D1-Q2": D1_Q2_ProblemDimensioningAnalyzer,
    "D1-Q3": D1_Q3_BudgetAllocationTracer,
    "D1-Q4": D1_Q4_InstitutionalCapacityIdentifier,
    "D1-Q5": D1_Q5_ScopeJustificationValidator,

    "D2-Q1": D2_Q1_StructuredPlanningValidator,
    "D2-Q2": D2_Q2_InterventionLogicInferencer,
    "D2-Q3": D2_Q3_RootCauseLinkageAnalyzer,
    "D2-Q4": D2_Q4_RiskManagementAnalyzer,
    "D2-Q5": D2_Q5_StrategicCoherenceEvaluator,

    "D3-Q1": D3_Q1_IndicatorQualityValidator,
    "D3-Q2": D3_Q2_TargetProportionalityAnalyzer,
    "D3-Q3": D3_Q3_TraceabilityValidator,
    "D3-Q4": D3_Q4_TechnicalFeasibilityEvaluator,
    "D3-Q5": D3_Q5_OutputOutcomeLinkageAnalyzer,

    "D4-Q1": D4_Q1_OutcomeMetricsValidator,
    "D4-Q2": D4_Q2_CausalChainValidator,
    "D4-Q3": D4_Q3_AmbitionJustificationAnalyzer,
    "D4-Q4": D4_Q4_ProblemSolvencyEvaluator,
    "D4-Q5": D4_Q5_VerticalAlignmentValidator,

    "D5-Q1": D5_Q1_LongTermVisionAnalyzer,
    "D5-Q2": D5_Q2_CompositeMeasurementValidator,
    "D5-Q3": D5_Q3_IntangibleMeasurementAnalyzer,
    "D5-Q4": D5_Q4_SystemicRiskEvaluator,
    "D5-Q5": D5_Q5_RealismAndSideEffectsAnalyzer,

    "D6-Q1": D6_Q1_ExplicitTheoryBuilder,
    "D6-Q2": D6_Q2_LogicalProportionalityValidator,
    "D6-Q3": D6_Q3_ValidationTestingAnalyzer,
    "D6-Q4": D6_Q4_FeedbackLoopAnalyzer,
    "D6-Q5": D6_Q5_ContextualAdaptabilityEvaluator,
}


# ============================================================================
# PHASE 2 ORCHESTRATION
# ============================================================================


def _build_method_executor() -> MethodExecutor:
    """Construct a canonical MethodExecutor via the factory wiring."""
    bundle = build_processor()
    method_executor = getattr(bundle, "method_executor", None)
    if not isinstance(method_executor, MethodExecutor):
        raise RuntimeError("ProcessorBundle did not provide a valid MethodExecutor instance.")
    return method_executor

```

```
def _canonical_metadata(executor_id: str) -> Dict[str, Any]:
    """Build canonical metadata block using canonical_notation."""
    metadata: Dict[str, Any] = {}
    try:
        dim_key = executor_id.split("-")[0]
        dim_info = get_dimension_info(dim_key)
        metadata["dimension_code"] = dim_info.code
        metadata["dimension_label"] = dim_info.label
    except Exception:
        pass

    if executor_id in CANONICAL_QUESTION_LABELS:
        metadata["canonical_question"] = CANONICAL_QUESTION_LABELS[executor_id]
    return metadata


def run_phase2_executors(context_package: Dict[str, Any],
                         policy_areas: List[str]) -> Dict[str, Any]:
    """
    Phase 2 Entry Point: Runs all 30 executors for each policy area.

    Args:
        context_package: Canonical package with document data from Phase 1
        policy_areas: List of policy area identifiers to analyze

    Returns:
        Dict mapping policy_area -> executor_id -> raw_evidence
    """
    results = {}
    method_executor = _build_method_executor()

    for policy_area in policy_areas:
        print(f"\n{'='*80}")
        print(f"Processing Policy Area: {policy_area}")
        print(f"{'='*80}\n")

        # Prepare context for this policy area
        area_context = {
            **context_package,
            "policy_area": policy_area
        }

        # Execute all 30 executors
        area_results = {}
        for executor_id, executor_class in EXECUTOR_REGISTRY.items():
            print(f"Running {executor_id}: {executor_class.__name__}...")

            try:
                # Instantiate executor with config
                config = load_executor_config(executor_id)
                executor = executor_class(executor_id, config, method_executor=method_executor)

                # Execute and collect results
                result = executor.execute(area_context)
                # Append canonical metadata consistently
```

```
                result_metadata = result.get("metadata", {})
                result_metadata.update(_canonical_metadata(executor_id))
                result["metadata"] = result_metadata
                area_results[executor_id] = result

                print(f"  â\234\223 Success: {len(result['metadata']['methods_executed'])} methods executed")

            except ExecutorFailure as e:
                print(f"  â\234\227 FAILED: {str(e)}")
                raise  # Re-raise to stop execution as per requirement

        results[policy_area] = area_results

    return results


def load_executor_config(executor_id: str) -> Dict[str, Any]:
    """
    Load executor configuration from JSON contract.

    Args:
        executor_id: Executor identifier (e.g., "D1-Q1")

    Returns:
        Configuration dictionary from JSON contract
    """
    import json
    from pathlib import Path

    config_path = Path(f"config/executor_contracts/{executor_id}.json")

    if not config_path.exists():
        raise FileNotFoundError(f"Executor config not found: {config_path}")

    with open(config_path, 'r', encoding='utf-8') as f:
        return json.load(f)


# ============================================================================
# EXAMPLE USAGE
# ============================================================================

if __name__ == "__main__":
    # Example context package from Phase 1
    context_package = {
        "document_path": "data/pdm_municipality_xyz.pdf",
        "document_text": "...",  # Full document text
        "tables": [],  # Extracted tables from Phase 1
        "embeddings": {},  # Precomputed embeddings
        "entities": [],  # Pre-extracted entities
        "metadata": {
            "municipality": "Municipality XYZ",
            "year": 2024,
            "pages": 150
        }
    }
```

```

    # Policy areas to analyze
    policy_areas = [
        "PA01",  # Education
        "PA02",  # Health
        "PA03",  # Infrastructure
        # ... up to 10+ policy areas
    ]

    # Run Phase 2
    try:
        results = run_phase2_executors(context_package, policy_areas)
        print("\n" + "="*80)
        print("PHASE 2 COMPLETED SUCCESSFULLY")
        print("="*80)
        print(f"Processed {len(policy_areas)} policy areas")
        print(f"Executed {len(EXECUTOR_REGISTRY)} executors per area")
        print(f"Total executions: {len(policy_areas) * len(EXECUTOR_REGISTRY)}")

    except ExecutorFailure as e:
        print("\n" + "="*80)
        print("PHASE 2 FAILED")
        print("="*80)
        print(f"Error: {str(e)}")
        print("Execution halted as per requirement: any method failure = executor failure")



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/factory.py
================================================================================

"""
Factory module â\200\224 canonical Dependency Injection (DI) and access control for F.A.R.F.A.N.

This module is the SINGLE AUTHORITATIVE BOUNDARY for:
- Canonical monolith access (CanonicalQuestionnaire) - loaded ONCE with integrity verification
- Signal registry construction (QuestionnaireSignalRegistry v2.0) from canonical source ONLY
- Method injection via MethodExecutor with signal registry DI
- Orchestrator construction with full DI (questionnaire, method_executor, executor_config)
- EnrichedSignalPack creation and injection per executor (30 executors)
- Hard contracts and validation constants for Phase 1
- SeedRegistry singleton initialization for determinism

METHOD DISPENSARY PATTERN - Core Architecture:
==============================================

The pipeline uses a "method dispensary" pattern where monolithic analyzer classes
serve as "dispensaries" that provide methods to executors. This architecture enables:

1. LOOSE COUPLING: Executors orchestrate methods without direct imports
2. PARTIAL REUSE: Same method used by multiple executors with different contexts
3. CENTRALIZED MANAGEMENT: All method routing through MethodExecutor with validation
4. SIGNAL AWARENESS: Methods receive signal packs for pattern matching

Dispensary Registry (~20 monolith classes, 240+ methods):
```

- IndustrialPolicyProcessor (17 methods): Pattern matching, evidence extraction
- PDETMunicipalPlanAnalyzer (52+ methods): LARGEST - financial, causal, entity analysis
- CausalExtractor (28 methods): Goal extraction, causal hierarchy, semantic distance
- FinancialAuditor (13 methods): Budget tracing, allocation gaps, sufficiency
- BayesianMechanismInference (14 methods): Necessity/sufficiency tests, coherence
- BayesianCounterfactualAuditor (9 methods): SCM construction, refutation
- TextMiningEngine (8 methods): Critical link diagnosis, intervention generation
- SemanticAnalyzer (12 methods): Semantic cube, domain classification
- PerformanceAnalyzer (5 methods): Performance metrics, loss functions
- PolicyContradictionDetector (8 methods): Contradiction detection, coherence
- [... 10+ more classes]

Executor Usage Pattern:

Each of 30 executors uses a UNIQUE COMBINATION of methods:
- D1-Q1 (QuantitativeBaselineExtractor): 17 methods from 9 classes
- D3-Q2 (TargetProportionalityAnalyzer): 24 methods from 7 classes
- D3-Q5 (OutputOutcomeLinkageAnalyzer): 28 methods from 6 classes
- D6-Q3 (ValidationTestingAnalyzer): 8 methods from 4 classes

Methods are orchestrated via:
```python
result = self.method_executor.execute(
    class_name="PDETMunicipalPlanAnalyzer",
    method_name="_score_indicators",
    document=doc,
    signal_pack=pack,
    **context
)
```

NOT ALL METHODS ARE USED:
- Monoliths contain more methods than executors need
- Only methods in executors_methods.json are actively used
- Phase 1 (ingestion) uses additional methods not in executor contracts
- 14 methods in validation failures (deprecated/private)

Design Principles (Factory Pattern + DI):
===

1. FACTORY PATTERN: AnalysisPipelineFactory is the ONLY place that instantiates:
   - Orchestrator, MethodExecutor, QuestionnaireSignalRegistry, BaseExecutor instances
   - NO other module should directly instantiate these classes

2. DEPENDENCY INJECTION: All components receive dependencies via __init__:
   - Orchestrator receives: questionnaire, method_executor, executor_config, validation_constants
   - MethodExecutor receives: method_registry, arg_router, signal_registry
   - BaseExecutor (30 classes) receive: enriched_signal_pack, method_executor, config

3. CANONICAL MONOLITH CONTROL:
   - load_questionnaire() called ONCE by factory only (singleton + integrity hash)
   - Orchestrator uses self.questionnaire object, NEVER file paths
   - Search codebase: NO other load_questionnaire() calls should exist

4. SIGNAL REGISTRY CONTROL:

```
   - create_signal_registry(questionnaire) - from canonical source ONLY
   - signal_loader.py MUST BE DELETED (legacy JSON loaders eliminated)
   - Registry injected into MethodExecutor, NOT accessed globally

5. ENRICHED SIGNAL PACK INJECTION:
   - Factory builds EnrichedSignalPack per executor (semantic expansion + context filtering)
   - Each BaseExecutor receives its specific pack, NOT full registry

6. DETERMINISM:
   - SeedRegistry singleton initialized by factory for reproducibility
   - ExecutorConfig encapsulates operational params (max_tokens, retries)

7. PHASE 1 HARD CONTRACTS:
   - Validation constants (P01_EXPECTED_CHUNK_COUNT=60, etc.) loaded by factory
   - Injected into Orchestrator for Phase 1 chunk validation
   - Execution FAILS if contracts violated

SIN_CARRETA Compliance:
- All construction paths emit structured telemetry with timestamps and hashes
- Determinism enforced via explicit validation of canonical questionnaire integrity
- Contract assertions guard all factory outputs (no silent degradation)
- Auditability via immutable ProcessorBundle with provenance metadata
"""

from __future__ import annotations

import hashlib
import json
import logging
import time
from collections.abc import Mapping
from dataclasses import dataclass, field
from typing import Any

from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
from farfan_pipeline.core.orchestrator.core import MethodExecutor, Orchestrator
from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
from farfan_pipeline.core.orchestrator.method_registry import (
    MethodRegistry,
    setup_default_instantiation_rules,
)
from farfan_pipeline.core.orchestrator.questionnaire import (
    CanonicalQuestionnaire,
    load_questionnaire,
)
from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
    EnrichedSignalPack,
    create_enriched_signal_pack,
)
from farfan_pipeline.core.orchestrator.signal_registry import (
    QuestionnaireSignalRegistry,
    create_signal_registry,
)

# Phase 1 validation constants module
```

```
try:
    from farfan_pipeline.config.validation_constants import (
        PHASE1_VALIDATION_CONSTANTS,
        load_validation_constants,
    )
    VALIDATION_CONSTANTS_AVAILABLE = True
except ImportError:
    PHASE1_VALIDATION_CONSTANTS = {}
    VALIDATION_CONSTANTS_AVAILABLE = False

# Optional: CoreModuleFactory for I/O helpers
try:
    from farfan_pipeline.core.orchestrator.core_module_factory import CoreModuleFactory
    CORE_MODULE_FACTORY_AVAILABLE = True
except ImportError:
    CoreModuleFactory = None  # type: ignore
    CORE_MODULE_FACTORY_AVAILABLE = False

# SeedRegistry for determinism (REQUIRED for production)
try:
    from farfan_pipeline.core.orchestrator.seed_registry import SeedRegistry
    SEED_REGISTRY_AVAILABLE = True
except ImportError:
    SeedRegistry = None  # type: ignore
    SEED_REGISTRY_AVAILABLE = False

logger = logging.getLogger(__name__)


# ============================================================================
# Exceptions
# ============================================================================


class FactoryError(Exception):
    """Base exception for factory construction failures."""
    pass


class QuestionnaireValidationError(FactoryError):
    """Raised when questionnaire validation fails."""
    pass


class IntegrityError(FactoryError):
    """Raised when questionnaire integrity check (SHA-256) fails."""
    pass


class RegistryConstructionError(FactoryError):
    """Raised when signal registry construction fails."""
    pass


class ExecutorConstructionError(FactoryError):
    """Raised when method executor construction fails."""
```

```
    pass


class SingletonViolationError(FactoryError):
    """Raised when singleton pattern is violated."""
    pass


# ===========================================================================
# Processor Bundle (typed DI container with provenance)
# ===========================================================================


@dataclass(frozen=True)
class ProcessorBundle:
    """Aggregated orchestrator dependencies built by the Factory.

    This is the COMPLETE DI container returned by AnalysisPipelineFactory.

    Attributes:
        orchestrator: Fully configured Orchestrator (main entry point).
        method_executor: MethodExecutor with signal registry injected.
        questionnaire: Immutable, validated CanonicalQuestionnaire (monolith).
        signal_registry: QuestionnaireSignalRegistry v2.0 from canonical source.
        executor_config: ExecutorConfig for operational parameters.
        enriched_signal_packs: Dict of EnrichedSignalPack per policy area.
        validation_constants: Phase 1 hard contracts (chunk counts, etc.).
        core_module_factory: Optional CoreModuleFactory for I/O helpers.
        seed_registry_initialized: Whether SeedRegistry singleton was set up.
        provenance: Construction metadata for audit trails.
    """

    orchestrator: Orchestrator
    method_executor: MethodExecutor
    questionnaire: CanonicalQuestionnaire
    signal_registry: QuestionnaireSignalRegistry
    executor_config: ExecutorConfig
    enriched_signal_packs: dict[str, EnrichedSignalPack]
    validation_constants: dict[str, Any]
    core_module_factory: Any | None = None
    seed_registry_initialized: bool = False
    provenance: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self) -> None:
        """SIN_CARRETA Â§ Contract Enforcement: validate bundle integrity."""
        errors = []

        # Critical components validation
        if self.orchestrator is None:
            errors.append("orchestrator must not be None")
        if self.method_executor is None:
            errors.append("method_executor must not be None")
        if self.questionnaire is None:
            errors.append("questionnaire must not be None")
        if self.signal_registry is None:
            errors.append("signal_registry must not be None")
```

```
        if self.executor_config is None:
            errors.append("executor_config must not be None")
        if self.enriched_signal_packs is None:
            errors.append("enriched_signal_packs must not be None")
        elif not isinstance(self.enriched_signal_packs, dict):
            errors.append("enriched_signal_packs must be dict[str, EnrichedSignalPack]")

        if self.validation_constants is None:
            errors.append("validation_constants must not be None")

        # Provenance validation
        if not self.provenance.get("construction_timestamp_utc"):
            errors.append("provenance must include construction_timestamp_utc")
        if not self.provenance.get("canonical_sha256"):
            errors.append("provenance must include canonical_sha256")
        if self.provenance.get("signal_registry_version") != "2.0":
            errors.append("provenance must indicate signal_registry_version=2.0")

        # Factory pattern enforcement check
        if not self.provenance.get("factory_instantiation_confirmed"):
            errors.append("provenance must confirm factory instantiation (not direct construction)")

        if errors:
            raise FactoryError(f"ProcessorBundle validation failed: {'; '.join(errors)}")

        logger.info(
            "processor_bundle_validated "
            "canonical_sha256=%s construction_ts=%s policy_areas=%d validation_constants=%d",
            self.provenance.get("canonical_sha256", "")[:16],
            self.provenance.get("construction_timestamp_utc"),
            len(self.enriched_signal_packs),
            len(self.validation_constants),
        )


# ============================================================================
# Analysis Pipeline Factory (Main Factory Class)
# ============================================================================


class AnalysisPipelineFactory:
    """Factory for constructing the complete analysis pipeline.

    This is the ONLY class that should instantiate:
    - Orchestrator
    - MethodExecutor
    - QuestionnaireSignalRegistry
    - BaseExecutor instances (30 executor classes)

    CRITICAL: No other module should directly instantiate these classes.
    All dependencies are injected via constructor parameters.

    Usage:
        factory = AnalysisPipelineFactory(
            questionnaire_path="path/to/questionnaire.json",
            expected_hash="abc123...",
```

```
            seed=42
        )
        bundle = factory.create_orchestrator()
        orchestrator = bundle.orchestrator
    """

    # Singleton tracking for load_questionnaire() call
    _questionnaire_loaded = False
    _questionnaire_instance: CanonicalQuestionnaire | None = None

    def __init__(
        self,
        *,
        questionnaire_path: str | None = None,
        expected_questionnaire_hash: str | None = None,
        executor_config: ExecutorConfig | None = None,
        validation_constants: dict[str, Any] | None = None,
        enable_intelligence_layer: bool = True,
        seed_for_determinism: int | None = None,
        strict_validation: bool = True,
    ):
        """Initialize the Analysis Pipeline Factory.

        Args:
            questionnaire_path: Path to canonical questionnaire JSON.
            expected_questionnaire_hash: Expected SHA-256 hash for integrity check.
            executor_config: Custom executor configuration (if None, uses default).
            validation_constants: Phase 1 validation constants (if None, loads from config).
            enable_intelligence_layer: Whether to build enriched signal packs.
            seed_for_determinism: Seed for SeedRegistry singleton.
            strict_validation: If True, fail on any validation error.
        """
        self._questionnaire_path = questionnaire_path
        self._expected_hash = expected_questionnaire_hash
        self._executor_config = executor_config
        self._validation_constants = validation_constants
        self._enable_intelligence = enable_intelligence_layer
        self._seed = seed_for_determinism
        self._strict = strict_validation

        # Internal state (set during construction)
        self._canonical_questionnaire: CanonicalQuestionnaire | None = None
        self._signal_registry: QuestionnaireSignalRegistry | None = None
        self._method_executor: MethodExecutor | None = None
        self._enriched_packs: dict[str, EnrichedSignalPack] = {}

        logger.info(
            "factory_initialized questionnaire_path=%s intelligence_layer=%s seed=%s",
            questionnaire_path or "default",
            enable_intelligence_layer,
            seed_for_determinism is not None,
        )

    def create_orchestrator(self) -> ProcessorBundle:
        """Create fully configured Orchestrator with all dependencies injected.

```

```
        This is the PRIMARY ENTRY POINT for the factory.
        Returns a complete ProcessorBundle with Orchestrator ready to use.

        Returns:
            ProcessorBundle: Immutable bundle with all dependencies wired.

        Raises:
            QuestionnaireValidationError: If questionnaire validation fails.
            IntegrityError: If questionnaire hash doesn’t match expected.
            RegistryConstructionError: If signal registry construction fails.
            ExecutorConstructionError: If method executor construction fails.
        """
        construction_start = time.time()
        timestamp_utc = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())

        logger.info("factory_create_orchestrator_start timestamp=%s", timestamp_utc)

        try:
            # Step 1: Load canonical questionnaire (ONCE, with integrity check)
            self._load_canonical_questionnaire()

            # Step 2: Build signal registry from canonical source
            self._build_signal_registry()

            # Step 3: Build enriched signal packs (intelligence layer)
            self._build_enriched_signal_packs()

            # Step 4: Initialize seed registry for determinism
            seed_initialized = self._initialize_seed_registry()

            # Step 5: Build method executor with signal registry DI
            self._build_method_executor()

            # Step 6: Load Phase 1 validation constants
            validation_constants = self._load_validation_constants()

            # Step 7: Get or create executor config
            executor_config = self._get_executor_config()

            # Step 8: Build orchestrator with full DI
            orchestrator = self._build_orchestrator(
                executor_config=executor_config,
                validation_constants=validation_constants,
            )

            # Step 9: Assemble provenance metadata
            construction_duration = time.time() - construction_start
            canonical_hash = self._compute_questionnaire_hash()

            provenance = {
                "construction_timestamp_utc": timestamp_utc,
                "canonical_sha256": canonical_hash,
                "signal_registry_version": "2.0",
                "intelligence_layer_enabled": self._enable_intelligence,
                "enriched_packs_count": len(self._enriched_packs),
                "validation_constants_count": len(validation_constants),
```

```
                "construction_duration_seconds": round(construction_duration, 3),
                "seed_registry_initialized": seed_initialized,
                "core_module_factory_available": CORE_MODULE_FACTORY_AVAILABLE,
                "strict_validation": self._strict,
                "factory_instantiation_confirmed": True,  # Critical for bundle validation
                "factory_class": "AnalysisPipelineFactory",
            }

            # Step 10: Build complete bundle
            bundle = ProcessorBundle(
                orchestrator=orchestrator,
                method_executor=self._method_executor,
                questionnaire=self._canonical_questionnaire,
                signal_registry=self._signal_registry,
                executor_config=executor_config,
                enriched_signal_packs=self._enriched_packs,
                validation_constants=validation_constants,
                core_module_factory=self._build_core_module_factory(),
                seed_registry_initialized=seed_initialized,
                provenance=provenance,
            )

            logger.info(
                "factory_create_orchestrator_complete duration=%.3fs hash=%s",
                construction_duration,
                canonical_hash[:16],
            )

            return bundle

        except Exception as e:
            logger.error("factory_create_orchestrator_failed error=%s", str(e), exc_info=True)
            raise FactoryError(f"Failed to create orchestrator: {e}") from e

    # ==========================================================================
    # Internal Construction Methods
    # ==========================================================================

    def _load_canonical_questionnaire(self) -> None:
        """Load canonical questionnaire with singleton enforcement and integrity check.

        CRITICAL REQUIREMENTS:
        1. This is the ONLY place in the codebase that calls load_questionnaire()
        2. Must enforce singleton pattern (only load once)
        3. Must verify SHA-256 hash for integrity
        4. Must raise IntegrityError if hash doesn’t match

        Raises:
            SingletonViolationError: If load_questionnaire() already called.
            IntegrityError: If questionnaire hash doesn’t match expected.
            QuestionnaireValidationError: If questionnaire structure invalid.
        """
        # Enforce singleton pattern
        if AnalysisPipelineFactory._questionnaire_loaded:
            if AnalysisPipelineFactory._questionnaire_instance is not None:
                logger.info("questionnaire_singleton_reused using_cached_instance")
```

```
                self._canonical_questionnaire = AnalysisPipelineFactory._questionnaire_instance
                return
            else:
                raise SingletonViolationError(
                    "load_questionnaire() was called but instance is None. "
                    "This indicates a singleton pattern violation."
                )

        logger.info("questionnaire_loading_start path=%s", self._questionnaire_path or "default")

        try:
            # Load questionnaire (this should be the ONLY call in the entire codebase)
            questionnaire = load_questionnaire(self._questionnaire_path)

            # Mark singleton as loaded
            AnalysisPipelineFactory._questionnaire_loaded = True
            AnalysisPipelineFactory._questionnaire_instance = questionnaire

            # Compute integrity hash
            actual_hash = self._compute_questionnaire_hash_from_instance(questionnaire)

            # Verify integrity if expected hash provided
            if self._expected_hash is not None:
                if actual_hash != self._expected_hash:
                    raise IntegrityError(
                        f"Questionnaire integrity check FAILED. "
                        f"Expected: {self._expected_hash[:16]}... "
                        f"Actual: {actual_hash[:16]}... "
                        f"The canonical questionnaire may have been tampered with."
                    )
                logger.info("questionnaire_integrity_verified hash=%s", actual_hash[:16])
            else:
                logger.warning(
                    "questionnaire_integrity_not_verified no_expected_hash_provided "
                    "actual_hash=%s",
                    actual_hash[:16]
                )

            # Validate structure
            if not hasattr(questionnaire, 'questions'):
                if self._strict:
                    raise QuestionnaireValidationError("Questionnaire missing 'questions' attribute")
                logger.warning("questionnaire_validation_warning missing_questions_attribute")

            questions = getattr(questionnaire, 'questions', [])
            if not questions:
                if self._strict:
                    raise QuestionnaireValidationError("Questionnaire has no questions")
                logger.warning("questionnaire_validation_warning no_questions")

            self._canonical_questionnaire = questionnaire

            logger.info(
                "questionnaire_loaded_successfully questions=%d hash=%s singleton=established",
                len(questions),
                actual_hash[:16],
```

```
            )

        except Exception as e:
            if isinstance(e, (IntegrityError, SingletonViolationError, QuestionnaireValidationError)):
                raise
            raise QuestionnaireValidationError(f"Failed to load questionnaire: {e}") from e

    def _build_signal_registry(self) -> None:
        """Build signal registry from canonical questionnaire.

        CRITICAL REQUIREMENTS:
        1. Use create_signal_registry(questionnaire) ONLY
        2. Pass self._canonical_questionnaire as ONLY argument
        3. NO other signal loading methods allowed (signal_loader.py DELETED)

        Raises:
            RegistryConstructionError: If registry construction fails.
        """
        if self._canonical_questionnaire is None:
            raise RegistryConstructionError(
                "Cannot build signal registry: canonical questionnaire not loaded"
            )

        logger.info("signal_registry_building_start")

        try:
            # Build registry from canonical source ONLY
            registry = create_signal_registry(self._canonical_questionnaire)

            # Validate registry
            if not hasattr(registry, 'get_all_policy_areas'):
                if self._strict:
                    raise RegistryConstructionError("Registry missing required methods")
                logger.warning("registry_validation_warning missing_methods")

            policy_areas = registry.get_all_policy_areas() if hasattr(registry, 'get_all_policy_areas') else []

            self._signal_registry = registry

            logger.info(
                "signal_registry_built_successfully version=2.0 policy_areas=%d",
                len(policy_areas),
            )

        except Exception as e:
            if isinstance(e, RegistryConstructionError):
                raise
            raise RegistryConstructionError(f"Failed to build signal registry: {e}") from e

    def _build_enriched_signal_packs(self) -> None:
        """Build enriched signal packs for all policy areas.

        Each BaseExecutor receives its own EnrichedSignalPack (NOT full registry).
        Pack includes semantic expansion and context filtering.

        Raises:
```

```
            RegistryConstructionError: If pack construction fails in strict mode.
        """
        if not self._enable_intelligence:
            logger.info("enriched_packs_disabled intelligence_layer=off")
            self._enriched_packs = {}
            return

        if self._signal_registry is None:
            raise RegistryConstructionError(
                "Cannot build enriched packs: signal registry not built"
            )

        logger.info("enriched_packs_building_start")

        enriched_packs: dict[str, EnrichedSignalPack] = {}

        try:
            policy_areas = self._signal_registry.get_all_policy_areas() if hasattr(self._signal_registry, 'get_all_policy_areas') else []

            if not policy_areas:
                logger.warning("enriched_packs_warning no_policy_areas_found")
                self._enriched_packs = enriched_packs
                return

            for policy_area_id in policy_areas:
                try:
                    # Get base pack from registry
                    base_pack = self._signal_registry.get(policy_area_id) if hasattr(self._signal_registry, 'get') else None

                    if base_pack is None:
                        logger.warning("base_pack_missing policy_area=%s", policy_area_id)
                        continue

                    # Create enriched pack (semantic expansion + context filtering)
                    enriched_pack = create_enriched_signal_pack(
                        base_pack=base_pack,
                        questionnaire=self._canonical_questionnaire,
                    )

                    enriched_packs[policy_area_id] = enriched_pack

                    logger.debug(
                        "enriched_pack_created policy_area=%s",
                        policy_area_id,
                    )

                except Exception as e:
                    msg = f"Failed to create enriched pack for {policy_area_id}: {e}"
                    if self._strict:
                        raise RegistryConstructionError(msg) from e
                    logger.error("enriched_pack_creation_failed policy_area=%s", policy_area_id, exc_info=True)

            self._enriched_packs = enriched_packs

            logger.info(
                "enriched_packs_built_successfully count=%d",
```

```
                len(enriched_packs),
            )

        except Exception as e:
            if isinstance(e, RegistryConstructionError):
                raise
            raise RegistryConstructionError(f"Failed to build enriched packs: {e}") from e

    def _initialize_seed_registry(self) -> bool:
        """Initialize SeedRegistry singleton for deterministic operations.

        Returns:
            bool: True if seed registry was initialized, False otherwise.
        """
        if not SEED_REGISTRY_AVAILABLE:
            logger.warning("seed_registry_unavailable module_not_found determinism_not_guaranteed")
            return False

        if self._seed is None:
            logger.info("seed_registry_not_initialized no_seed_provided")
            return False

        try:
            SeedRegistry.initialize(master_seed=self._seed)
            logger.info("seed_registry_initialized master_seed=%d determinism=enabled", self._seed)
            return True
        except Exception:
            logger.error("seed_registry_initialization_failed", exc_info=True)
            return False

    def _build_method_executor(self) -> None:
        """Build MethodExecutor with full dependency wiring.

        CRITICAL INTEGRATION POINT - Method Dispensary Pattern:
        ==========================================================

        This is where the "monolith dispensaries" get wired into the pipeline.
        The 30 executors orchestrate methods from these dispensaries WITHOUT
        direct imports or tight coupling to the monolith implementations.

        Architecture Flow:
        -----------------
        1. build_class_registry() loads the "method dispensaries" (monoliths):
           - IndustrialPolicyProcessor: 17 methods used across D1-Q1, D1-Q5, D2-Q2, D3-Q1
           - BayesianEvidenceScorer: 8 methods for confidence calculation
           - PDETMunicipalPlanAnalyzer: 52+ methods (LARGEST dispensary)
             Used in: D1-Q2, D1-Q3, D1-Q4, D2-Q1, D3-Q2, D3-Q3, D3-Q4, D3-Q5,
                      D4-Q1, D4-Q2, D4-Q3, D5-Q1, D5-Q2, D5-Q4, D5-Q5
           - CausalExtractor: 28 methods for causal inference
           - FinancialAuditor: 13 methods for financial analysis
           - BayesianMechanismInference: 14 methods for mechanism testing
           - [... 15+ more classes from farfan_core]

           Total: ˜240 method pairs validated (see executor_factory_validation.json)

        2. These classes are NOT instantiated here - they’re registered as TYPES.
```

```
           Instantiation happens lazily via MethodRegistry when methods are called.

        3. ExtendedArgRouter receives the class registry and provides:
           - 30+ special routes for high-traffic methods (see arg_router.py)
           - Generic routing via signature inspection for all other methods
           - Strict argument validation (no silent parameter drops)
           - **kwargs awareness for forward compatibility

        4. MethodExecutor combines three critical components:
           - MethodRegistry: Instantiation rules + shared instances (e.g., MunicipalOntology)
           - ArgRouter: Method routing + argument validation
           - SignalRegistry: Injected for signal-aware methods

        5. Each of the 30 Executors orchestrates methods via:
           ```python
           result = self.method_executor.execute(
               class_name="PDETMunicipalPlanAnalyzer",
               method_name="_score_indicators",
               **payload  # document, question_id, signal_pack, etc.
           )
           ```

        Method Reuse Pattern:

        - Methods are PARTIALLY reused across executors (not fully shared)
        - Example: "_score_indicators" used in D3-Q1, D3-Q2, D4-Q1
        - Example: "_test_sufficiency" used in D2-Q2, D3-Q2, D3-Q4
        - Each executor uses a DIFFERENT COMBINATION of methods
        - Total unique combinations: 30 executors Ã\227 avg 12 methods = ~360 method calls

        Not All Methods Are Used:

        The monoliths contain MORE methods than executors need.
        Only methods listed in executors_methods.json are actively used.
        Phase 1 (ingestion) uses additional methods not in executor contracts.

        Validation:

        - executor_factory_validation.json: 243 pairs validated, 14 failures
        - Failures are methods NOT in catalog (likely private/deprecated)
        - All executor contracts reference validated methods only

        Signal Registry Integration:

        Signal registry is injected so methods can access:
        - Policy-area-specific patterns
        - Expected elements for validation
        - Semantic enrichment via intelligence layer

        Raises:
            ExecutorConstructionError: If executor construction fails.

        See Also:
            - executors_methods.json: Complete executorâ\206\222methods mapping
            - executor_factory_validation.json: Method catalog validation
            - arg_router.py: Special routes and routing logic
```

```
            - class_registry.py: Monolith class paths (_CLASS_PATHS)
        """
        if self._signal_registry is None:
            raise ExecutorConstructionError(
                "Cannot build method executor: signal registry not built"
            )

        logger.info("method_executor_building_start dispensaries=loading")

        try:
            # Step 1: Build method registry with special instantiation rules
            # MethodRegistry handles shared instances (e.g., MunicipalOntology singleton)
            # and custom instantiation logic for complex analyzers
            method_registry = MethodRegistry()
            setup_default_instantiation_rules(method_registry)

            logger.info("method_registry_built instantiation_rules=configured")

            # Step 2: Build class registry - THE METHOD DISPENSARIES
            # This loads ~20 monolith classes with 240+ methods total
            # Each class is a "dispensary" that provides methods to executors
            class_registry = build_class_registry()

            logger.info(
                "class_registry_built dispensaries=%d total_methods=240+",
                len(class_registry)
            )

            # Step 3: Build extended arg router with special routes
            # Handles 30+ high-traffic method routes + generic routing
            arg_router = ExtendedArgRouter(class_registry)

            special_routes = arg_router.get_special_route_coverage() if hasattr(arg_router, 'get_special_route_coverage') else 0

            logger.info(
                "arg_router_built special_routes=%d generic_routing=enabled",
                special_routes
            )

            # Step 4: Build method executor WITH signal registry injected
            # This is the CORE integration point - executors call methods through this
            method_executor = MethodExecutor(
                method_registry=method_registry,
                arg_router=arg_router,
                signal_registry=self._signal_registry,  # DI: inject signal registry
            )

            # Step 5: PRE-EXECUTION CONTRACT VERIFICATION
            # Verify all 30 base executor contracts (D1-Q1 through D6-Q5) before execution
            # This ensures contract integrity and method class availability at startup
            logger.info("contract_verification_start verifying_30_base_contracts")

            from farfan_pipeline.core.orchestrator.base_executor_with_contract import (
                BaseExecutorWithContract,
            )

```

```
            verification_result = BaseExecutorWithContract.verify_all_base_contracts(
                class_registry=class_registry
            )

            if not verification_result["passed"]:
                error_summary = f"{len(verification_result['errors'])} contract validation errors"
                logger.error(
                    "contract_verification_failed errors=%d warnings=%d",
                    len(verification_result["errors"]),
                    len(verification_result.get("warnings", [])),
                )

                for error in verification_result["errors"][:10]:
                    logger.error("contract_error: %s", error)

                if self._strict:
                    raise ExecutorConstructionError(
                        f"Pre-execution contract verification failed: {error_summary}. "
                        f"See logs for details. Total errors: {len(verification_result['errors'])}"
                    )
                else:
                    logger.warning(
                        "contract_verification_failed_non_strict continuing_with_errors=%d",
                        len(verification_result["errors"])
                    )
            else:
                logger.info(
                    "contract_verification_passed verified=%d warnings=%d",
                    len(verification_result["verified_contracts"]),
                    len(verification_result.get("warnings", []))
                )

                for warning in verification_result.get("warnings", [])[:5]:
                    logger.warning("contract_warning: %s", warning)

            # Validate construction
            if not hasattr(method_executor, 'execute'):
                if self._strict:
                    raise ExecutorConstructionError("MethodExecutor missing 'execute' method")
                logger.warning("method_executor_validation_warning missing_execute")

            self._method_executor = method_executor

            logger.info(
                "method_executor_built_successfully "
                "dispensaries=%d special_routes=%d signal_registry=injected",
                len(class_registry),
                special_routes,
            )

        except Exception as e:
            if isinstance(e, ExecutorConstructionError):
                raise
            raise ExecutorConstructionError(f"Failed to build method executor: {e}") from e

    def _load_validation_constants(self) -> dict[str, Any]:
```

```
        """Load Phase 1 validation constants (hard contracts).

        These constants are injected into Orchestrator for Phase 1 validation:
        - P01_EXPECTED_CHUNK_COUNT = 60
        - P02_MIN_TABLE_COUNT = 5
        - etc.

        Returns:
            dict[str, Any]: Validation constants.
        """
        if self._validation_constants is not None:
            logger.info("validation_constants_using_provided count=%d", len(self._validation_constants))
            return self._validation_constants

        if VALIDATION_CONSTANTS_AVAILABLE:
            try:
                raw_constants = (
                    load_validation_constants()
                    if callable(load_validation_constants)
                    else PHASE1_VALIDATION_CONSTANTS
                )
                if not isinstance(raw_constants, Mapping):
                    raise TypeError(
                        f"Validation constants must be a mapping, got {type(raw_constants)!r}"
                    )

                constants = dict(raw_constants)
                logger.info("validation_constants_loaded_from_config count=%d", len(constants))
                return constants
            except Exception:
                logger.error("validation_constants_load_failed using_defaults", exc_info=True)

        # Default validation constants
        default_constants = {
            "P01_EXPECTED_CHUNK_COUNT": 60,
            "P01_MIN_CHUNK_LENGTH": 100,
            "P01_MAX_CHUNK_LENGTH": 2000,
            "P02_MIN_TABLE_COUNT": 5,
            "P02_MAX_TABLES_PER_DOCUMENT": 100,
        }

        logger.warning(
            "validation_constants_using_defaults count=%d constants_module_unavailable",
            len(default_constants),
        )

        return default_constants

    def _get_executor_config(self) -> ExecutorConfig:
        """Get or create ExecutorConfig."""
        if self._executor_config is not None:
            return self._executor_config
        return ExecutorConfig.default()

    def _build_orchestrator(
        self,
```

```
        executor_config: ExecutorConfig,
        validation_constants: dict[str, Any],
    ) -> Orchestrator:
        """Build Orchestrator with full dependency injection.

        CRITICAL: Orchestrator receives:
        1. questionnaire: CanonicalQuestionnaire (NOT file path)
        2. method_executor: MethodExecutor
        3. executor_config: ExecutorConfig
        4. validation_constants: dict (Phase 1 hard contracts)

        Args:
            executor_config: ExecutorConfig instance.
            validation_constants: Phase 1 validation constants.

        Returns:
            Orchestrator: Fully configured orchestrator.

        Raises:
            ExecutorConstructionError: If orchestrator construction fails.
        """
        if self._canonical_questionnaire is None:
            raise ExecutorConstructionError("Cannot build orchestrator: questionnaire not loaded")
        if self._method_executor is None:
            raise ExecutorConstructionError("Cannot build orchestrator: method executor not built")

        logger.info("orchestrator_building_start")

        try:
            # Build orchestrator with FULL dependency injection
            orchestrator = Orchestrator(
                questionnaire=self._canonical_questionnaire,  # DI: inject questionnaire object
                method_executor=self._method_executor,  # DI: inject method executor
                executor_config=executor_config,  # DI: inject config
                validation_constants=validation_constants,  # DI: inject Phase 1 contracts
                signal_registry=self._signal_registry,  # DI: inject signal registry
            )

            logger.info("orchestrator_built_successfully")

            return orchestrator

        except Exception as e:
            raise ExecutorConstructionError(f"Failed to build orchestrator: {e}") from e

    def _build_core_module_factory(self) -> Any | None:
        """Build CoreModuleFactory if available."""
        if not CORE_MODULE_FACTORY_AVAILABLE:
            return None

        try:
            factory = CoreModuleFactory()
            logger.info("core_module_factory_built")
            return factory
        except Exception:
            logger.error("core_module_factory_construction_error", exc_info=True)
```

```
            return None

    def _compute_questionnaire_hash(self) -> str:
        """Compute SHA-256 hash of loaded questionnaire."""
        if self._canonical_questionnaire is None:
            return ""
        return self._compute_questionnaire_hash_from_instance(self._canonical_questionnaire)

    @staticmethod
    def _compute_questionnaire_hash_from_instance(questionnaire: CanonicalQuestionnaire) -> str:
        """Compute deterministic SHA-256 hash of questionnaire content."""
        try:
            # Try to get JSON representation if available
            if hasattr(questionnaire, 'to_dict'):
                content = json.dumps(questionnaire.to_dict(), sort_keys=True)
            elif hasattr(questionnaire, '__dict__'):
                content = json.dumps(questionnaire.__dict__, sort_keys=True, default=str)
            else:
                content = str(questionnaire)

            return hashlib.sha256(content.encode('utf-8')).hexdigest()

        except Exception as e:
            logger.warning("questionnaire_hash_computation_degraded error=%s", str(e))
            # Fallback to simple string hash
            return hashlib.sha256(str(questionnaire).encode('utf-8')).hexdigest()

    def create_executor_instance(
        self,
        executor_class: type,
        policy_area_id: str,
        **extra_kwargs: Any,
    ) -> Any:
        """Create BaseExecutor instance with EnrichedSignalPack injected.

        This method is called for each of the ˜30 BaseExecutor classes.
        Each executor receives its specific EnrichedSignalPack, NOT the full registry.

        Args:
            executor_class: BaseExecutor subclass to instantiate.
            policy_area_id: Policy area identifier for signal pack selection.
            **extra_kwargs: Additional kwargs to pass to constructor.

        Returns:
            BaseExecutor instance with dependencies injected.

        Raises:
            ExecutorConstructionError: If executor instantiation fails.
        """
        if self._method_executor is None:
            raise ExecutorConstructionError(
                "Cannot create executor: method executor not built"
            )

        # Get enriched signal pack for this policy area
        enriched_pack = self._enriched_packs.get(policy_area_id)
```

```

        if enriched_pack is None and self._enable_intelligence:
            logger.warning(
                "executor_creation_warning no_enriched_pack policy_area=%s executor=%s",
                policy_area_id,
                executor_class.__name__,
            )

        try:
            # Inject dependencies into executor
            executor_instance = executor_class(
                method_executor=self._method_executor,  # DI: inject method executor
                signal_registry=self._signal_registry,  # DI: inject signal registry
                config=self._get_executor_config(),  # DI: inject config
                questionnaire_provider=self._canonical_questionnaire,  # DI: inject questionnaire
                enriched_pack=enriched_pack,  # DI: inject enriched signal pack (specific to policy area)
                **extra_kwargs,
            )

            logger.debug(
                "executor_instance_created executor=%s policy_area=%s",
                executor_class.__name__,
                policy_area_id,
            )

            return executor_instance

        except Exception as e:
            raise ExecutorConstructionError(
                f"Failed to create executor {executor_class.__name__}: {e}"
            ) from e


# ============================================================================
# Convenience Functions
# ============================================================================


def create_analysis_pipeline(
    questionnaire_path: str | None = None,
    expected_hash: str | None = None,
    seed: int | None = None,
) -> ProcessorBundle:
    """Convenience function to create complete analysis pipeline.

    This is the RECOMMENDED entry point for most use cases.

    Args:
        questionnaire_path: Path to canonical questionnaire JSON.
        expected_hash: Expected SHA-256 hash for integrity check.
        seed: Seed for reproducibility.

    Returns:
        ProcessorBundle with Orchestrator ready to use.
    """
    factory = AnalysisPipelineFactory(
```

```
        questionnaire_path=questionnaire_path,
        expected_questionnaire_hash=expected_hash,
        seed_for_determinism=seed,
        enable_intelligence_layer=True,
        strict_validation=True,
    )
    return factory.create_orchestrator()


def create_minimal_pipeline(
    questionnaire_path: str | None = None,
) -> ProcessorBundle:
    """Create minimal pipeline without intelligence layer.

    Useful for testing or when enriched signals are not needed.

    Args:
        questionnaire_path: Path to canonical questionnaire JSON.

    Returns:
        ProcessorBundle with basic dependencies only.
    """
    factory = AnalysisPipelineFactory(
        questionnaire_path=questionnaire_path,
        enable_intelligence_layer=False,
        strict_validation=False,
    )
    return factory.create_orchestrator()


# ============================================================================
# Validation and Diagnostics
# ============================================================================


def validate_factory_singleton() -> dict[str, Any]:
    """Validate that load_questionnaire() was called exactly once.

    Returns:
        dict with validation results.
    """
    return {
        "questionnaire_loaded": AnalysisPipelineFactory._questionnaire_loaded,
        "questionnaire_instance_exists": AnalysisPipelineFactory._questionnaire_instance is not None,
        "singleton_pattern_valid": (
            AnalysisPipelineFactory._questionnaire_loaded and
            AnalysisPipelineFactory._questionnaire_instance is not None
        ),
    }


def validate_bundle(bundle: ProcessorBundle) -> dict[str, Any]:
    """Validate bundle integrity and return diagnostics."""
    diagnostics = {
        "valid": True,
        "errors": [],
```

```
        "warnings": [],
        "components": {},
        "metrics": {},
    }

    # Validate orchestrator
    if bundle.orchestrator is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("orchestrator is None")
    else:
        diagnostics["components"]["orchestrator"] = "present"

    # Validate method executor
    if bundle.method_executor is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("method_executor is None")
    else:
        diagnostics["components"]["method_executor"] = "present"
        if hasattr(bundle.method_executor, 'arg_router'):
            router = bundle.method_executor.arg_router
            if hasattr(router, 'get_special_route_coverage'):
                diagnostics["metrics"]["special_routes"] = router.get_special_route_coverage()

    # Validate questionnaire
    if bundle.questionnaire is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("questionnaire is None")
    else:
        diagnostics["components"]["questionnaire"] = "present"
        if hasattr(bundle.questionnaire, 'questions'):
            diagnostics["metrics"]["question_count"] = len(bundle.questionnaire.questions)

    # Validate signal registry
    if bundle.signal_registry is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("signal_registry is None")
    else:
        diagnostics["components"]["signal_registry"] = "present"
        if hasattr(bundle.signal_registry, 'get_all_policy_areas'):
            diagnostics["metrics"]["policy_areas"] = len(bundle.signal_registry.get_all_policy_areas())

    # Validate enriched packs
    diagnostics["components"]["enriched_packs"] = len(bundle.enriched_signal_packs)
    diagnostics["metrics"]["enriched_pack_count"] = len(bundle.enriched_signal_packs)

    # Validate validation constants
    diagnostics["components"]["validation_constants"] = len(bundle.validation_constants)
    diagnostics["metrics"]["validation_constant_count"] = len(bundle.validation_constants)

    # Validate seed registry
    if not bundle.seed_registry_initialized:
        diagnostics["warnings"].append("SeedRegistry not initialized - determinism not guaranteed")

    # Check factory instantiation
    if not bundle.provenance.get("factory_instantiation_confirmed"):
        diagnostics["errors"].append("Bundle not created via AnalysisPipelineFactory")
```

```
        diagnostics["valid"] = False

    return diagnostics


def get_bundle_info(bundle: ProcessorBundle) -> dict[str, Any]:
    """Get human-readable information about bundle."""
    return {
        "construction_time": bundle.provenance.get("construction_timestamp_utc"),
        "canonical_hash": bundle.provenance.get("canonical_sha256", "")[:16],
        "policy_areas": sorted(bundle.enriched_signal_packs.keys()),
        "policy_area_count": len(bundle.enriched_signal_packs),
        "intelligence_layer": bundle.provenance.get("intelligence_layer_enabled"),
        "validation_constants": len(bundle.validation_constants),
        "construction_duration": bundle.provenance.get("construction_duration_seconds"),
        "seed_initialized": bundle.seed_registry_initialized,
        "factory_class": bundle.provenance.get("factory_class"),
    }


# ============================================================================
# Module-level Checks
# ============================================================================


def check_legacy_signal_loader_deleted() -> dict[str, Any]:
    """Check that signal_loader.py has been deleted.

    Returns:
        dict with check results.
    """
    try:
        import farfan_pipeline.core.orchestrator.signal_loader
        return {
            "legacy_loader_deleted": False,
            "error": "signal_loader.py still exists - must be deleted per architecture requirements",
        }
    except ImportError:
        return {
            "legacy_loader_deleted": True,
            "message": "signal_loader.py correctly deleted - no legacy signal loading",
        }


def verify_single_questionnaire_load_point() -> dict[str, Any]:
    """Verify that only AnalysisPipelineFactory calls load_questionnaire().

    This requires manual code search but provides guidance.

    Returns:
        dict with verification instructions.
    """
    return {
        "verification_required": True,
        "search_command": "grep -r 'load_questionnaire(' --exclude-dir=__pycache__ --exclude='*.pyc'",
        "expected_result": "Should ONLY appear in: factory.py (AnalysisPipelineFactory._load_canonical_questionnaire)",
```

```
        "instructions": (
            "1. Run grep command above\n"
            "2. Verify ONLY factory.py calls load_questionnaire()\n"
            "3. Remove any other calls found\n"
            "4. Update tests to use AnalysisPipelineFactory"
        ),
    }


def get_method_dispensary_info() -> dict[str, Any]:
    """Get information about the method dispensary pattern.

    Returns detailed statistics about:
    - Which monolith classes serve as dispensaries
    - How many methods each dispensary provides
    - Which executors use which dispensaries
    - Method reuse patterns

    Returns:
        dict with dispensary statistics and usage patterns.
    """
    from farfan_pipeline.core.orchestrator.class_registry import get_class_paths

    class_paths = get_class_paths()

    # Load executorâ\206\222methods mapping
    try:
        import json
        from pathlib import Path
        executors_methods_path = Path(__file__).parent / "executors_methods.json"
        if executors_methods_path.exists():
            with open(executors_methods_path) as f:
                executors_methods = json.load(f)
        else:
            executors_methods = []
    except Exception:
        executors_methods = []

    # Build dispensary statistics
    dispensaries = {}
    for class_name in class_paths.keys():
        dispensaries[class_name] = {
            "module": class_paths[class_name],
            "methods_provided": [],
            "used_by_executors": [],
            "total_usage_count": 0,
        }

    # Count method usage per dispensary
    for executor_info in executors_methods:
        executor_id = executor_info.get("executor_id")
        methods = executor_info.get("methods", [])

        for method_info in methods:
            class_name = method_info.get("class")
            method_name = method_info.get("method")
```

```
            if class_name in dispensaries:
                if method_name not in dispensaries[class_name]["methods_provided"]:
                    dispensaries[class_name]["methods_provided"].append(method_name)

                if executor_id not in dispensaries[class_name]["used_by_executors"]:
                    dispensaries[class_name]["used_by_executors"].append(executor_id)

                dispensaries[class_name]["total_usage_count"] += 1

    # Sort by usage count
    sorted_dispensaries = sorted(
        dispensaries.items(),
        key=lambda x: x[1]["total_usage_count"],
        reverse=True
    )

    # Build summary statistics
    total_methods = sum(len(d["methods_provided"]) for _, d in sorted_dispensaries)
    total_usage = sum(d["total_usage_count"] for _, d in sorted_dispensaries)

    return {
        "pattern": "method_dispensary",
        "description": "Monolith classes serve as method dispensaries for 30 executors",
        "total_dispensaries": len(dispensaries),
        "total_unique_methods": total_methods,
        "total_method_calls": total_usage,
        "avg_reuse_per_method": round(total_usage / max(total_methods, 1), 2),
        "dispensaries": {
            name: {
                "methods_count": len(info["methods_provided"]),
                "executor_count": len(info["used_by_executors"]),
                "total_calls": info["total_usage_count"],
                "reuse_factor": round(info["total_usage_count"] / max(len(info["methods_provided"]), 1), 2),
            }
            for name, info in sorted_dispensaries[:10]  # Top 10
        },
        "top_dispensaries": [
            {
                "class": name,
                "methods": len(info["methods_provided"]),
                "executors": len(info["used_by_executors"]),
                "calls": info["total_usage_count"],
            }
            for name, info in sorted_dispensaries[:5]
        ],
    }


def validate_method_dispensary_pattern() -> dict[str, Any]:
    """Validate that the method dispensary pattern is correctly implemented.

    Checks:
    1. All executor methods exist in class_registry
    2. No executor directly imports monolith classes
    3. All methods route through MethodExecutor
```

```
    4. Signal registry is injected (not globally accessed)

    Returns:
        dict with validation results.
    """
    from farfan_pipeline.core.orchestrator.class_registry import get_class_paths

    class_paths = get_class_paths()
    validation_results = {
        "pattern_valid": True,
        "errors": [],
        "warnings": [],
        "checks": {},
    }

    # Check 1: Verify class_registry is populated
    if not class_paths:
        validation_results["pattern_valid"] = False
        validation_results["errors"].append(
            "class_registry is empty - no dispensaries registered"
        )
    else:
        validation_results["checks"]["dispensaries_registered"] = len(class_paths)

    # Check 2: Verify executor_methods.json exists
    try:
        import json
        from pathlib import Path
        executors_methods_path = Path(__file__).parent / "executors_methods.json"
        if not executors_methods_path.exists():
            validation_results["warnings"].append(
                "executors_methods.json not found - cannot validate method mappings"
            )
        else:
            with open(executors_methods_path) as f:
                executors_methods = json.load(f)
            validation_results["checks"]["executor_method_mappings"] = len(executors_methods)
    except Exception as e:
        validation_results["warnings"].append(
            f"Failed to load executors_methods.json: {e}"
        )

    # Check 3: Verify validation file exists
    try:
        validation_path = Path(__file__).parent / "executor_factory_validation.json"
        if not validation_path.exists():
            validation_results["warnings"].append(
                "executor_factory_validation.json not found - cannot validate method catalog"
            )
        else:
            with open(validation_path) as f:
                validation_data = json.load(f)
            validation_results["checks"]["method_pairs_validated"] = validation_data.get("validated_against_catalog", 0)
            validation_results["checks"]["validation_failures"] = len(validation_data.get("failures", []))
    except Exception as e:
        validation_results["warnings"].append(
```

```
                f"Failed to load executor_factory_validation.json: {e}"
            )

    return validation_results



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/irrigation_synchronizer.py
================================================================================

"""Irrigation Synchronizer - Questionâ\206\222Chunkâ\206\222Taskâ\206\222Plan Coordination.

This module implements the synchronization layer that maps questionnaire questions
to document chunks, generating an ExecutionPlan with 300 tasks (6 dimensions Ã\227 50
questions/dimension Ã\227 10 policy areas) for deterministic pipeline execution.

Architecture:
- IrrigationSynchronizer: Orchestrates chunkâ\206\222questionâ\206\222taskâ\206\222plan flow
- ExecutionPlan: Immutable plan with deterministic plan_id and integrity_hash
- Task: Single unit of work (question + chunk + policy_area)
- Observability: Structured JSON logs with correlation_id tracking

Design Principles:
- Deterministic task generation (stable ordering, reproducible plan_id)
- Full observability (correlation_id propagates through all 10 phases)
- Prometheus metrics for synchronization health
- Blake3-based integrity hashing for plan verification
"""

from __future__ import annotations

import hashlib
import json
import logging
import statistics
import time
import uuid
from collections import Counter
from dataclasses import dataclass, field
from pathlib import Path
from typing import TYPE_CHECKING, Any, Protocol

if TYPE_CHECKING:
    from farfan_pipeline.core.orchestrator.signals import SignalRegistry

from farfan_pipeline.core.orchestrator.task_planner import ExecutableTask
from farfan_pipeline.core.orchestrator.phase6_validation import (
    validate_phase6_schema_compatibility,
)
from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
from farfan_pipeline.synchronization import ChunkMatrix

try:
    from farfan_pipeline.core.orchestrator.signals import (
        SignalRegistry as _SignalRegistry,
```

```
    )
except ImportError:
    _SignalRegistry = None  # type: ignore

try:
    import blake3

    BLAKE3_AVAILABLE = True
except ImportError:
    BLAKE3_AVAILABLE = False

try:
    from prometheus_client import Counter, Histogram

    PROMETHEUS_AVAILABLE = True
except ImportError:
    PROMETHEUS_AVAILABLE = False

logger = logging.getLogger(__name__)

SHA256_HEX_DIGEST_LENGTH = 64

SKEW_THRESHOLD_CV = 0.3

class SignalRegistry(Protocol):
    """Protocol for signal registry implementations.

    Defines the interface that signal registries must implement for
    use with IrrigationSynchronizer signal resolution.
    """

    def get_signals_for_chunk(
        self, chunk: ChunkData, requirements: list[str]
    ) -> list[Any]:
        """Get signals for a chunk matching the given requirements.

        Args:
            chunk: Target chunk to get signals for
            requirements: List of required signal types

        Returns:
            List of signals, each with signal_id, signal_type, and content fields
        """
        ...


if PROMETHEUS_AVAILABLE:
    synchronization_duration = Histogram(
        "synchronization_duration_seconds",
        "Time spent building execution plan",
        buckets=[0.1, 0.5, 1.0, 2.0, 5.0, 10.0],
    )
    tasks_constructed = Counter(
        "synchronization_tasks_constructed_total",
        "Total number of tasks constructed",
        ["dimension", "policy_area"],
```

```
    )
    synchronization_failures = Counter(
        "synchronization_failures_total",
        "Total synchronization failures",
        ["error_type"],
    )
    synchronization_chunk_matches = Counter(
        "synchronization_chunk_matches_total",
        "Total chunk routing matches during synchronization",
        ["dimension", "policy_area", "status"],
    )
else:

    class DummyMetric:
        def time(self):
            class DummyContextManager:
                def __call__(self, func):
                    def wrapper(*args, **kwargs):
                        return func(*args, **kwargs)

                    return wrapper

                def __enter__(self):
                    return self

                def __exit__(self, *args):
                    pass

            return DummyContextManager()

        def labels(self, **kwargs):
            return self

        def inc(self, *args, **kwargs) -> None:
            pass

    synchronization_duration = DummyMetric()
    tasks_constructed = DummyMetric()
    synchronization_failures = DummyMetric()
    synchronization_chunk_matches = DummyMetric()

SHA256_HEX_DIGEST_LENGTH = 64


@dataclass(frozen=True)
class ChunkRoutingResult:
    """Result of Phase 3 chunk routing verification.

    Contains validated chunk reference and extracted metadata for task construction.
    """

    target_chunk: ChunkData
    chunk_id: str
    policy_area_id: str
    dimension_id: str
    text_content: str
```

```
    expected_elements: list[dict[str, Any]]
    document_position: tuple[int, int] | None


@dataclass(frozen=True)
class Task:
    """Single unit of work in the execution plan.

    Represents the mapping of one question to one chunk in a specific policy area.
    """

    task_id: str
    dimension: str
    question_id: str
    policy_area: str
    chunk_id: str
    chunk_index: int
    question_text: str


@dataclass
class ExecutionPlan:
    """Immutable execution plan with deterministic identifiers.

    Contains all tasks to be executed, with cryptographic integrity verification.
    """

    plan_id: str
    tasks: tuple[Task, ...]
    chunk_count: int
    question_count: int
    integrity_hash: str
    created_at: str
    correlation_id: str
    metadata: dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> dict[str, Any]:
        """Convert plan to dictionary for serialization."""
        return {
            "plan_id": self.plan_id,
            "tasks": [
                {
                    "task_id": t.task_id,
                    "dimension": t.dimension,
                    "question_id": t.question_id,
                    "policy_area": t.policy_area,
                    "chunk_id": t.chunk_id,
                    "chunk_index": t.chunk_index,
                    "question_text": t.question_text,
                }
                for t in self.tasks
            ],
            "chunk_count": self.chunk_count,
            "question_count": self.question_count,
            "integrity_hash": self.integrity_hash,
            "created_at": self.created_at,
```

```
            "correlation_id": self.correlation_id,
            "metadata": self.metadata,
        }

    @classmethod
    def from_dict(cls, data: dict[str, Any]) -> ExecutionPlan:
        """Reconstruct ExecutionPlan from dictionary.

        Args:
            data: Dictionary representation of ExecutionPlan

        Returns:
            ExecutionPlan instance reconstructed from dictionary
        """
        tasks = tuple(
            Task(
                task_id=t["task_id"],
                dimension=t["dimension"],
                question_id=t["question_id"],
                policy_area=t["policy_area"],
                chunk_id=t["chunk_id"],
                chunk_index=t["chunk_index"],
                question_text=t["question_text"],
            )
            for t in data["tasks"]
        )

        return cls(
            plan_id=data["plan_id"],
            tasks=tasks,
            chunk_count=data["chunk_count"],
            question_count=data["question_count"],
            integrity_hash=data["integrity_hash"],
            created_at=data["created_at"],
            correlation_id=data["correlation_id"],
        )


class IrrigationSynchronizer:
    """Synchronizes questionnaire questions with document chunks.

    Generates deterministic execution plans mapping questions to chunks across
    all policy areas, with full observability and integrity verification.
    """

    def __init__(
        self,
        questionnaire: dict[str, Any],
        preprocessed_document: PreprocessedDocument | None = None,
        document_chunks: list[dict[str, Any]] | None = None,
        signal_registry: SignalRegistry | None = None,
    ) -> None:
        """Initialize synchronizer with questionnaire and chunks.

        Args:
            questionnaire: Loaded questionnaire_monolith.json data
```

```
            preprocessed_document: PreprocessedDocument containing validated chunks
            document_chunks: Legacy list of document chunks (deprecated)
            signal_registry: SignalRegistry for Phase 5 signal resolution (initialized if None)

        Raises:
            ValueError: If chunk matrix validation fails or no chunks provided
        """
        self.questionnaire = questionnaire
        self.correlation_id = str(uuid.uuid4())
        self.question_count = self._count_questions()
        self.chunk_matrix: ChunkMatrix | None = None
        self.document_chunks: list[dict[str, Any]] | None = None

        if signal_registry is None and _SignalRegistry is not None:
            self.signal_registry: SignalRegistry | None = _SignalRegistry()
        else:
            self.signal_registry = signal_registry

        if preprocessed_document is not None:
            try:
                self.chunk_matrix = ChunkMatrix(preprocessed_document)
                self.chunk_count = ChunkMatrix.EXPECTED_CHUNK_COUNT

                logger.info(
                    json.dumps(
                        {
                            "event": "irrigation_synchronizer_init",
                            "correlation_id": self.correlation_id,
                            "question_count": self.question_count,
                            "chunk_count": self.chunk_count,
                            "chunk_matrix_validated": True,
                            "mode": "preprocessed_document",
                            "timestamp": time.time(),
                        }
                    )
                )
            except ValueError as e:
                synchronization_failures.labels(
                    error_type="chunk_matrix_validation"
                ).inc()
                logger.error(
                    json.dumps(
                        {
                            "event": "irrigation_synchronizer_init_failed",
                            "correlation_id": self.correlation_id,
                            "error": str(e),
                            "error_type": "chunk_matrix_validation",
                            "timestamp": time.time(),
                        }
                    )
                )
                raise ValueError(
                    f"Chunk matrix validation failed during synchronizer initialization: {e}"
                ) from e
        elif document_chunks is not None:
            self.document_chunks = document_chunks
```

```
            self.chunk_count = len(document_chunks)

            logger.info(
                json.dumps(
                    {
                        "event": "irrigation_synchronizer_init",
                        "correlation_id": self.correlation_id,
                        "question_count": self.question_count,
                        "chunk_count": self.chunk_count,
                        "mode": "legacy_document_chunks",
                        "timestamp": time.time(),
                    }
                )
            )
        else:
            raise ValueError(
                "Either preprocessed_document or document_chunks must be provided"
            )

    def _count_questions(self) -> int:
        """Count total questions across all dimensions."""
        count = 0
        blocks = self.questionnaire.get("blocks", {})

        for dimension_key in ["D1", "D2", "D3", "D4", "D5", "D6"]:
            for i in range(1, 51):
                question_key = f"D{dimension_key[1]}_Q{i:02d}"
                if question_key in blocks:
                    count += 1

        return count

    def validate_chunk_routing(self, question: dict[str, Any]) -> ChunkRoutingResult:
        """Phase 3: Validate chunk routing and extract metadata.

        Verifies that a chunk exists in the matrix for the question’s routing keys,
        validates chunk consistency, and extracts metadata for task construction.

        Args:
            question: Question dict with routing keys (policy_area_id, dimension_id)

        Returns:
            ChunkRoutingResult with validated chunk and extracted metadata

        Raises:
            ValueError: If chunk not found or validation fails
        """
        question_id = question.get("question_id", "UNKNOWN")
        policy_area_id = question.get("policy_area_id")
        dimension_id = question.get("dimension_id")

        if not policy_area_id:
            raise ValueError(
                f"Question {question_id} missing required field: policy_area_id"
            )

```

```
        if not dimension_id:
            raise ValueError(
                f"Question {question_id} missing required field: dimension_id"
            )

        try:
            target_chunk = self.chunk_matrix.get_chunk(policy_area_id, dimension_id)

            chunk_id = target_chunk.chunk_id or f"{policy_area_id}-{dimension_id}"

            if not target_chunk.text or not target_chunk.text.strip():
                raise ValueError(
                    f"Chunk {chunk_id} has empty text content for question {question_id}"
                )

            if (
                target_chunk.policy_area_id
                and target_chunk.policy_area_id != policy_area_id
            ):
                raise ValueError(
                    f"Chunk routing key mismatch for {question_id}: "
                    f"question policy_area={policy_area_id} but chunk has {target_chunk.policy_area_id}"
                )

            if target_chunk.dimension_id and target_chunk.dimension_id != dimension_id:
                raise ValueError(
                    f"Chunk routing key mismatch for {question_id}: "
                    f"question dimension={dimension_id} but chunk has {target_chunk.dimension_id}"
                )

            expected_elements = question.get("expected_elements", [])

            document_position = None
            if target_chunk.start_pos is not None and target_chunk.end_pos is not None:
                document_position = (target_chunk.start_pos, target_chunk.end_pos)

            synchronization_chunk_matches.labels(
                dimension=dimension_id, policy_area=policy_area_id, status="success"
            ).inc()

            logger.debug(
                json.dumps(
                    {
                        "event": "chunk_routing_success",
                        "question_id": question_id,
                        "chunk_id": chunk_id,
                        "policy_area_id": policy_area_id,
                        "dimension_id": dimension_id,
                        "text_length": len(target_chunk.text),
                        "has_expected_elements": len(expected_elements) > 0,
                        "has_document_position": document_position is not None,
                        "correlation_id": self.correlation_id,
                    }
                )
            )

```

```
            return ChunkRoutingResult(
                target_chunk=target_chunk,
                chunk_id=chunk_id,
                policy_area_id=policy_area_id,
                dimension_id=dimension_id,
                text_content=target_chunk.text,
                expected_elements=expected_elements,
                document_position=document_position,
            )

        except KeyError as e:
            synchronization_chunk_matches.labels(
                dimension=dimension_id, policy_area=policy_area_id, status="failure"
            ).inc()

            error_msg = (
                f"Synchronization Failure for MQC {question_id}: "
                f"PA={policy_area_id}, DIM={dimension_id}. "
                f"No corresponding chunk found in matrix."
            )

            logger.error(
                json.dumps(
                    {
                        "event": "chunk_routing_failure",
                        "question_id": question_id,
                        "policy_area_id": policy_area_id,
                        "dimension_id": dimension_id,
                        "error": error_msg,
                        "correlation_id": self.correlation_id,
                    }
                )
            )

            raise ValueError(error_msg) from e

    def _extract_questions(self) -> list[dict[str, Any]]:
        """Extract all questions from questionnaire in deterministic order."""
        questions = []
        blocks = self.questionnaire.get("blocks", {})

        for dimension in range(1, 7):
            dim_key = f"D{dimension}"
            dimension_id = f"DIM{dimension:02d}"

            for q_num in range(1, 51):
                question_key = f"{dim_key}_Q{q_num:02d}"

                if question_key in blocks:
                    block = blocks[question_key]
                    questions.append(
                        {
                            "dimension": dim_key,
                            "question_id": question_key,
                            "question_num": q_num,
                            "question_global": block.get("question_global", 0),
```

```
                            "question_text": block.get("question", ""),
                            "policy_area_id": block.get("policy_area_id"),
                            "dimension_id": dimension_id,
                            "patterns": block.get("patterns", []),
                            "expected_elements": block.get("expected_elements", []),
                            "signal_requirements": block.get("signal_requirements", {}),
                        }
                    )

        questions.sort(key=lambda q: (q["dimension_id"], q["question_id"]))

        return questions

    def _filter_patterns(
        self,
        patterns: list[dict[str, Any]] | tuple[dict[str, Any], ...],
        policy_area_id: str,
    ) -> tuple[dict[str, Any], ...]:
        """Filter patterns by policy_area_id using strict equality.

        Filters patterns to include only those where pattern.policy_area_id == policy_area_id
        (strict equality). Patterns lacking a policy_area_id attribute are excluded.

        Args:
            patterns: Iterable of pattern objects (typically dicts with optional policy_area_id)
            policy_area_id: Policy area ID string (e.g., "PA01") to filter by

        Returns:
            Immutable tuple of filtered pattern dicts. Returns empty tuple if no patterns match.

        Filtering Rules:
            - Strict equality: pattern.policy_area_id == policy_area_id
            - Exclude patterns without policy_area_id attribute
            - Result is immutable (tuple)
        """
        included = []
        excluded = []
        included_ids = []
        excluded_ids = []

        for pattern in patterns:
            pattern_id = (
                pattern.get("id", "UNKNOWN") if isinstance(pattern, dict) else "UNKNOWN"
            )

            if isinstance(pattern, dict) and "policy_area_id" in pattern:
                if pattern["policy_area_id"] == policy_area_id:
                    included.append(pattern)
                    included_ids.append(pattern_id)
                else:
                    excluded.append(pattern)
                    excluded_ids.append(pattern_id)
            else:
                excluded.append(pattern)
                excluded_ids.append(pattern_id)

```

```
        total_count = len(included) + len(excluded)

        logger.info(
            json.dumps(
                {
                    "event": "IrrigationSynchronizer._filter_patterns",
                    "total": total_count,
                    "included": len(included),
                    "excluded": len(excluded),
                    "included_ids": included_ids,
                    "excluded_ids": excluded_ids,
                    "policy_area_id": policy_area_id,
                    "correlation_id": self.correlation_id,
                }
            )
        )

        return tuple(included)

    def _construct_task(
        self,
        question: dict[str, Any],
        routing_result: ChunkRoutingResult,
        applicable_patterns: tuple[dict[str, Any], ...],
        resolved_signals: tuple[Any, ...],
        generated_task_ids: set[str],
    ) -> ExecutableTask:
        """Construct ExecutableTask from question and routing result.

        Extracts all fields from validated inputs, converts tuples to lists for patterns,
        builds signals dict keyed by signal_type, generates creation_timestamp, populates
        metadata with all required keys, validates all mandatory fields are non-None, and
        catches TypeError from dataclass validation to re-raise as ValueError.

        Args:
            question: Question dict from questionnaire
            routing_result: Validated routing result from Phase 3
            applicable_patterns: Filtered tuple of patterns applicable to the routed policy area
            resolved_signals: Resolved signals tuple from Phase 5
            generated_task_ids: Set of task IDs generated in current synchronization run

        Returns:
            ExecutableTask ready for execution

        Raises:
            ValueError: If duplicate task_id is detected or required fields are missing/invalid
        """
        # Phase 7.1: Validate and extract question_global
        question_global = question.get("question_global")
        if question_global is None:
            raise ValueError("question_global field is required but missing")
        if not isinstance(question_global, int):
            raise ValueError(
                f"question_global must be an integer, got {type(question_global).__name__}"
            )

```

```
        # Phase 7.1: Construct task_id from validated question_global
        task_id = f"MQC-{question_global:03d}_{routing_result.policy_area_id}"

        if task_id in generated_task_ids:
            raise ValueError(f"Duplicate task_id detected: {task_id}")

        generated_task_ids.add(task_id)

        # Field extraction in declaration order for validation priority
        # Extract question_id with bracket notation and KeyError conversion
        try:
            question_id = question["question_id"]
        except KeyError as e:
            raise ValueError("question_id field is required but missing") from e

        # Assign question_global (already validated above)
        # Extract routing fields via attribute access (guaranteed by ChunkRoutingResult schema)
        policy_area_id = routing_result.policy_area_id
        dimension_id = routing_result.dimension_id
        chunk_id = routing_result.chunk_id

        expected_elements_list = list(routing_result.expected_elements)
        document_position = routing_result.document_position

        patterns_list = list(applicable_patterns)

        signals_dict: dict[str, Any] = {}
        for signal in resolved_signals:
            if isinstance(signal, dict) and "signal_type" in signal:
                signals_dict[signal["signal_type"]] = signal
            elif hasattr(signal, "signal_type"):
                signals_dict[signal.signal_type] = signal

        from datetime import datetime, timezone

        creation_timestamp = datetime.now(timezone.utc).isoformat()

        metadata = {
            "document_position": document_position,
            "synchronizer_version": "1.0.0",
            "correlation_id": self.correlation_id,
            "original_pattern_count": len(applicable_patterns),
            "original_signal_count": len(resolved_signals),
        }

        if task_id is None or not task_id:
            raise ValueError("Task construction failure: task_id is None or empty")
        if question_id is None or not question_id:
            raise ValueError("Task construction failure: question_id is None or empty")
        if question_global is None:
            raise ValueError("Task construction failure: question_global is None")
        if policy_area_id is None or not policy_area_id:
            raise ValueError(
                "Task construction failure: policy_area_id is None or empty"
            )
        if dimension_id is None or not dimension_id:
```

```
            raise ValueError("Task construction failure: dimension_id is None or empty")
        if chunk_id is None or not chunk_id:
            raise ValueError("Task construction failure: chunk_id is None or empty")
        if creation_timestamp is None or not creation_timestamp:
            raise ValueError(
                "Task construction failure: creation_timestamp is None or empty"
            )

        try:
            task = ExecutableTask(
                task_id=task_id,
                question_id=question_id,
                question_global=question_global,
                policy_area_id=policy_area_id,
                dimension_id=dimension_id,
                chunk_id=chunk_id,
                patterns=patterns_list,
                signals=signals_dict,
                creation_timestamp=creation_timestamp,
                expected_elements=expected_elements_list,
                metadata=metadata,
            )
        except TypeError as e:
            raise ValueError(
                f"Task construction failed for {task_id}: dataclass validation error - {e}"
            ) from e

        logger.debug(
            f"Constructed task: task_id={task_id}, question_id={question_id}, "
            f"chunk_id={chunk_id}, pattern_count={len(patterns_list)}, "
            f"signal_count={len(signals_dict)}"
        )

        return task

    def _assemble_execution_plan(
        self,
        executable_tasks: list[ExecutableTask],
        questions: list[dict[str, Any]],
        correlation_id: str,  # noqa: ARG002
    ) -> tuple[list[ExecutableTask], str]:
        """Phase 8: Assemble execution plan with validation and deterministic ordering.

        Performs four-phase assembly process:
        - Phase 8.1: Pre-assembly validation (duplicate detection, count validation)
        - Phase 8.2: Deterministic task ordering (lexicographic by task_id)
        - Phase 8.3: Plan identifier computation (SHA256 of deterministic JSON)
        - Phase 8.4: Plan identifier validation (format and length checks)

        Validates that task count matches question count and that no duplicate
        task identifiers exist. Then sorts tasks lexicographically by task_id to ensure
        deterministic plan identifier generation across runs. Computes plan_id by
        encoding deterministic JSON serialization (sort_keys=True, compact separators)
        to UTF-8 bytes, computing SHA256 hash, and validating result matches expected
        64-character lowercase hexadecimal format.

```

```
        Args:
            executable_tasks: List of constructed ExecutableTask objects
            questions: List of question dictionaries
            correlation_id: Correlation ID for tracing

        Returns:
            Tuple of (sorted list of ExecutableTask objects, plan_id string)

        Raises:
            ValueError: If task count doesn't match question count, duplicates exist,
                        or plan_id validation fails
            RuntimeError: When sorting operation corrupts task list length
        """
        from collections import Counter

        question_count = len(questions)
        task_count = len(executable_tasks)

        if task_count != question_count:
            raise ValueError(
                f"Execution plan assembly failure: expected {question_count} tasks "
                f"but constructed {task_count}; task construction loop corrupted"
            )

        task_ids = [t.task_id for t in executable_tasks]
        unique_count = len(set(task_ids))

        if unique_count != len(task_ids):
            counter = Counter(task_ids)
            duplicates = [task_id for task_id, count in counter.items() if count > 1]
            duplicate_count = len(task_ids) - unique_count

            raise ValueError(
                f"Execution plan assembly failure: found {duplicate_count} duplicate "
                f"task identifiers; duplicates are {sorted(duplicates)}"
            )

        sorted_tasks = sorted(executable_tasks, key=lambda t: t.task_id)

        if len(sorted_tasks) != len(executable_tasks):
            raise RuntimeError(
                f"Task ordering corruption detected: sorted task count {len(sorted_tasks)} "
                f"does not match input task count {len(executable_tasks)}"
            )

        task_serialization = [
            {
                "task_id": t.task_id,
                "question_id": t.question_id,
                "question_global": t.question_global,
                "policy_area_id": t.policy_area_id,
                "dimension_id": t.dimension_id,
                "chunk_id": t.chunk_id,
            }
            for t in sorted_tasks
        ]
```

```

        json_bytes = json.dumps(
            task_serialization, sort_keys=True, separators=(",", ":")
        ).encode("utf-8")

        plan_id = hashlib.sha256(json_bytes).hexdigest()

        if len(plan_id) != SHA256_HEX_DIGEST_LENGTH:
            raise ValueError(
                f"Plan identifier validation failure: expected length {SHA256_HEX_DIGEST_LENGTH} but got {len(plan_id)}; "
                "SHA256 implementation may be compromised or monkey-patched"
            )

        if not all(c in "0123456789abcdef" for c in plan_id):
            raise ValueError(
                "Plan identifier validation failure: expected lowercase hexadecimal but got "
                "characters outside '0123456789abcdef' set; SHA256 implementation may be "
                "compromised or monkey-patched"
            )

        return sorted_tasks, plan_id

    def _compute_integrity_hash(self, tasks: list[Task]) -> str:
        """Compute Blake3 or SHA256 integrity hash of execution plan."""
        task_data = json.dumps(
            [
                {
                    "task_id": t.task_id,
                    "dimension": t.dimension,
                    "question_id": t.question_id,
                    "policy_area": t.policy_area,
                    "chunk_id": t.chunk_id,
                }
                for t in tasks
            ],
            sort_keys=True,
        ).encode("utf-8")

        if BLAKE3_AVAILABLE:
            return blake3.blake3(task_data).hexdigest()
        else:
            return hashlib.sha256(task_data).hexdigest()

    def _construct_execution_plan_phase_8_4(
        self,
        sorted_tasks: list[Task],
        plan_id: str,
        chunk_count: int,
        question_count: int,
        integrity_hash: str,
    ) -> ExecutionPlan:
        """Phase 8.4: ExecutionPlan dataclass construction.

        Constructs the final execution artifact from the sorted task list produced in
        Phase 8.2, converting sorted_tasks to an immutable tuple, constructing a
        metadata dictionary with generation_timestamp (UTC ISO 8601),
```

```
        synchronizer_version "2.0.0", chunk_count from the chunk matrix,
        question_count and task_count, invoking the ExecutionPlan constructor with
        plan_id from Phase 8.3 and tasks_tuple with metadata_dict as keyword arguments,
        wrapping the constructor call in try-except to catch TypeError from dataclass
        validation and re-raise as ValueError with context-specific message, then
        verifying task order preservation by checking that all adjacent task_id pairs
        maintain lexicographic ordering and raising ValueError if any violation is
        detected before emitting an info-level structured log event and returning the
        constructed ExecutionPlan instance.

        Args:
            sorted_tasks: List of Task objects sorted by task_id (from Phase 8.2)
            plan_id: Plan identifier string (from Phase 8.3)
            chunk_count: Number of chunks in the document
            question_count: Number of questions in the questionnaire
            integrity_hash: Blake3 or SHA256 hash of the task list

        Returns:
            ExecutionPlan instance with validated task ordering

        Raises:
            ValueError: If dataclass validation fails or task ordering is violated
        """
        tasks_tuple = tuple(sorted_tasks)

        metadata_dict = {
            "generation_timestamp": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
            "synchronizer_version": "2.0.0",
            "chunk_count": chunk_count,
            "question_count": question_count,
            "task_count": len(tasks_tuple),
        }

        try:
            plan = ExecutionPlan(
                plan_id=plan_id,
                tasks=tasks_tuple,
                chunk_count=metadata_dict["chunk_count"],
                question_count=metadata_dict["question_count"],
                integrity_hash=integrity_hash,
                created_at=metadata_dict["generation_timestamp"],
                correlation_id=self.correlation_id,
            )
        except TypeError as e:
            raise ValueError(
                f"ExecutionPlan dataclass construction failed: {e}. "
                f"Constructor validation rejected arguments (plan_id={plan_id}, "
                f"task_count={len(tasks_tuple)}, chunk_count={chunk_count}, "
                f"question_count={question_count})"
            ) from e

        for i in range(len(tasks_tuple) - 1):
            current_task_id = tasks_tuple[i].task_id
            next_task_id = tasks_tuple[i + 1].task_id

            if current_task_id >= next_task_id:
```

```
                raise ValueError(
                    f"Task order preservation violation detected at index {i}: "
                    f"task_id '{current_task_id}' >= task_id '{next_task_id}'. "
                    f"Expected strict lexicographic ordering maintained after Phase 8.2 sort."
                )

        logger.info(
            json.dumps(
                {
                    "event": "execution_plan_phase_8_4_complete",
                    "plan_id": plan_id,
                    "task_count": len(tasks_tuple),
                    "chunk_count": chunk_count,
                    "question_count": question_count,
                    "integrity_hash": integrity_hash,
                    "synchronizer_version": metadata_dict["synchronizer_version"],
                    "generation_timestamp": metadata_dict["generation_timestamp"],
                    "correlation_id": self.correlation_id,
                    "phase": "execution_plan_construction_phase_8_4",
                }
            )
        )

        return plan

    def _validate_cross_task_cardinality(
        self, plan: ExecutionPlan, questions: list[dict[str, Any]]
    ) -> None:
        """Validate cross-task cardinality and log task distribution statistics.

        Extracts unique chunk IDs from execution plan tasks, computes expected
        reference counts by filtering questions for matching policy_area_id and
        dimension_id (parsed from chunk_id), compares actual task counts per chunk
        against expected counts, and emits warning-level logs for mismatches.

        Also collects chunk usage statistics (mean, median, min, max) across all
        unique chunks, policy area task distribution mapping, and dimension coverage
        validation, culminating in a single info-level log entry with complete
        observability into task distribution patterns.

        Args:
            plan: ExecutionPlan containing all constructed tasks
            questions: List of original question dictionaries

        Raises:
            None - Discrepancies emit warnings but do not raise exceptions since
                   they may reflect legitimate sparse coverage rather than errors
        """
        unique_chunks: set[str] = set()
        chunk_task_counts: dict[str, int] = {}

        for task in plan.tasks:
            chunk_id = task.chunk_id
            unique_chunks.add(chunk_id)
            chunk_task_counts[chunk_id] = chunk_task_counts.get(chunk_id, 0) + 1

```

```
        for chunk_id, actual_count in chunk_task_counts.items():
            try:
                parts = chunk_id.split("-")
                if len(parts) >= 2:
                    policy_area_id = parts[0]
                    dimension_id = parts[1]

                    expected_count = sum(
                        1
                        for q in questions
                        if q.get("policy_area_id") == policy_area_id
                        and q.get("dimension_id") == dimension_id
                    )

                    if actual_count != expected_count:
                        logger.warning(
                            json.dumps(
                                {
                                    "event": "cross_task_cardinality_mismatch",
                                    "chunk_id": chunk_id,
                                    "policy_area_id": policy_area_id,
                                    "dimension_id": dimension_id,
                                    "expected_count": expected_count,
                                    "actual_count": actual_count,
                                    "correlation_id": self.correlation_id,
                                    "timestamp": time.time(),
                                }
                            )
                        )
            except (IndexError, ValueError) as e:
                logger.warning(
                    json.dumps(
                        {
                            "event": "chunk_id_parse_error",
                            "chunk_id": chunk_id,
                            "error": str(e),
                            "correlation_id": self.correlation_id,
                            "timestamp": time.time(),
                        }
                    )
                )

        chunk_counts = list(chunk_task_counts.values())
        chunk_usage_stats: dict[str, float] = {}

        if chunk_counts:
            chunk_usage_stats = {
                "mean": statistics.mean(chunk_counts),
                "median": statistics.median(chunk_counts),
                "min": float(min(chunk_counts)),
                "max": float(max(chunk_counts)),
            }

        tasks_per_policy_area: dict[str, int] = {}
        for task in plan.tasks:
            try:
```

```
                parts = task.chunk_id.split("-")
                if len(parts) >= 1:
                    policy_area_id = parts[0]
                    tasks_per_policy_area[policy_area_id] = (
                        tasks_per_policy_area.get(policy_area_id, 0) + 1
                    )
            except (IndexError, ValueError):
                pass

        tasks_per_dimension: dict[str, int] = {}
        for task in plan.tasks:
            try:
                parts = task.chunk_id.split("-")
                if len(parts) >= 2:
                    dimension_id = parts[1]
                    tasks_per_dimension[dimension_id] = (
                        tasks_per_dimension.get(dimension_id, 0) + 1
                    )
            except (IndexError, ValueError):
                pass

        logger.info(
            json.dumps(
                {
                    "event": "cross_task_cardinality_validation_complete",
                    "total_unique_chunks": len(unique_chunks),
                    "tasks_per_policy_area": tasks_per_policy_area,
                    "tasks_per_dimension": tasks_per_dimension,
                    "chunk_usage_stats": chunk_usage_stats,
                    "correlation_id": self.correlation_id,
                    "timestamp": time.time(),
                }
            )
        )

    @synchronization_duration.time()
    def build_execution_plan(self) -> ExecutionPlan:
        """Build deterministic execution plan mapping questions to chunks.

        Uses validated chunk matrix if available, otherwise falls back to
        legacy document_chunks iteration mode.

        Returns:
            ExecutionPlan with deterministic plan_id and integrity_hash

        Raises:
            ValueError: If question data is invalid or chunk matrix lookup fails
        """
        if self.chunk_matrix is not None:
            return self._build_with_chunk_matrix()
        else:
            return self._build_with_legacy_chunks()

    def _build_with_chunk_matrix(self) -> ExecutionPlan:
        """Build execution plan using validated chunk matrix.

```

```
        Orchestrates Phases 2-7 of irrigation synchronization:
        - Phase 2: Question extraction
        - Phase 3: Chunk routing (OBJECTIVE 3 INTEGRATION)
        - Phase 4: Pattern filtering (policy_area_id-based filtering)
        - Phase 5: Signal resolution (future)
        - Phase 6: Schema validation (future)
        - Phase 7: Task construction

        Returns:
            ExecutionPlan with validated tasks

        Raises:
            ValueError: On routing failures, validation errors
        """
        logger.info(
            json.dumps(
                {
                    "event": "task_construction_start",
                    "correlation_id": self.correlation_id,
                    "question_count": self.question_count,
                    "chunk_count": self.chunk_count,
                    "mode": "chunk_matrix",
                    "phase": "synchronization_phase_2",
                    "timestamp": time.time(),
                }
            )
        )

        try:
            if self.question_count == 0:
                synchronization_failures.labels(error_type="empty_questions").inc()
                raise ValueError(
                    "No questions extracted from questionnaire. "
                    "Cannot build tasks with empty question set."
                )

            questions = self._extract_questions()

            if not questions:
                raise ValueError(
                    "No questions extracted from questionnaire. "
                    "Cannot build tasks with empty question set."
                )

            tasks: list[ExecutableTask] = []
            routing_successes = 0
            routing_failures = 0
            generated_task_ids: set[str] = set()

            for idx, question in enumerate(questions, start=1):
                question_id = question.get("question_id", f"UNKNOWN_{idx}")
                policy_area_id = question.get("policy_area_id", "UNKNOWN")
                dimension_id = question.get("dimension_id", "UNKNOWN")
                chunk_id = "UNKNOWN"

                try:
```

```
                routing_result = self.validate_chunk_routing(question)
                routing_successes += 1
                chunk_id = routing_result.chunk_id

                patterns_raw = question.get("patterns", [])
                applicable_patterns = self._filter_patterns(
                    patterns_raw, routing_result.policy_area_id
                )

                # Phase 5 validation: Ensure signal_registry initialized
                if self.signal_registry is None:
                    raise ValueError(
                        f"SignalRegistry required for Phase 5 signal resolution "
                        f"but not initialized for question {question_id}"
                    )

                resolved_signals = self._resolve_signals_for_question(
                    question,
                    routing_result.target_chunk,
                    self.signal_registry,
                )

                # Phase 6: Schema validation (four subphase pipeline)
                # Validates structural compatibility and semantic constraints
                # Allows TypeError/ValueError to propagate to outer handler
                validate_phase6_schema_compatibility(
                    question=question,
                    chunk_expected_elements=routing_result.expected_elements,
                    chunk_id=routing_result.chunk_id,
                    policy_area_id=routing_result.policy_area_id,
                    correlation_id=self.correlation_id,
                )

                task = self._construct_task(
                    question,
                    routing_result,
                    applicable_patterns,
                    resolved_signals,
                    generated_task_ids,
                )
                tasks.append(task)

                if idx % 50 == 0:
                    logger.info(
                        json.dumps(
                            {
                                "event": "task_construction_progress",
                                "tasks_completed": idx,
                                "total_questions": len(questions),
                                "progress_pct": round(
                                    100 * idx / len(questions), 2
                                ),
                                "correlation_id": self.correlation_id,
                            }
                        )
                    )
```

```
                except (ValueError, TypeError) as e:
                    routing_failures += 1

                    logger.error(
                        json.dumps(
                            {
                                "event": "task_construction_failure",
                                "error_event": "routing_or_signal_failure",
                                "question_id": question_id,
                                "question_index": idx,
                                "policy_area_id": policy_area_id,
                                "dimension_id": dimension_id,
                                "chunk_id": chunk_id,
                                "error_type": type(e).__name__,
                                "error_message": str(e),
                                "correlation_id": self.correlation_id,
                                "timestamp": time.time(),
                            }
                        ),
                        exc_info=True,
                    )

                    raise

            expected_task_count = len(questions)
            actual_task_count = len(tasks)

            if actual_task_count != expected_task_count:
                raise ValueError(
                    f"Task count mismatch: Expected {expected_task_count} tasks "
                    f"but constructed {actual_task_count}. "
                    f"Routing successes: {routing_successes}, failures: {routing_failures}"
                )

            tasks, plan_id = self._assemble_execution_plan(
                tasks, questions, self.correlation_id
            )

            logger.info(
                json.dumps(
                    {
                        "event": "task_construction_complete",
                        "total_tasks": actual_task_count,
                        "routing_successes": routing_successes,
                        "routing_failures": routing_failures,
                        "success_rate": round(
                            100 * routing_successes / max(expected_task_count, 1), 2
                        ),
                        "correlation_id": self.correlation_id,
                        "timestamp": time.time(),
                    }
                )
            )

            legacy_tasks = []
```

```
            for task in tasks:
                legacy_task = Task(
                    task_id=task.task_id,
                    dimension=task.dimension_id,
                    question_id=task.question_id,
                    policy_area=task.policy_area_id,
                    chunk_id=task.chunk_id,
                    chunk_index=0,
                    question_text="",
                )
                legacy_tasks.append(legacy_task)

            integrity_hash = self._compute_integrity_hash(legacy_tasks)

            plan = self._construct_execution_plan_phase_8_4(
                sorted_tasks=legacy_tasks,
                plan_id=plan_id,
                chunk_count=self.chunk_count,
                question_count=len(questions),
                integrity_hash=integrity_hash,
            )

            self._validate_cross_task_cardinality(plan, questions)

            logger.info(
                json.dumps(
                    {
                        "event": "build_execution_plan_complete",
                        "correlation_id": self.correlation_id,
                        "plan_id": plan_id,
                        "task_count": len(legacy_tasks),
                        "chunk_count": self.chunk_count,
                        "question_count": len(questions),
                        "integrity_hash": integrity_hash,
                        "chunk_matrix_validated": True,
                        "mode": "chunk_matrix",
                        "phase": "synchronization_phase_complete",
                    }
                )
            )

            return plan

        except ValueError as e:
            synchronization_failures.labels(error_type="validation_failure").inc()
            logger.error(
                json.dumps(
                    {
                        "event": "build_execution_plan_error",
                        "correlation_id": self.correlation_id,
                        "error": str(e),
                        "error_type": "validation_failure",
                    }
                )
            )
            raise
```

```
        except Exception as e:
            synchronization_failures.labels(error_type=type(e).__name__).inc()
            logger.error(
                json.dumps(
                    {
                        "event": "build_execution_plan_error",
                        "correlation_id": self.correlation_id,
                        "error": str(e),
                        "error_type": type(e).__name__,
                    }
                )
            )
            raise

    def _build_with_legacy_chunks(self) -> ExecutionPlan:
        """Build execution plan using legacy document_chunks list."""
        logger.info(
            json.dumps(
                {
                    "event": "build_execution_plan_start",
                    "correlation_id": self.correlation_id,
                    "question_count": self.question_count,
                    "chunk_count": self.chunk_count,
                    "mode": "legacy_chunks",
                    "phase": "synchronization_phase_0",
                }
            )
        )

        try:
            if not self.document_chunks:
                synchronization_failures.labels(error_type="empty_chunks").inc()
                raise ValueError("No document chunks provided")

            if self.question_count == 0:
                synchronization_failures.labels(error_type="empty_questions").inc()
                raise ValueError("No questions found in questionnaire")

            questions = self._extract_questions()
            policy_areas = [f"PA{i:02d}" for i in range(1, 11)]

            tasks: list[Task] = []

            for question in questions:
                for policy_area in policy_areas:
                    for chunk_idx, chunk in enumerate(self.document_chunks):
                        chunk_id = chunk.get("chunk_id", f"chunk_{chunk_idx:04d}")

                        task_id = f"{question['question_id']}_{policy_area}_{chunk_id}"

                        task = Task(
                            task_id=task_id,
                            dimension=question["dimension"],
                            question_id=question["question_id"],
                            policy_area=policy_area,
                            chunk_id=chunk_id,
```

```
                        chunk_index=chunk_idx,
                        question_text=question["question_text"],
                    )

                    tasks.append(task)

                    tasks_constructed.labels(
                        dimension=question["dimension"], policy_area=policy_area
                    ).inc()

        sorted_tasks = sorted(tasks, key=lambda t: t.task_id)

        if len(sorted_tasks) != len(tasks):
            raise RuntimeError(
                f"Task ordering corruption detected: sorted task count {len(sorted_tasks)} "
                f"does not match input task count {len(tasks)}"
            )

        task_serialization = [
            {
                "task_id": t.task_id,
                "question_id": t.question_id,
                "dimension": t.dimension,
                "policy_area": t.policy_area,
                "chunk_id": t.chunk_id,
            }
            for t in sorted_tasks
        ]

        json_bytes = json.dumps(
            task_serialization, sort_keys=True, separators=(",", ":")
        ).encode("utf-8")

        plan_id = hashlib.sha256(json_bytes).hexdigest()

        if len(plan_id) != SHA256_HEX_DIGEST_LENGTH:
            raise ValueError(
                f"Plan identifier validation failure: expected length {SHA256_HEX_DIGEST_LENGTH} but got {len(plan_id)}; "
                "SHA256 implementation may be compromised or monkey-patched"
            )

        if not all(c in "0123456789abcdef" for c in plan_id):
            raise ValueError(
                "Plan identifier validation failure: expected lowercase hexadecimal but got "
                "characters outside '0123456789abcdef' set; SHA256 implementation may be "
                "compromised or monkey-patched"
            )

        integrity_hash = self._compute_integrity_hash(sorted_tasks)

        plan = self._construct_execution_plan_phase_8_4(
            sorted_tasks=sorted_tasks,
            plan_id=plan_id,
            chunk_count=self.chunk_count,
            question_count=len(questions),
            integrity_hash=integrity_hash,
```

```
            )

            self._validate_cross_task_cardinality(plan, questions)

            logger.info(
                json.dumps(
                    {
                        "event": "build_execution_plan_complete",
                        "correlation_id": self.correlation_id,
                        "plan_id": plan_id,
                        "task_count": len(tasks),
                        "chunk_count": self.chunk_count,
                        "question_count": len(questions),
                        "integrity_hash": integrity_hash,
                        "mode": "legacy_chunks",
                        "phase": "synchronization_phase_complete",
                    }
                )
            )

            return plan

        except Exception as e:
            synchronization_failures.labels(error_type=type(e).__name__).inc()
            logger.error(
                json.dumps(
                    {
                        "event": "build_execution_plan_error",
                        "correlation_id": self.correlation_id,
                        "error": str(e),
                        "error_type": type(e).__name__,
                    }
                )
            )
            raise

    def _validate_cross_task_contamination(self, execution_plan: ExecutionPlan) -> None:
        """Build traceability mappings for task-chunk relationship queries.

        Constructs two bidirectional dictionaries enabling efficient task-chunk
        relationship queries and stores them in ExecutionPlan metadata:
        - task_chunk_mapping: Maps each task_id to its chunk_id (one-to-one)
        - chunk_task_mapping: Maps each chunk_id to list of task_ids (one-to-many)

        Args:
            execution_plan: ExecutionPlan to enrich with traceability mappings

        Returns:
            None (modifies execution_plan.metadata in place)
        """
        task_chunk_mapping = {t.task_id: t.chunk_id for t in execution_plan.tasks}

        chunk_task_mapping: dict[str, list[str]] = {}
        for t in execution_plan.tasks:
            chunk_task_mapping.setdefault(t.chunk_id, []).append(t.task_id)

```

```
        execution_plan.metadata["task_chunk_mapping"] = task_chunk_mapping
        execution_plan.metadata["chunk_task_mapping"] = chunk_task_mapping

    def _resolve_signals_for_question(
        self,
        question: dict[str, Any],
        target_chunk: ChunkData,
        signal_registry: SignalRegistry,
    ) -> tuple[Any, ...]:
        """Resolve signals for a question from registry.

        Performs signal resolution with comprehensive validation:
        - Normalizes signal_requirements to empty list if missing/None
        - Calls signal_registry.get_signals_for_chunk with requirements
        - Validates return type is list (raises TypeError if None)
        - Validates each signal has required fields (signal_id, signal_type, content)
        - Detects missing required signals (HARD STOP with ValueError)
        - Detects and warns about duplicate signal types
        - Returns immutable tuple of resolved signals

        Args:
            question: Question dict with signal_requirements field
            target_chunk: Target ChunkData for signal resolution
            signal_registry: Registry implementing get_signals_for_chunk(chunk, requirements)

        Returns:
            Immutable tuple of resolved signals

        Raises:
            TypeError: If signal_registry returns non-list type
            ValueError: If signal missing required field or required signals not found
        """
        question_id = question.get("question_id", "UNKNOWN")
        chunk_id = getattr(target_chunk, "chunk_id", "UNKNOWN")

        # Normalize signal_requirements to empty list if missing or None
        signal_requirements = question.get("signal_requirements")
        if signal_requirements is None:
            signal_requirements = []
        elif not isinstance(signal_requirements, list):
            # If it’s a dict or other type, extract as list if possible
            if isinstance(signal_requirements, dict):
                signal_requirements = list(signal_requirements.keys())
            else:
                signal_requirements = []

        # Call signal_registry.get_signals_for_chunk
        resolved_signals = signal_registry.get_signals_for_chunk(
            target_chunk, signal_requirements
        )

        # Validate return is list type (raise TypeError if None)
        if resolved_signals is None:
            raise TypeError(
                f"SignalRegistry returned {type(None).__name__} for question {question_id} "
                f"chunk {chunk_id}, expected list"
```

```
            )

        if not isinstance(resolved_signals, list):
            raise TypeError(
                f"SignalRegistry returned {type(resolved_signals).__name__} for question {question_id} "
                f"chunk {chunk_id}, expected list"
            )

        # Validate each signal has required fields
        required_fields = ["signal_id", "signal_type", "content"]
        for i, signal in enumerate(resolved_signals):
            for field in required_fields:
                # Try both attribute and dict access
                has_field = False
                try:
                    if hasattr(signal, field):
                        getattr(signal, field)
                        has_field = True
                except (AttributeError, KeyError):
                    pass

                if not has_field:
                    try:
                        if isinstance(signal, dict) and field in signal:
                            has_field = True
                    except (TypeError, KeyError):
                        pass

                if not has_field:
                    raise ValueError(
                        f"Signal at index {i} missing field {field} for question {question_id}"
                    )

        # Extract signal_types into set
        signal_types = set()
        for signal in resolved_signals:
            # Try attribute access first, then dict access
            signal_type = None
            try:
                if hasattr(signal, "signal_type"):
                    signal_type = signal.signal_type
            except AttributeError:
                pass

            if signal_type is None:
                try:
                    if isinstance(signal, dict):
                        signal_type = signal["signal_type"]
                except (KeyError, TypeError):
                    pass

            if signal_type is not None:
                signal_types.add(signal_type)

        # Compute missing signals
        requirements_set = set(signal_requirements) if signal_requirements else set()
```

```
        missing_signals = requirements_set - signal_types

        # Raise ValueError if non-empty (HARD STOP)
        if missing_signals:
            missing_sorted = sorted(missing_signals)
            raise ValueError(
                f"Synchronization Failure for MQC {question_id}: "
                f"Missing required signals {missing_sorted} for chunk {chunk_id}"
            )

        # Detect duplicates
        if len(resolved_signals) > len(signal_types):
            # Find duplicate types for logging
            type_counts: dict[Any, int] = {}
            for signal in resolved_signals:
                signal_type = None
                try:
                    if hasattr(signal, "signal_type"):
                        signal_type = signal.signal_type
                except AttributeError:
                    pass

                if signal_type is None:
                    try:
                        if isinstance(signal, dict):
                            signal_type = signal["signal_type"]
                    except (KeyError, TypeError):
                        pass

                if signal_type is not None:
                    type_counts[signal_type] = type_counts.get(signal_type, 0) + 1

            duplicate_types = [t for t, count in type_counts.items() if count > 1]

            logger.warning(
                "signal_resolution_duplicates",
                extra={
                    "question_id": question_id,
                    "chunk_id": chunk_id,
                    "correlation_id": self.correlation_id,
                    "duplicate_types": duplicate_types,
                },
            )

        # Emit success log
        logger.debug(
            "signal_resolution_success",
            extra={
                "question_id": question_id,
                "chunk_id": chunk_id,
                "correlation_id": self.correlation_id,
                "resolved_count": len(resolved_signals),
                "required_count": len(signal_requirements),
                "signal_types": list(signal_types),
            },
        )
```

```

        # Return tuple for immutability
        return tuple(resolved_signals)

    def _serialize_and_verify_plan(self, plan: ExecutionPlan) -> str:
        """Serialize ExecutionPlan and verify round-trip integrity.

        Serializes the execution plan to JSON, deserializes it back, reconstructs
        an ExecutionPlan instance, and validates that plan_id and task count match
        the original to ensure serialization is lossless.

        Args:
            plan: ExecutionPlan instance to serialize and verify

        Returns:
            Validated serialized JSON string ready for persistent storage

        Raises:
            ValueError: If plan_id mismatch or task count mismatch detected
        """
        plan_dict = plan.to_dict()
        serialized_json = json.dumps(plan_dict, sort_keys=True, separators=(",", ":"))

        deserialized_dict = json.loads(serialized_json)
        reconstructed_plan = ExecutionPlan.from_dict(deserialized_dict)

        if reconstructed_plan.plan_id != plan.plan_id:
            raise ValueError(
                f"Serialization verification failed: plan_id mismatch "
                f"(original={plan.plan_id}, reconstructed={reconstructed_plan.plan_id})"
            )

        original_task_count = len(plan.tasks)
        reconstructed_task_count = len(reconstructed_plan.tasks)

        if reconstructed_task_count != original_task_count:
            raise ValueError(
                f"Serialization verification failed: task count mismatch "
                f"(original={original_task_count}, reconstructed={reconstructed_task_count})"
            )

        return serialized_json

    def _archive_to_storage(
        self,
        serialized_json: str,
        execution_plan: ExecutionPlan,
        base_dir: Path,
    ) -> ExecutionPlan:
        """Archive execution plan to storage with atomic index update and rollback.

        Constructs storage path as base_dir / 'execution_plans' / f'{plan_id}.json',
        writes serialized JSON with verification, and atomically updates index with
        rollback logic for orphaned files.

        Args:
```

```
            serialized_json: Serialized JSON string of execution plan
            execution_plan: ExecutionPlan instance to archive
            base_dir: Base directory path for storage

        Returns:
            Original ExecutionPlan instance unchanged

        Raises:
            ValueError: If write fails (re-raised from IOError)
            IOError: If write verification fails (content mismatch)
        """
        plan_id = execution_plan.plan_id
        storage_path = base_dir / "execution_plans" / f"{plan_id}.json"

        try:
            storage_path.parent.mkdir(parents=True, exist_ok=True)
        except IOError as e:
            raise ValueError(
                f"Failed to create parent directories for plan_id={plan_id}, "
                f"storage_path={storage_path}: {e}"
            ) from e

        try:
            storage_path.write_text(serialized_json, encoding="utf-8")
        except IOError as e:
            raise ValueError(
                f"Failed to write execution plan for plan_id={plan_id}, "
                f"storage_path={storage_path}: {e}"
            ) from e

        try:
            read_content = storage_path.read_text(encoding="utf-8")
            if read_content != serialized_json:
                storage_path.unlink()
                raise IOError(
                    f"Write verification failed for plan_id={plan_id}, "
                    f"storage_path={storage_path}: content mismatch after write"
                )
        except IOError as e:
            if storage_path.exists():
                storage_path.unlink()
            raise

        index_path = base_dir / "execution_plans" / "index.jsonl"
        index_entry = {
            "plan_id": plan_id,
            "storage_path": str(storage_path),
            "created_at": execution_plan.created_at,
            "task_count": len(execution_plan.tasks),
            "integrity_hash": execution_plan.integrity_hash,
            "correlation_id": execution_plan.correlation_id,
        }

        try:
            with open(index_path, "a", encoding="utf-8") as f:
                f.write(json.dumps(index_entry) + "\n")
```

```
        except IOError as e:
            if storage_path.exists():
                storage_path.unlink()
            raise ValueError(
                f"Failed to update index for plan_id={plan_id}, "
                f"storage_path={storage_path}: {e}"
            ) from e

        logger.info(
            "execution_plan_archived",
            extra={
                "event": "execution_plan_archived",
                "plan_id": plan_id,
                "storage_path": str(storage_path),
                "task_count": len(execution_plan.tasks),
                "integrity_hash": execution_plan.integrity_hash,
                "correlation_id": execution_plan.correlation_id,
                "created_at": execution_plan.created_at,
            },
        )

        return execution_plan


__all__ = [
    "IrrigationSynchronizer",
    "ExecutionPlan",
    "Task",
    "ChunkRoutingResult",
    "SignalRegistry",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/memory_safety.py
================================================================================

"""
Memory Safety Guards for Executor System

Provides systematic memory safety across all executors processing large objects
(entities, DAGs, causal effects, etc.) with:
- Size estimation for Python objects and JSON serialization
- Configurable limits per executor type
- Memory pressure detection using psutil
- Fallback strategies (sampling, truncation) with logging and metrics
"""

from __future__ import annotations

import json
import logging
import sys
from dataclasses import dataclass
from enum import Enum
```

```
from typing import Any, TypeVar

try:
    import psutil

    PSUTIL_AVAILABLE = True
except ImportError:
    PSUTIL_AVAILABLE = False
    psutil = None

logger = logging.getLogger(__name__)

T = TypeVar("T")


class ExecutorType(Enum):
    """Executor classification for memory limit configuration."""

    ENTITY = "entity"
    DAG = "dag"
    CAUSAL_EFFECTS = "causal_effects"
    SEMANTIC = "semantic"
    FINANCIAL = "financial"
    GENERIC = "generic"


@dataclass
class MemorySafetyConfig:
    """Configuration for memory safety per executor type."""

    entity_limit_mb: float = 1.0
    dag_limit_mb: float = 5.0
    causal_effects_limit_mb: float = 10.0
    semantic_limit_mb: float = 2.0
    financial_limit_mb: float = 2.0
    generic_limit_mb: float = 5.0

    memory_pressure_threshold_pct: float = 80.0
    enable_pressure_detection: bool = True
    enable_auto_sampling: bool = True
    enable_auto_truncation: bool = True

    max_list_elements: int = 1000
    max_string_length: int = 100_000
    max_dict_keys: int = 500

    def get_limit_bytes(self, executor_type: ExecutorType) -> int:
        """Get memory limit in bytes for executor type."""
        limits = {
            ExecutorType.ENTITY: self.entity_limit_mb,
            ExecutorType.DAG: self.dag_limit_mb,
            ExecutorType.CAUSAL_EFFECTS: self.causal_effects_limit_mb,
            ExecutorType.SEMANTIC: self.semantic_limit_mb,
            ExecutorType.FINANCIAL: self.financial_limit_mb,
            ExecutorType.GENERIC: self.generic_limit_mb,
        }
```

```
        return int(limits[executor_type] * 1024 * 1024)


@dataclass
class MemoryMetrics:
    """Memory usage metrics for monitoring."""

    object_size_bytes: int
    json_size_bytes: int
    pressure_pct: float | None
    was_truncated: bool
    was_sampled: bool
    fallback_strategy: str | None
    elements_before: int | None
    elements_after: int | None


class MemoryPressureDetector:
    """Detects system memory pressure to trigger fallback strategies."""

    @staticmethod
    def get_memory_pressure_pct() -> float | None:
        """Get current system memory pressure percentage (0-100).

        Returns None if psutil is not available.
        """
        if not PSUTIL_AVAILABLE:
            return None

        try:
            memory = psutil.virtual_memory()
            return memory.percent
        except Exception as e:
            logger.warning(f"Failed to read memory pressure: {e}")
            return None

    @staticmethod
    def is_under_pressure(threshold_pct: float = 80.0) -> bool:
        """Check if system is under memory pressure."""
        pressure = MemoryPressureDetector.get_memory_pressure_pct()
        if pressure is None:
            return False
        return pressure >= threshold_pct


class ObjectSizeEstimator:
    """Estimates size of Python objects including deep structures."""

    @staticmethod
    def estimate_object_size(obj: Any) -> int:
        """Estimate size of Python object in bytes using sys.getsizeof.

        For containers, recursively estimates contents up to reasonable depth.
        """
        size = sys.getsizeof(obj)

```

```
        if isinstance(obj, dict):
            size += sum(
                ObjectSizeEstimator.estimate_object_size(k)
                + ObjectSizeEstimator.estimate_object_size(v)
                for k, v in obj.items()
            )
        elif isinstance(obj, (list, tuple, set)):
            size += sum(ObjectSizeEstimator.estimate_object_size(item) for item in obj)
        elif hasattr(obj, "__dict__"):
            size += ObjectSizeEstimator.estimate_object_size(obj.__dict__)

        return size

    @staticmethod
    def estimate_json_size(obj: Any) -> int:
        """Estimate serialized JSON size without full serialization.

        Fast approximation:
        - Strings: len(str) * 1.2 (accounting for escaping)
        - Numbers: ~20 bytes
        - Booleans/None: ~10 bytes
        - Containers: sum of contents + overhead
        """
        if obj is None:
            return 4

        if isinstance(obj, bool):
            return 5

        if isinstance(obj, int):
            return len(str(obj)) + 2

        if isinstance(obj, float):
            return 20

        if isinstance(obj, str):
            return int(len(obj) * 1.2) + 2

        if isinstance(obj, (list, tuple)):
            return 2 + sum(ObjectSizeEstimator.estimate_json_size(item) for item in obj)

        if isinstance(obj, dict):
            size = 2
            for k, v in obj.items():
                size += ObjectSizeEstimator.estimate_json_size(k) + 1
                size += ObjectSizeEstimator.estimate_json_size(v) + 1
            return size

        try:
            return len(json.dumps(obj))
        except (TypeError, ValueError):
            return sys.getsizeof(obj)


class FallbackStrategy:
    """Fallback strategies for handling objects exceeding size limits."""
```

```

    @staticmethod
    def sample_list(
        items: list[T], max_elements: int, *, preserve_order: bool = True
    ) -> list[T]:
        """Sample list to max_elements using systematic sampling.

        Args:
            items: List to sample
            max_elements: Maximum elements to keep
            preserve_order: Whether to maintain original order

        Returns:
            Sampled list
        """
        if len(items) <= max_elements:
            return items

        if preserve_order:
            step = len(items) / max_elements
            indices = [int(i * step) for i in range(max_elements)]
            return [items[i] for i in indices]

        import random

        return random.sample(items, max_elements)

    @staticmethod
    def truncate_string(s: str, max_length: int) -> str:
        """Truncate string to max_length with ellipsis."""
        if len(s) <= max_length:
            return s
        return s[: max_length - 3] + "..."

    @staticmethod
    def truncate_dict(d: dict[str, Any], max_keys: int) -> dict[str, Any]:
        """Truncate dictionary to max_keys, preserving most important keys."""
        if len(d) <= max_keys:
            return d

        priority_keys = ["id", "name", "type", "label", "value", "score", "confidence"]

        result = {}
        for key in priority_keys:
            if key in d and len(result) < max_keys:
                result[key] = d[key]

        remaining = max_keys - len(result)
        for key in d:
            if key not in result and remaining > 0:
                result[key] = d[key]
                remaining -= 1

        return result

    @staticmethod
```

```
    def apply_recursive_truncation(
        obj: Any, config: MemorySafetyConfig, depth: int = 0, max_depth: int = 10
    ) -> tuple[Any, bool]:
        """Recursively apply truncation strategies to object tree.

        Returns:
            (truncated_object, was_modified)
        """
        if depth > max_depth:
            return obj, False

        modified = False

        if isinstance(obj, str):
            if len(obj) > config.max_string_length:
                obj = FallbackStrategy.truncate_string(obj, config.max_string_length)
                modified = True

        elif isinstance(obj, list):
            if len(obj) > config.max_list_elements:
                obj = FallbackStrategy.sample_list(obj, config.max_list_elements)
                modified = True

            new_items = []
            for item in obj:
                new_item, item_modified = FallbackStrategy.apply_recursive_truncation(
                    item, config, depth + 1, max_depth
                )
                new_items.append(new_item)
                modified = modified or item_modified
            obj = new_items

        elif isinstance(obj, dict):
            if len(obj) > config.max_dict_keys:
                obj = FallbackStrategy.truncate_dict(obj, config.max_dict_keys)
                modified = True

            new_dict = {}
            for k, v in obj.items():
                new_v, v_modified = FallbackStrategy.apply_recursive_truncation(
                    v, config, depth + 1, max_depth
                )
                new_dict[k] = new_v
                modified = modified or v_modified
            obj = new_dict

        return obj, modified


class MemorySafetyGuard:
    """Main guard for memory-safe object processing."""

    def __init__(self, config: MemorySafetyConfig | None = None):
        self.config = config or MemorySafetyConfig()
        self.metrics: list[MemoryMetrics] = []

```

```
    def check_and_process(
        self, obj: Any, executor_type: ExecutorType, label: str = "object"
    ) -> tuple[Any, MemoryMetrics]:
        """Check object size and apply fallback strategies if needed.

        Args:
            obj: Object to check
            executor_type: Type of executor processing this object
            label: Human-readable label for logging

        Returns:
            (processed_object, metrics)
        """
        obj_size = ObjectSizeEstimator.estimate_object_size(obj)
        json_size = ObjectSizeEstimator.estimate_json_size(obj)
        limit_bytes = self.config.get_limit_bytes(executor_type)

        pressure_pct = None
        if self.config.enable_pressure_detection:
            pressure_pct = MemoryPressureDetector.get_memory_pressure_pct()

        was_truncated = False
        was_sampled = False
        fallback_strategy = None
        elements_before = self._count_elements(obj)

        under_pressure = (
            pressure_pct is not None
            and pressure_pct >= self.config.memory_pressure_threshold_pct
        )

        if obj_size > limit_bytes or json_size > limit_bytes or under_pressure:
            reason = []
            if obj_size > limit_bytes:
                reason.append(
                    f"object size {obj_size / (1024*1024):.2f}MB exceeds {limit_bytes / (1024*1024):.2f}MB"
                )
            if json_size > limit_bytes:
                reason.append(
                    f"JSON size {json_size / (1024*1024):.2f}MB exceeds {limit_bytes / (1024*1024):.2f}MB"
                )
            if under_pressure:
                reason.append(
                    f"memory pressure {pressure_pct:.1f}% >= {self.config.memory_pressure_threshold_pct}%"
                )

            logger.warning(
                f"Memory safety triggered for {label} ({executor_type.value}): {'; '.join(reason)}"
            )

            if self.config.enable_auto_truncation:
                obj, was_truncated = FallbackStrategy.apply_recursive_truncation(
                    obj, self.config
                )
                fallback_strategy = "truncation"

```

```
                obj_size = ObjectSizeEstimator.estimate_object_size(obj)
                json_size = ObjectSizeEstimator.estimate_json_size(obj)
                logger.info(
                    f"Applied truncation to {label}: "
                    f"new size {obj_size / (1024*1024):.2f}MB object, "
                    f"{json_size / (1024*1024):.2f}MB JSON"
                )

        elements_after = self._count_elements(obj)

        metrics = MemoryMetrics(
            object_size_bytes=obj_size,
            json_size_bytes=json_size,
            pressure_pct=pressure_pct,
            was_truncated=was_truncated,
            was_sampled=was_sampled,
            fallback_strategy=fallback_strategy,
            elements_before=elements_before,
            elements_after=elements_after,
        )

        self.metrics.append(metrics)
        return obj, metrics

    def _count_elements(self, obj: Any) -> int | None:
        """Count elements in container objects."""
        if isinstance(obj, (list, tuple)):
            return len(obj)
        if isinstance(obj, dict):
            return len(obj)
        return None

    def get_metrics_summary(self) -> dict[str, Any]:
        """Get summary of all memory operations."""
        if not self.metrics:
            return {
                "total_operations": 0,
                "truncations": 0,
                "samplings": 0,
                "avg_object_size_mb": 0.0,
                "avg_json_size_mb": 0.0,
                "max_object_size_mb": 0.0,
                "max_json_size_mb": 0.0,
            }

        return {
            "total_operations": len(self.metrics),
            "truncations": sum(1 for m in self.metrics if m.was_truncated),
            "samplings": sum(1 for m in self.metrics if m.was_sampled),
            "avg_object_size_mb": sum(m.object_size_bytes for m in self.metrics)
            / len(self.metrics)
            / (1024 * 1024),
            "avg_json_size_mb": sum(m.json_size_bytes for m in self.metrics)
            / len(self.metrics)
            / (1024 * 1024),
            "max_object_size_mb": max(m.object_size_bytes for m in self.metrics)
```

```
            / (1024 * 1024),
            "max_json_size_mb": max(m.json_size_bytes for m in self.metrics)
            / (1024 * 1024),
            "pressure_samples": [
                m.pressure_pct for m in self.metrics if m.pressure_pct is not None
            ],
        }


def create_default_guard() -> MemorySafetyGuard:
    """Create memory safety guard with default configuration."""
    return MemorySafetyGuard(MemorySafetyConfig())


__all__ = [
    "ExecutorType",
    "MemorySafetyConfig",
    "MemoryMetrics",
    "MemoryPressureDetector",
    "ObjectSizeEstimator",
    "FallbackStrategy",
    "MemorySafetyGuard",
    "create_default_guard",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/method_registry.py
================================================================================

"""Method Registry with lazy instantiation and injection pattern.

This module implements a method injection factory that:
1. Loads only the methods needed (not full classes)
2. Instantiates classes lazily (only when first method is called)
3. Caches instances for reuse
4. Isolates errors per method (failures don’t cascade)
5. Allows direct function injection (bypassing classes)

Architecture:
    MethodRegistry
        â\224\234â\224\200 _class_paths: mapping of class names to import paths
        â\224\234â\224\200 _instance_cache: lazily instantiated class instances
        â\224\234â\224\200 _direct_methods: directly injected functions
        â\224\224â\224\200 get_method(): returns callable for (class_name, method_name)

Benefits:
- No upfront class loading (lightweight imports)
- Failed classes don’t block working methods
- Direct function injection for custom implementations
- Instance reuse through caching
"""
from __future__ import annotations

import logging
```

```
import threading
from importlib import import_module
from typing import Any, Callable

logger = logging.getLogger(__name__)


class MethodRegistryError(RuntimeError):
    """Raised when a method cannot be retrieved."""


class MethodRegistry:
    """Registry for lazy method injection without full class instantiation."""

    def __init__(self, class_paths: dict[str, str] | None = None) -> None:
        """Initialize the method registry.

        Args:
            class_paths: Optional mapping of class names to import paths.
                        If None, uses default paths from class_registry.
        """
        # Import class paths from existing registry
        if class_paths is None:
            from farfan_pipeline.core.orchestrator.class_registry import get_class_paths
            class_paths = dict(get_class_paths())

        self._class_paths = class_paths
        self._instance_cache: dict[str, Any] = {}
        self._direct_methods: dict[tuple[str, str], Callable[..., Any]] = {}
        self._failed_classes: set[str] = set()
        self._lock = threading.Lock()

        # Special instantiation rules (from original MethodExecutor)
        self._special_instantiation: dict[str, Callable[[type], Any]] = {}

    def inject_method(
        self,
        class_name: str,
        method_name: str,
        method: Callable[..., Any],
    ) -> None:
        """Directly inject a method without needing a class.

        This allows bypassing class instantiation entirely.

        Args:
            class_name: Virtual class name for routing
            method_name: Method name
            method: Callable to inject
        """
        key = (class_name, method_name)
        self._direct_methods[key] = method
        logger.info(
            "method_injected_directly",
            class_name=class_name,
            method_name=method_name,
```

```
        )

    def register_instantiation_rule(
        self,
        class_name: str,
        instantiator: Callable[[type], Any],
    ) -> None:
        """Register special instantiation logic for a class.

        Args:
            class_name: Class name requiring special instantiation
            instantiator: Function that takes class type and returns instance
        """
        self._special_instantiation[class_name] = instantiator
        logger.debug(
            "instantiation_rule_registered",
            class_name=class_name,
        )

    def _load_class(self, class_name: str) -> type:
        """Load a class type from import path.

        Args:
            class_name: Name of class to load

        Returns:
            Class type

        Raises:
            MethodRegistryError: If class cannot be loaded
        """
        if class_name not in self._class_paths:
            raise MethodRegistryError(
                f"Class '{class_name}' not found in registry paths"
            )

        path = self._class_paths[class_name]
        module_name, _, attr_name = path.rpartition(".")

        if not module_name:
            raise MethodRegistryError(
                f"Invalid path for '{class_name}': {path}"
            )

        try:
            module = import_module(module_name)
            cls = getattr(module, attr_name)

            if not isinstance(cls, type):
                raise MethodRegistryError(
                    f"'{class_name}' is not a class: {type(cls).__name__}"
                )

            return cls

        except ImportError as exc:
```

```
            raise MethodRegistryError(
                f"Cannot import class '{class_name}' from {path}: {exc}"
            ) from exc
        except AttributeError as exc:
            raise MethodRegistryError(
                f"Class '{attr_name}' not found in module {module_name}: {exc}"
            ) from exc

    def _instantiate_class(self, class_name: str, cls: type) -> Any:
        """Instantiate a class using special rules or default constructor.

        Args:
            class_name: Name of class (for special rule lookup)
            cls: Class type to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If instantiation fails
        """
        # Use special instantiation rule if registered
        if class_name in self._special_instantiation:
            try:
                instantiator = self._special_instantiation[class_name]
                instance = instantiator(cls)
                logger.debug(
                    "class_instantiated_with_special_rule",
                    class_name=class_name,
                )
                return instance
            except Exception as exc:
                raise MethodRegistryError(
                    f"Special instantiation failed for '{class_name}': {exc}"
                ) from exc

        # Default instantiation (no-args constructor)
        try:
            instance = cls()
            logger.debug(
                "class_instantiated_default",
                class_name=class_name,
            )
            return instance
        except Exception as exc:
            raise MethodRegistryError(
                f"Default instantiation failed for '{class_name}': {exc}"
            ) from exc

    def _get_instance(self, class_name: str) -> Any:
        """Get or create instance of a class (lazy + cached).

        Args:
            class_name: Name of class to instantiate

        Returns:
```

```
            Instance of the class

        Raises:
            MethodRegistryError: If class cannot be instantiated
        """
        # Check if already failed
        if class_name in self._failed_classes:
            raise MethodRegistryError(
                f"Class '{class_name}' previously failed to instantiate"
            )

        # Use a lock to ensure thread-safe instantiation
        with self._lock:
            # Double-check if another thread instantiated it while waiting for the lock
            if class_name in self._instance_cache:
                return self._instance_cache[class_name]

            # Load and instantiate class
            try:
                cls = self._load_class(class_name)
                instance = self._instantiate_class(class_name, cls)
                self._instance_cache[class_name] = instance
                logger.info(
                    "class_instantiated_lazy",
                    class_name=class_name,
                )
                return instance

            except MethodRegistryError:
                # Mark as failed to avoid repeated attempts
                self._failed_classes.add(class_name)
                raise

    def get_method(
        self,
        class_name: str,
        method_name: str,
    ) -> Callable[..., Any]:
        """Get method callable with lazy instantiation.

        This is the main entry point for retrieving methods.

        Args:
            class_name: Name of class containing the method
            method_name: Name of method to retrieve

        Returns:
            Callable method (bound or injected)

        Raises:
            MethodRegistryError: If method cannot be retrieved
        """
        # Check for directly injected method first
        key = (class_name, method_name)
        if key in self._direct_methods:
            logger.debug(
```

```
                "method_retrieved_direct",
                class_name=class_name,
                method_name=method_name,
            )
            return self._direct_methods[key]

        # Get instance (lazy) and retrieve method
        try:
            instance = self._get_instance(class_name)
            method = getattr(instance, method_name)

            if not callable(method):
                raise MethodRegistryError(
                    f"'{class_name}.{method_name}' is not callable"
                )

            logger.debug(
                "method_retrieved_from_instance",
                class_name=class_name,
                method_name=method_name,
            )
            return method

        except AttributeError as exc:
            raise MethodRegistryError(
                f"Method '{method_name}' not found on class '{class_name}'"
            ) from exc

    def has_method(self, class_name: str, method_name: str) -> bool:
        """Check if a method is available (without instantiating).

        Args:
            class_name: Name of class
            method_name: Name of method

        Returns:
            True if method exists (or is directly injected)
        """
        # Check direct injection
        key = (class_name, method_name)
        if key in self._direct_methods:
            return True

        # Check if class is known and not failed
        if class_name in self._failed_classes:
            return False

        if class_name not in self._class_paths:
            return False

        # If instance exists, check method
        if class_name in self._instance_cache:
            instance = self._instance_cache[class_name]
            return hasattr(instance, method_name)

        # Otherwise, assume it exists (lazy check)
```

```
        # Full validation happens on first get_method() call
        return True

    def get_stats(self) -> dict[str, Any]:
        """Get registry statistics.

        Returns:
            Dictionary with registry stats
        """
        return {
            "total_classes_registered": len(self._class_paths),
            "instantiated_classes": len(self._instance_cache),
            "failed_classes": len(self._failed_classes),
            "direct_methods_injected": len(self._direct_methods),
            "instantiated_class_names": list(self._instance_cache.keys()),
            "failed_class_names": list(self._failed_classes),
        }


def setup_default_instantiation_rules(registry: MethodRegistry) -> None:
    """Setup default special instantiation rules.

    These rules replicate the logic from the original MethodExecutor
    for classes that need non-default instantiation.

    Args:
        registry: MethodRegistry to configure
    """
    # MunicipalOntology - shared instance pattern
    ontology_instance = None

    def instantiate_ontology(cls: type) -> Any:
        nonlocal ontology_instance
        if ontology_instance is None:
            ontology_instance = cls()
        return ontology_instance

    registry.register_instantiation_rule("MunicipalOntology", instantiate_ontology)

    # SemanticAnalyzer, PerformanceAnalyzer, TextMiningEngine - need ontology
    def instantiate_with_ontology(cls: type) -> Any:
        if ontology_instance is None:
            raise MethodRegistryError(
                f"Cannot instantiate {cls.__name__}: MunicipalOntology not available"
            )
        return cls(ontology_instance)

    for class_name in ["SemanticAnalyzer", "PerformanceAnalyzer", "TextMiningEngine"]:
        registry.register_instantiation_rule(class_name, instantiate_with_ontology)

    # PolicyTextProcessor - needs ProcessorConfig
    def instantiate_policy_processor(cls: type) -> Any:
        try:
            from farfan_pipeline.processing.policy_processor import ProcessorConfig
            return cls(ProcessorConfig())
        except ImportError as exc:
```

```
            raise MethodRegistryError(
                "Cannot instantiate PolicyTextProcessor: ProcessorConfig unavailable"
            ) from exc

    registry.register_instantiation_rule("PolicyTextProcessor", instantiate_policy_processor)


__all__ = [
    "MethodRegistry",
    "MethodRegistryError",
    "setup_default_instantiation_rules",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/method_source_validator.py
================================================================================


import ast
import os
import json
from typing import Dict, List, Any

class MethodSourceValidator:
    def __init__(self, base_path: str = "src/farfan_pipeline"):
        self.base_path = base_path
        self.source_map = self._build_source_map()

    def _build_source_map(self) -> Dict[str, Dict[str, Any]]:
        class_map = {}
        for root, _, files in os.walk(self.base_path):
            for file in files:
                if file.endswith(".py"):
                    file_path = os.path.join(root, file)
                    with open(file_path, "r", encoding="utf-8") as f:
                        try:
                            tree = ast.parse(f.read(), filename=file_path)
                            for node in ast.walk(tree):
                                if isinstance(node, ast.ClassDef):
                                    class_name = node.name
                                    methods = []
                                    for item in node.body:
                                        if isinstance(item, ast.FunctionDef):
                                            methods.append(item.name)

                                    if class_name in class_map:
                                        # In case of duplicate class names, we might need a more robust way
                                        # to handle this, but for now we'll just overwrite.
                                        # A better approach could be to store a list of locations.
                                        pass

                                    class_map[class_name] = {
                                        "file_path": file_path,
                                        "methods": methods,
```

```
                                    }
                        except Exception as e:
                            print(f"Error parsing {file_path}: {e}")
        return class_map

    def validate_executor_methods(self, executor_methods_path: str = "src/farfan_pipeline/core/orchestrator/executors_methods.json") -> Dict[str, List[str]]
:
        with open(executor_methods_path, "r") as f:
            executor_data = json.load(f)

        declared_methods = set()
        for executor_info in executor_data:
            for method_info in executor_info.get("methods", []):
                class_name = method_info.get("class")
                method_name = method_info.get("method")
                if class_name and method_name:
                    declared_methods.add(f"{class_name}.{method_name}")

        valid = []
        missing = []

        for method_fqn in declared_methods:
            if "." not in method_fqn:
                # Assuming methods are always Class.method
                continue

            class_name, method_name = method_fqn.split(".", 1)

            if class_name not in self.source_map:
                missing.append(method_fqn)
                continue

            class_info = self.source_map[class_name]
            if method_name not in class_info["methods"]:
                missing.append(method_fqn)
            else:
                valid.append(method_fqn)

        # Phantom methods would be those in source but not declared.
        # The user’s request seems to focus on missing/valid from declaration.
        # "phantom" is defined by user as "executors call fantasy methods"
        # which is covered by "missing"
        return {"valid": valid, "missing": missing, "phantom": []}


    def generate_source_truth_map(self) -> Dict[str, Dict[str, Any]]:
        source_truth = {}
        for class_name, info in self.source_map.items():
            file_path = info["file_path"]
            with open(file_path, "r", encoding="utf-8") as f:
                tree = ast.parse(f.read(), filename=file_path)
                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_name = item.name
```

```
                                fqn = f"{class_name}.{method_name}"

                                # Basic signature extraction
                                args = [arg.arg for arg in item.args.args]
                                signature = f"({', '.join(args)})"
                                # A more advanced version would parse type hints if they exist

                                source_truth[fqn] = {
                                    "exists": True,
                                    "file": file_path,
                                    "line": item.lineno,
                                    "signature": signature,
                                }
        return source_truth

if __name__ == "__main__":
    validator = MethodSourceValidator()

    # 1. Generate the ground-truth map
    source_truth_map = validator.generate_source_truth_map()
    output_path = "method_source_truth.json"
    with open(output_path, "w") as f:
        json.dump(source_truth_map, f, indent=4)
    print(f"Generated source truth map at {output_path}")

    # 2. Validate executor methods
    validation_report = validator.validate_executor_methods()
    report_path = "executor_validation_report.json"
    with open(report_path, "w") as f:
        json.dump(validation_report, f, indent=4)
    print(f"Validation report generated at {report_path}")

    print("\nValidation Summary:")
    print(f"  - Valid methods: {len(validation_report['valid'])}")
    print(f"  - Missing methods: {len(validation_report['missing'])}")
    if validation_report['missing']:
        print("\nMissing methods:")
        for method in validation_report['missing']:
            print(f"  - {method}")



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/phase6_validation.py
================================================================================

"""Phase 6: Schema Validation Pipeline - Four Subphase Architecture.

This module implements Phase 6 as a complete validation pipeline with four subphases:

Phase 6.1: Classification & Extraction
    - Extracts question_global via bracket notation (question["question_global"])
    - Extracts expected_elements via get method with None handling
    - Classifies types using isinstance checks in None-list-dict-invalid order
    - Stores classification tuple before any iteration occurs

```

```
Phase 6.2: Structural Validation
    - Checks invalid types first with human-readable type names
    - Enforces homogeneity allowing None compatibility
    - Validates list length equality and dict key set equality
    - Uses symmetric difference computation for dict key validation
    - Returns silently on dual-None without logging

Phase 6.3: Semantic Validation
    - Iterates deterministically via enumerate-zip for lists and sorted keys for dicts
    - Extracts type-required-minimum fields with get defaults
    - Implements asymmetric required implication as not-q-or-c boolean expression
    - Enforces c-min-greater-equal-q-min threshold ordering
    - Returns validated element count

Phase 6.4: Orchestrator
    - Invokes structural then semantic layers in sequence
    - Captures element count return value
    - Emits debug log with has_required_fields and has_minimum_thresholds computed
      via any-element-iteration
    - Logs info warning for None chunk schema with non-None question schema
    - Integrates into build_with_chunk_matrix loop after Phase 5 before construct_task
    - Allows TypeError-ValueError propagation to outer handler without try-except wrapping

Architecture:
    Phase 6.1 â\206\222 Phase 6.2 â\206\222 Phase 6.3 â\206\222 Phase 6.4
    (Sequential root) (Structural) (Semantic) (Synchronization barrier)

Parallelization:
    - Phase 6.1: Sequential root (extracts and classifies)
    - Phase 6.2-6.3: Concurrency potential (independent validation layers)
    - Phase 6.4: Synchronization barrier (aggregates results)
"""

from __future__ import annotations

import logging
from typing import Any

logger = logging.getLogger(__name__)

MAX_QUESTION_GLOBAL = 999


def _classify_expected_elements_type(value: Any) -> str:  # noqa: ANN401
    """Phase 6.1: Classify expected_elements type using isinstance checks.

    Performs type classification in None-list-dict-invalid order via identity
    test for None, then isinstance checks for list and dict, with any other
    type classified as invalid.

    Args:
        value: Value to classify (expected_elements from question or chunk)

    Returns:
        Type classification string: "none", "list", "dict", or "invalid"

```

```
    Classification Order:
        1. None via identity test (value is None)
        2. list via isinstance(value, list)
        3. dict via isinstance(value, dict)
        4. invalid for all other types
    """
    if value is None:
        return "none"
    elif isinstance(value, list):
        return "list"
    elif isinstance(value, dict):
        return "dict"
    else:
        return "invalid"


def _extract_and_classify_schemas(
    question: dict[str, Any],
    chunk_expected_elements: list[dict[str, Any]] | dict[str, Any] | None,
    question_id: str,
) -> tuple[int, Any, Any, str, str]:  # noqa: ANN401
    """Phase 6.1: Extract question_global and expected_elements, classify types.

    Extracts question_global via bracket notation (question["question_global"])
    and expected_elements via get method with None default. Classifies both
    schema types and stores classification tuple before any iteration occurs.

    Args:
        question: Question dictionary from questionnaire
        chunk_expected_elements: expected_elements from chunk routing result
        question_id: Question identifier for error reporting

    Returns:
        Tuple of (question_global, question_schema, chunk_schema,
                  question_type, chunk_type)

    Raises:
        ValueError: If question_global is missing, invalid type, or out of range
    """
    # Extract question_global via bracket notation
    try:
        question_global = question["question_global"]
    except KeyError as e:
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            "question_global field is required but missing"
        ) from e

    # Validate question_global
    if not isinstance(question_global, int):
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            f"question_global must be an integer, got {type(question_global).__name__}"
        )

    if not (0 <= question_global <= MAX_QUESTION_GLOBAL):
```

```
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            f"question_global must be between 0 and {MAX_QUESTION_GLOBAL} inclusive, got {question_global}"
        )

    # Extract expected_elements via get method with None handling
    question_schema = question.get("expected_elements")
    chunk_schema = chunk_expected_elements

    # Classify types using isinstance checks in None-list-dict-invalid order
    question_type = _classify_expected_elements_type(question_schema)
    chunk_type = _classify_expected_elements_type(chunk_schema)

    # Store classification tuple before any iteration occurs
    return question_global, question_schema, chunk_schema, question_type, chunk_type


def _validate_structural_compatibility(
    question_schema: Any,  # noqa: ANN401
    chunk_schema: Any,  # noqa: ANN401
    question_type: str,
    chunk_type: str,
    question_id: str,
    correlation_id: str,
) -> None:
    """Phase 6.2: Validate structural compatibility with type homogeneity checks.

    Checks invalid types first with human-readable type names, enforces
    homogeneity allowing None compatibility, validates list length equality
    and dict key set equality with symmetric difference computation, and
    returns silently on dual-None without logging.

    Args:
        question_schema: expected_elements from question
        chunk_schema: expected_elements from chunk
        question_type: Classified type of question schema
        chunk_type: Classified type of chunk schema
        question_id: Question identifier for error messages
        correlation_id: Correlation ID for distributed tracing

    Raises:
        TypeError: If either schema has invalid type (not list, dict, or None)
        ValueError: If schemas have heterogeneous types (not allowing None),
                    list length mismatch, or dict key set mismatch

    Returns:
        None (returns silently on dual-None or successful validation)
    """
    # Check invalid types first with human-readable type names
    if question_type == "invalid":
        raise TypeError(
            f"Schema validation failure for question {question_id}: "
            f"expected_elements from question has invalid type "
            f"{type(question_schema).__name__}, expected list, dict, or None "
            f"[correlation_id={correlation_id}]"
        )
```

```

    if chunk_type == "invalid":
        raise TypeError(
            f"Schema validation failure for question {question_id}: "
            f"expected_elements from chunk has invalid type "
            f"{type(chunk_schema).__name__}, expected list, dict, or None "
            f"[correlation_id={correlation_id}]"
        )

    # Return silently on dual-None without logging
    if question_type == "none" and chunk_type == "none":
        return

    # Enforce homogeneity allowing None compatibility
    # None is compatible with any type, but non-None types must match
    if question_type not in ("none", chunk_type) and chunk_type != "none":
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            f"heterogeneous types detected (question has {question_type}, "
            f"chunk has {chunk_type}) [correlation_id={correlation_id}]"
        )

    # Validate list length equality
    if question_type == "list" and chunk_type == "list":
        question_len = len(question_schema)
        chunk_len = len(chunk_schema)
        if question_len != chunk_len:
            raise ValueError(
                f"Schema validation failure for question {question_id}: "
                f"list length mismatch (question has {question_len} elements, "
                f"chunk has {chunk_len} elements) [correlation_id={correlation_id}]"
            )

    # Validate dict key set equality with symmetric difference computation
    if question_type == "dict" and chunk_type == "dict":
        question_keys = set(question_schema.keys())
        chunk_keys = set(chunk_schema.keys())

        # Compute symmetric difference
        symmetric_diff = question_keys ^ chunk_keys

        if symmetric_diff:
            missing_in_chunk = question_keys - chunk_keys
            extra_in_chunk = chunk_keys - question_keys

            details = []
            if missing_in_chunk:
                details.append(f"missing in chunk: {sorted(missing_in_chunk)}")
            if extra_in_chunk:
                details.append(f"extra in chunk: {sorted(extra_in_chunk)}")

            raise ValueError(
                f"Schema validation failure for question {question_id}: "
                f"dict key set mismatch ({’, ’.join(details)}) "
                f"[correlation_id={correlation_id}]"
            )
```

```python


def _validate_semantic_constraints(
    question_schema: Any,  # noqa: ANN401
    chunk_schema: Any,  # noqa: ANN401
    question_type: str,
    chunk_type: str,
    provisional_task_id: str,
    question_id: str,
    chunk_id: str,
    correlation_id: str,
) -> int:
    """Phase 6.3: Validate semantic constraints and return validated element count.

    Iterates deterministically via enumerate-zip for lists and sorted keys for
    dicts, extracts type-required-minimum fields with get defaults, implements
    asymmetric required implication as not-q-or-c boolean expression, enforces
    c-min-greater-equal-q-min threshold ordering, and returns validated element
    count.

    Args:
        question_schema: expected_elements from question
        chunk_schema: expected_elements from chunk
        question_type: Classified type of question schema
        chunk_type: Classified type of chunk schema
        provisional_task_id: Task ID for error reporting
        question_id: Question identifier for error messages
        chunk_id: Chunk identifier for error messages
        correlation_id: Correlation ID for distributed tracing

    Returns:
        Validated element count (number of elements validated)

    Raises:
        ValueError: If required field implication violated or threshold ordering violated

    Semantic Constraints:
        - Asymmetric required implication: not q_required or c_required
        - Threshold ordering: c_minimum >= q_minimum
    """
    validated_count = 0

    # Iterate deterministically via enumerate-zip for lists
    if question_type == "list" and chunk_type == "list":
        for idx, (q_elem, c_elem) in enumerate(
            zip(question_schema, chunk_schema, strict=True)
        ):
            if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
                continue

            # Extract type-required-minimum fields with get defaults
            element_type = q_elem.get("type", f"element_at_index_{idx}")
            q_required = q_elem.get("required", False)
            c_required = c_elem.get("required", False)
            q_minimum = q_elem.get("minimum", 0)
            c_minimum = c_elem.get("minimum", 0)
```

```

            # Implement asymmetric required implication as not-q-or-c boolean expression
            if q_required and not c_required:
                raise ValueError(
                    f"Task {provisional_task_id}: Required field implication violation "
                    f"at index {idx}: element type '{element_type}' is required in "
                    f"question but marked as optional in chunk "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            # Enforce c-min-greater-equal-q-min threshold ordering
            if (
                isinstance(q_minimum, int | float)
                and isinstance(c_minimum, int | float)
                and c_minimum < q_minimum
            ):
                raise ValueError(
                    f"Task {provisional_task_id}: Threshold ordering violation "
                    f"at index {idx}: element type '{element_type}' has chunk "
                    f"minimum ({c_minimum}) < question minimum ({q_minimum}) "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            validated_count += 1

    # Iterate deterministically via sorted keys for dicts
    elif question_type == "dict" and chunk_type == "dict":
        common_keys = set(question_schema.keys()) & set(chunk_schema.keys())

        for key in sorted(common_keys):
            q_elem = question_schema[key]
            c_elem = chunk_schema[key]

            if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
                continue

            # Extract type-required-minimum fields with get defaults
            element_type = q_elem.get("type", key)
            q_required = q_elem.get("required", False)
            c_required = c_elem.get("required", False)
            q_minimum = q_elem.get("minimum", 0)
            c_minimum = c_elem.get("minimum", 0)

            # Implement asymmetric required implication as not-q-or-c boolean expression
            if q_required and not c_required:
                raise ValueError(
                    f"Task {provisional_task_id}: Required field implication violation "
                    f"for key '{key}': element type '{element_type}' is required in "
                    f"question but marked as optional in chunk "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            # Enforce c-min-greater-equal-q-min threshold ordering
```

```
            if (
                isinstance(q_minimum, int | float)
                and isinstance(c_minimum, int | float)
                and c_minimum < q_minimum
            ):
                raise ValueError(
                    f"Task {provisional_task_id}: Threshold ordering violation "
                    f"for key '{key}': element type '{element_type}' has chunk "
                    f"minimum ({c_minimum}) < question minimum ({q_minimum}) "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            validated_count += 1

    return validated_count


def validate_phase6_schema_compatibility(
    question: dict[str, Any],
    chunk_expected_elements: list[dict[str, Any]] | dict[str, Any] | None,
    chunk_id: str,
    policy_area_id: str,
    correlation_id: str,
) -> int:
    """Phase 6.4: Orchestrator - Coordinate complete validation pipeline.

    Invokes structural then semantic layers in sequence, captures element count
    return value, emits debug log with has_required_fields and has_minimum_thresholds
    computed via any-element-iteration, logs info warning for None chunk schema
    with non-None question schema, and allows TypeError-ValueError propagation
    to outer handler without try-except wrapping.

    This is the main entry point for Phase 6 validation, designed to integrate
    into the build_with_chunk_matrix loop after Phase 5 (signal resolution) and
    before construct_task.

    Args:
        question: Question dictionary from questionnaire
        chunk_expected_elements: expected_elements from chunk routing result
        chunk_id: Chunk identifier for logging
        policy_area_id: Policy area identifier for task ID construction
        correlation_id: Correlation ID for distributed tracing

    Returns:
        Validated element count (number of elements validated)

    Raises:
        TypeError: If either schema has invalid type (propagated from Phase 6.2)
        ValueError: If validation fails (propagated from Phase 6.1, 6.2, or 6.3)

    Integration Point:
        Called within build_with_chunk_matrix loop:
        1. After Phase 5: Signal resolution completes
        2. Before construct_task: Task construction begins
        3. No try-except wrapper: Exceptions propagate to outer handler
```

```
    Orchestration Flow:
        Phase 6.1: Extract and classify schemas
        Phase 6.2: Validate structural compatibility
        Phase 6.3: Validate semantic constraints (if both schemas non-None)
        Phase 6.4: Emit debug logs and return validated count
    """
    question_id = question.get("question_id", "UNKNOWN")

    # Phase 6.1: Classification & Extraction
    (
        question_global,
        question_schema,
        chunk_schema,
        question_type,
        chunk_type,
    ) = _extract_and_classify_schemas(question, chunk_expected_elements, question_id)

    # Construct provisional task ID for error reporting
    provisional_task_id = f"MQC-{question_global:03d}_{policy_area_id}"

    # Phase 6.2: Structural Validation
    _validate_structural_compatibility(
        question_schema,
        chunk_schema,
        question_type,
        chunk_type,
        question_id,
        correlation_id,
    )

    # Phase 6.3: Semantic Validation (if both schemas non-None)
    validated_count = 0
    if question_schema is not None and chunk_schema is not None:
        validated_count = _validate_semantic_constraints(
            question_schema,
            chunk_schema,
            question_type,
            chunk_type,
            provisional_task_id,
            question_id,
            chunk_id,
            correlation_id,
        )

    # Phase 6.4: Emit debug log with has_required_fields and has_minimum_thresholds
    # Compute via any-element-iteration
    has_required_fields = False
    has_minimum_thresholds = False

    if question_schema is not None:
        if isinstance(question_schema, list):
            has_required_fields = any(
                elem.get("required", False)
                for elem in question_schema
                if isinstance(elem, dict)
```

```
            )
            has_minimum_thresholds = any(
                "minimum" in elem for elem in question_schema if isinstance(elem, dict)
            )
        elif isinstance(question_schema, dict):
            has_required_fields = any(
                elem.get("required", False)
                for elem in question_schema.values()
                if isinstance(elem, dict)
            )
            has_minimum_thresholds = any(
                "minimum" in elem
                for elem in question_schema.values()
                if isinstance(elem, dict)
            )

    logger.debug(
        f"Phase 6 validation complete: question_id={question_id}, "
        f"chunk_id={chunk_id}, provisional_task_id={provisional_task_id}, "
        f"validated_count={validated_count}, "
        f"has_required_fields={has_required_fields}, "
        f"has_minimum_thresholds={has_minimum_thresholds}, "
        f"question_type={question_type}, chunk_type={chunk_type}, "
        f"correlation_id={correlation_id}"
    )

    # Log info warning for None chunk schema with non-None question schema
    if question_schema is not None and chunk_schema is None:
        logger.info(
            f"Schema asymmetry detected: question_id={question_id}, "
            f"chunk_id={chunk_id}, question_schema_type={question_type}, "
            f"chunk_schema_type=none, message='Question specifies required elements "
            f"but chunk provides no schema', "
            f"validation_status='compatible_via_constraint_relaxation', "
            f"correlation_id={correlation_id}"
        )

    return validated_count


__all__ = [
    "validate_phase6_schema_compatibility",
    "_extract_and_classify_schemas",
    "_validate_structural_compatibility",
    "_validate_semantic_constraints",
    "_classify_expected_elements_type",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/precision_tracking.py
================================================================================

"""
Precision Improvement Tracking for Context Filtering
```

```
====================================================

Enhanced validation and comprehensive stats tracking for the 60% precision
improvement target from filter_patterns_by_context integration.

This module provides:
1. Enhanced get_patterns_for_context() wrapper with validation
2. Detailed validation status tracking
3. Comprehensive logging and metrics
4. Target achievement verification

Usage:
    >>> from farfan_pipeline.core.orchestrator.precision_tracking import (
    ...     get_patterns_with_validation
    ... )
    >>> patterns, stats = get_patterns_with_validation(
    ...     enriched_pack, document_context
    ... )
    >>> assert stats['integration_validated']
    >>> assert stats['target_achieved']

Author: F.A.R.F.A.N Pipeline
Date: 2025-12-03
"""

from datetime import datetime, timezone
from typing import Any

try:
    import structlog

    logger = structlog.get_logger(__name__)
except ImportError:
    import logging

    logger = logging.getLogger(__name__)


PRECISION_TARGET_THRESHOLD = 0.55


def get_patterns_with_validation(
    enriched_pack: Any,
    document_context: dict[str, Any],
    track_precision_improvement: bool = True,
) -> tuple[list[dict[str, Any]], dict[str, Any]]:
    """
    Enhanced wrapper for get_patterns_for_context() with comprehensive validation.

    This function wraps EnrichedSignalPack.get_patterns_for_context() and adds:
    - Pre-filtering validation
    - Post-filtering verification
    - Integration status checking
    - Target achievement tracking
    - Detailed logging

```

```
    Args:
        enriched_pack: EnrichedSignalPack instance
        document_context: Document context dict
        track_precision_improvement: Enable precision tracking

    Returns:
        Tuple of (filtered_patterns, comprehensive_stats) with enhanced fields:
            - validation_timestamp: ISO timestamp
            - validation_details: Detailed validation info
            - target_achieved: Boolean for 60% target
            - validation_status: Status string
            - target_status: Status string
            - pre_filter_count: Patterns before filtering
            - post_filter_count: Patterns after filtering
            - filtering_successful: Boolean validation

    Example:
        >>> enriched = create_enriched_signal_pack(base_pack)
        >>> context = create_document_context(section='budget', chapter=3)
        >>> patterns, stats = get_patterns_with_validation(enriched, context)
        >>> print(f"Validation: {stats['validation_status']}")
        >>> print(f"Target: {stats['target_status']}")
        >>> assert stats['integration_validated']
        >>> assert stats['target_achieved']
    """
    if not isinstance(document_context, dict):
        logger.warning(
            "invalid_document_context_type",
            context_type=type(document_context).__name__,
            expected="dict",
        )
        document_context = {}

    validation_timestamp = datetime.now(timezone.utc).isoformat()

    pre_filter_count = (
        len(enriched_pack.patterns) if hasattr(enriched_pack, "patterns") else 0
    )

    filtered, base_stats = enriched_pack.get_patterns_for_context(
        document_context, track_precision_improvement=track_precision_improvement
    )

    post_filter_count = len(filtered)

    validation_details = {
        "filter_function_called": True,
        "pre_filter_count": pre_filter_count,
        "post_filter_count": post_filter_count,
        "context_fields": list(document_context.keys()),
        "context_field_count": len(document_context),
        "filtering_successful": post_filter_count <= pre_filter_count,
        "patterns_reduced": pre_filter_count - post_filter_count,
        "reduction_percentage": (
            (pre_filter_count - post_filter_count) / pre_filter_count * 100
            if pre_filter_count > 0
```

```
            else 0.0
        ),
    }

    enhanced_stats = {**base_stats}
    enhanced_stats["validation_timestamp"] = validation_timestamp
    enhanced_stats["validation_details"] = validation_details
    enhanced_stats["pre_filter_count"] = pre_filter_count
    enhanced_stats["post_filter_count"] = post_filter_count
    enhanced_stats["filtering_successful"] = validation_details["filtering_successful"]

    if track_precision_improvement:
        integration_validated = base_stats.get("integration_validated", False)
        false_positive_reduction = base_stats.get("false_positive_reduction", 0.0)
        target_achieved = false_positive_reduction >= PRECISION_TARGET_THRESHOLD

        enhanced_stats["target_achieved"] = target_achieved

        if integration_validated:
            enhanced_stats["validation_status"] = "VALIDATED"
            validation_message = "â\234\223 filter_patterns_by_context integration VALIDATED"
        else:
            enhanced_stats["validation_status"] = "NOT_VALIDATED"
            validation_message = (
                "â\234\227 filter_patterns_by_context integration NOT validated"
            )

        target_status = "ACHIEVED" if target_achieved else "NOT_MET"
        enhanced_stats["target_status"] = target_status

        if not validation_details["filtering_successful"]:
            logger.error(
                "context_filtering_validation_failed",
                pre_filter_count=pre_filter_count,
                post_filter_count=post_filter_count,
                reason="filtered_count_exceeds_original",
            )
            enhanced_stats["integration_validated"] = False
            enhanced_stats["validation_status"] = "FAILED"

        logger.info(
            "enhanced_context_filtering_validation",
            pre_filter_count=pre_filter_count,
            post_filter_count=post_filter_count,
            patterns_reduced=validation_details["patterns_reduced"],
            reduction_percentage=f"{validation_details['reduction_percentage']:.1f}%",
            filter_rate=f"{base_stats.get('filter_rate', 0.0):.1%}",
            precision_improvement=f"{base_stats.get('precision_improvement', 0.0):.1%}",
            false_positive_reduction=f"{false_positive_reduction:.1%}",
            integration_validated=integration_validated,
            validation_status=enhanced_stats["validation_status"],
            target_achieved=target_achieved,
            target_status=target_status,
            validation_message=validation_message,
            validation_timestamp=validation_timestamp,
        )
```

```
        if target_achieved:
            logger.info(
                "precision_target_achieved",
                false_positive_reduction=f"{false_positive_reduction:.1%}",
                target_threshold=f"{PRECISION_TARGET_THRESHOLD:.1%}",
                message="â\234\223 60% precision improvement target ACHIEVED",
            )
        else:
            logger.warning(
                "precision_target_not_met",
                false_positive_reduction=f"{false_positive_reduction:.1%}",
                target_threshold=f"{PRECISION_TARGET_THRESHOLD:.1%}",
                shortfall=f"{(PRECISION_TARGET_THRESHOLD - false_positive_reduction):.1%}",
                message="â\234\227 60% precision improvement target NOT met",
            )
    else:
        enhanced_stats["target_achieved"] = False
        enhanced_stats["validation_status"] = "TRACKING_DISABLED"
        enhanced_stats["target_status"] = "UNKNOWN"
        logger.debug("context_filtering_applied_without_tracking", **validation_details)

    return filtered, enhanced_stats


def validate_filter_integration(
    enriched_pack: Any, test_contexts: list[dict[str, Any]] | None = None
) -> dict[str, Any]:
    """
    Comprehensive validation of filter_patterns_by_context integration.

    Tests the filtering functionality across multiple contexts and validates:
    - Integration is working correctly
    - Patterns are being filtered
    - 60% target is achievable
    - No errors occur during filtering

    Args:
        enriched_pack: EnrichedSignalPack instance to test
        test_contexts: Optional list of test contexts. If None, uses defaults.

    Returns:
        Validation report dict with:
            - total_tests: Number of contexts tested
            - successful_tests: Tests that completed without error
            - integration_validated: Overall integration status
            - target_achieved_count: Number of tests achieving 60% target
            - target_achievement_rate: Percentage achieving target
            - average_filter_rate: Average pattern reduction
            - average_fp_reduction: Average false positive reduction
            - validation_summary: Human-readable summary

    Example:
        >>> enriched = create_enriched_signal_pack(base_pack)
        >>> report = validate_filter_integration(enriched)
        >>> print(report['validation_summary'])
```

```
        >>> assert report['integration_validated']
        >>> assert report['target_achievement_rate'] > 0.5
    """
    if test_contexts is None:
        test_contexts = [
            {},
            {"section": "budget"},
            {"section": "indicators", "chapter": 5},
            {"section": "financial", "chapter": 2, "page": 10},
            {"policy_area": "economic_development"},
        ]

    results = []
    errors = []

    for idx, context in enumerate(test_contexts):
        try:
            patterns, stats = get_patterns_with_validation(
                enriched_pack, context, track_precision_improvement=True
            )
            results.append(stats)
        except Exception as e:
            logger.error(
                "filter_validation_test_failed",
                test_index=idx,
                context=context,
                error=str(e),
                error_type=type(e).__name__,
            )
            errors.append(
                {
                    "test_index": idx,
                    "context": context,
                    "error": str(e),
                    "error_type": type(e).__name__,
                }
            )

    total_tests = len(test_contexts)
    successful_tests = len(results)
    failed_tests = len(errors)

    if successful_tests == 0:
        return {
            "total_tests": total_tests,
            "successful_tests": 0,
            "failed_tests": failed_tests,
            "integration_validated": False,
            "target_achieved_count": 0,
            "target_achievement_rate": 0.0,
            "average_filter_rate": 0.0,
            "average_fp_reduction": 0.0,
            "errors": errors,
            "validation_summary": "â\234\227 ALL TESTS FAILED - Integration NOT working",
        }

```

```
    integration_validated_count = sum(
        1 for r in results if r.get("integration_validated", False)
    )
    target_achieved_count = sum(1 for r in results if r.get("target_achieved", False))

    average_filter_rate = (
        sum(r.get("filter_rate", 0.0) for r in results) / successful_tests
    )
    average_fp_reduction = (
        sum(r.get("false_positive_reduction", 0.0) for r in results) / successful_tests
    )

    integration_rate = integration_validated_count / successful_tests
    target_achievement_rate = target_achieved_count / successful_tests

    overall_integration_validated = integration_rate >= 0.8

    validation_summary = (
        f"Filter Integration Validation Report:\n"
        f"  Tests: {successful_tests}/{total_tests} successful ({failed_tests} failed)\n"
        f"  Integration validated: {integration_validated_count}/{successful_tests} "
        f"({integration_rate:.0%})\n"
        f"  60% target achieved: {target_achieved_count}/{successful_tests} "
        f"({target_achievement_rate:.0%})\n"
        f"  Average filter rate: {average_filter_rate:.1%}\n"
        f"  Average FP reduction: {average_fp_reduction:.1%}\n"
        f"  Overall status: "
        f"{'â\234\223 VALIDATED' if overall_integration_validated else 'â\234\227 NOT VALIDATED'}\n"
        f"  Target status: "
        f"{'â\234\223 ACHIEVABLE' if target_achievement_rate > 0 else 'â\234\227 NOT ACHIEVABLE'}"
    )

    report = {
        "total_tests": total_tests,
        "successful_tests": successful_tests,
        "failed_tests": failed_tests,
        "integration_validated": overall_integration_validated,
        "integration_validated_count": integration_validated_count,
        "integration_rate": integration_rate,
        "target_achieved_count": target_achieved_count,
        "target_achievement_rate": target_achievement_rate,
        "average_filter_rate": average_filter_rate,
        "average_fp_reduction": average_fp_reduction,
        "max_fp_reduction": (
            max(r.get("false_positive_reduction", 0.0) for r in results)
            if results
            else 0.0
        ),
        "min_fp_reduction": (
            min(r.get("false_positive_reduction", 0.0) for r in results)
            if results
            else 0.0
        ),
        "errors": errors,
        "validation_summary": validation_summary,
        "all_results": results,
```

```
    }

    logger.info(
        "filter_integration_validation_complete",
        total_tests=total_tests,
        successful_tests=successful_tests,
        failed_tests=failed_tests,
        integration_validated=overall_integration_validated,
        target_achievement_rate=f"{target_achievement_rate:.0%}",
        summary=validation_summary,
    )

    return report


def create_precision_tracking_session(
    enriched_pack: Any, session_id: str | None = None
) -> dict[str, Any]:
    """
    Create a precision tracking session for continuous monitoring.

    This creates a session object that tracks multiple measurements over time,
    useful for monitoring precision improvement during production analysis.

    Args:
        enriched_pack: EnrichedSignalPack instance
        session_id: Optional session identifier

    Returns:
        Session object with tracking state and methods

    Example:
        >>> session = create_precision_tracking_session(enriched_pack, "prod_001")
        >>> # Use session throughout analysis...
        >>> results = finalize_precision_tracking_session(session)
    """
    from datetime import datetime, timezone
    from uuid import uuid4

    if session_id is None:
        session_id = f"precision_session_{uuid4().hex[:8]}"

    session = {
        "session_id": session_id,
        "start_timestamp": datetime.now(timezone.utc).isoformat(),
        "enriched_pack": enriched_pack,
        "measurements": [],
        "measurement_count": 0,
        "contexts_tested": [],
        "cumulative_stats": {
            "total_patterns_processed": 0,
            "total_patterns_filtered": 0,
            "total_filtering_time_ms": 0.0,
        },
        "status": "ACTIVE",
    }
```

```

    logger.info(
        "precision_tracking_session_created",
        session_id=session_id,
        start_timestamp=session["start_timestamp"],
    )

    return session


def add_measurement_to_session(
    session: dict[str, Any],
    document_context: dict[str, Any],
    track_precision: bool = True,
) -> tuple[list[dict[str, Any]], dict[str, Any]]:
    """
    Add a measurement to an active precision tracking session.

    Args:
        session: Active session from create_precision_tracking_session
        document_context: Document context for this measurement
        track_precision: Enable precision tracking

    Returns:
        Tuple of (filtered_patterns, stats) from get_patterns_for_context

    Example:
        >>> session = create_precision_tracking_session(enriched_pack)
        >>> for context in contexts:
        ...     patterns, stats = add_measurement_to_session(session, context)
    """
    if session["status"] != "ACTIVE":
        logger.warning(
            "measurement_to_inactive_session",
            session_id=session["session_id"],
            status=session["status"],
        )

    enriched_pack = session["enriched_pack"]
    patterns, stats = get_patterns_with_validation(
        enriched_pack, document_context, track_precision
    )

    session["measurements"].append(stats)
    session["measurement_count"] += 1
    session["contexts_tested"].append(document_context)

    session["cumulative_stats"]["total_patterns_processed"] += stats.get(
        "total_patterns", 0
    )
    session["cumulative_stats"]["total_patterns_filtered"] += stats.get(
        "total_patterns", 0
    ) - stats.get("passed", 0)
    session["cumulative_stats"]["total_filtering_time_ms"] += stats.get(
        "filtering_duration_ms", 0.0
    )
```

```

    return patterns, stats


def finalize_precision_tracking_session(
    session: dict[str, Any], generate_full_report: bool = True
) -> dict[str, Any]:
    """
    Finalize a precision tracking session and generate summary.

    Args:
        session: Active session to finalize
        generate_full_report: Include full detailed report

    Returns:
        Finalized session report with comprehensive metrics

    Example:
        >>> session = create_precision_tracking_session(enriched_pack)
        >>> # ... add measurements ...
        >>> results = finalize_precision_tracking_session(session)
        >>> print(results['summary'])
    """
    from datetime import datetime, timezone

    from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
        generate_precision_improvement_report,
    )

    end_timestamp = datetime.now(timezone.utc).isoformat()
    session["end_timestamp"] = end_timestamp
    session["status"] = "FINALIZED"

    if not session["measurements"]:
        return {
            "session_id": session["session_id"],
            "status": "FINALIZED",
            "measurement_count": 0,
            "summary": "No measurements recorded",
        }

    full_report = None
    if generate_full_report:
        full_report = generate_precision_improvement_report(
            session["measurements"], include_detailed_breakdown=True
        )

    session_summary = {
        "session_id": session["session_id"],
        "start_timestamp": session["start_timestamp"],
        "end_timestamp": end_timestamp,
        "status": session["status"],
        "measurement_count": session["measurement_count"],
        "cumulative_stats": session["cumulative_stats"],
        "contexts_tested_count": len(session["contexts_tested"]),
    }
```

```
    if full_report:
        session_summary["aggregate_report"] = full_report
        session_summary["summary"] = full_report["summary"]
        session_summary["target_achievement_rate"] = full_report[
            "target_achievement_rate"
        ]
        session_summary["integration_validated"] = full_report["validation_rate"] >= 0.8
        session_summary["validation_health"] = full_report["validation_health"]

    logger.info(
        "precision_tracking_session_finalized",
        session_id=session["session_id"],
        measurement_count=session["measurement_count"],
        total_patterns_processed=session["cumulative_stats"][
            "total_patterns_processed"
        ],
        total_filtering_time_ms=session["cumulative_stats"]["total_filtering_time_ms"],
        target_achievement_rate=(session_summary.get("target_achievement_rate", 0.0)),
    )

    return session_summary


def compare_precision_across_policy_areas(
    policy_area_packs: dict[str, Any], test_contexts: list[dict[str, Any]] | None = None
) -> dict[str, Any]:
    """
    Compare precision improvement across multiple policy areas.

    Useful for identifying which policy areas achieve the 60% target and which need improvement.

    Args:
        policy_area_packs: Dict mapping policy_area_id to EnrichedSignalPack
        test_contexts: Optional test contexts (uses defaults if None)

    Returns:
        Comparison report with per-area metrics and rankings

    Example:
        >>> packs = {
        ...     "PA01": create_enriched_signal_pack(base_pack_01),
        ...     "PA02": create_enriched_signal_pack(base_pack_02),
        ... }
        >>> comparison = compare_precision_across_policy_areas(packs)
        >>> print(comparison['rankings']['by_target_achievement'])
    """
    from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
        generate_precision_improvement_report,
    )

    if test_contexts is None:
        test_contexts = [
            {},
            {"section": "budget"},
            {"section": "indicators"},
```

```
            {"section": "financial"},
        ]

    area_results = {}

    for policy_area_id, enriched_pack in policy_area_packs.items():
        measurements = []
        for context in test_contexts:
            try:
                _, stats = enriched_pack.get_patterns_for_context(
                    context, track_precision_improvement=True
                )
                measurements.append(stats)
            except Exception as e:
                logger.error(
                    "policy_area_precision_test_failed",
                    policy_area=policy_area_id,
                    context=context,
                    error=str(e),
                )

        if measurements:
            report = generate_precision_improvement_report(
                measurements, include_detailed_breakdown=False
            )
            area_results[policy_area_id] = report

    if not area_results:
        return {
            "policy_areas_tested": 0,
            "comparison_status": "FAILED",
            "message": "No successful measurements",
        }

    rankings = {
        "by_target_achievement": sorted(
            area_results.items(),
            key=lambda x: x[1]["target_achievement_rate"],
            reverse=True,
        ),
        "by_avg_fp_reduction": sorted(
            area_results.items(),
            key=lambda x: x[1]["avg_false_positive_reduction"],
            reverse=True,
        ),
        "by_validation_rate": sorted(
            area_results.items(), key=lambda x: x[1]["validation_rate"], reverse=True
        ),
    }

    best_performer = rankings["by_target_achievement"][0]
    worst_performer = rankings["by_target_achievement"][-1]

    areas_meeting_target = sum(
        1
        for _, report in area_results.items()
```

```
        if report["max_false_positive_reduction"] >= PRECISION_TARGET_THRESHOLD
    )

    comparison_summary = (
        f"Policy Area Precision Comparison:\n"
        f"  Areas tested: {len(area_results)}\n"
        f"  Areas meeting 60% target: {areas_meeting_target}/{len(area_results)}\n"
        f"  Best performer: {best_performer[0]} "
        f"({100*best_performer[1]['target_achievement_rate']:.0f}% target achievement)\n"
        f"  Worst performer: {worst_performer[0]} "
        f"({100*worst_performer[1]['target_achievement_rate']:.0f}% target achievement)\n"
        f"  Overall status: "
        f"{'â\234\223 GOOD' if areas_meeting_target >= len(area_results) * 0.7 else 'â\234\227 NEEDS IMPROVEMENT'}"
    )

    return {
        "policy_areas_tested": len(area_results),
        "areas_meeting_target": areas_meeting_target,
        "target_achievement_coverage": areas_meeting_target / len(area_results),
        "rankings": rankings,
        "best_performer": {
            "policy_area": best_performer[0],
            "metrics": best_performer[1],
        },
        "worst_performer": {
            "policy_area": worst_performer[0],
            "metrics": worst_performer[1],
        },
        "all_results": area_results,
        "comparison_summary": comparison_summary,
    }


def export_precision_metrics_for_monitoring(
    measurements: list[dict[str, Any]], output_format: str = "json"
) -> str | dict[str, Any]:
    """
    Export precision metrics in format suitable for external monitoring systems.

    Args:
        measurements: List of stats dicts from get_patterns_for_context
        output_format: 'json', 'prometheus', or 'datadog'

    Returns:
        Formatted metrics string or dict

    Example:
        >>> measurements = [...]
        >>> metrics = export_precision_metrics_for_monitoring(measurements, 'json')
    """
    import json
    from datetime import datetime, timezone

    timestamp = datetime.now(timezone.utc).isoformat()

    if not measurements:
```

```
        if output_format == "json":
            return json.dumps({"error": "No measurements", "timestamp": timestamp})
        return ""

    total = len(measurements)
    meets_target = sum(
        1
        for m in measurements
        if m.get("false_positive_reduction", 0.0) >= PRECISION_TARGET_THRESHOLD
    )
    validated = sum(1 for m in measurements if m.get("integration_validated", False))

    avg_fp_reduction = (
        sum(m.get("false_positive_reduction", 0.0) for m in measurements) / total
    )
    avg_filter_rate = sum(m.get("filter_rate", 0.0) for m in measurements) / total

    if output_format == "json":
        return json.dumps(
            {
                "timestamp": timestamp,
                "measurement_count": total,
                "target_achievement_count": meets_target,
                "target_achievement_rate": meets_target / total,
                "integration_validated_count": validated,
                "integration_validation_rate": validated / total,
                "avg_false_positive_reduction": avg_fp_reduction,
                "avg_filter_rate": avg_filter_rate,
                "meets_60_percent_target": meets_target / total >= 0.5,
            },
            indent=2,
        )

    elif output_format == "prometheus":
        lines = [
            "# HELP precision_target_achievement_rate Rate of measurements meeting 60% target",
            "# TYPE precision_target_achievement_rate gauge",
            f"precision_target_achievement_rate {meets_target / total}",
            "# HELP precision_avg_fp_reduction Average false positive reduction",
            "# TYPE precision_avg_fp_reduction gauge",
            f"precision_avg_fp_reduction {avg_fp_reduction}",
            "# HELP precision_measurement_count Total measurements",
            "# TYPE precision_measurement_count counter",
            f"precision_measurement_count {total}",
        ]
        return "\n".join(lines)

    elif output_format == "datadog":
        return json.dumps(
            [
                {
                    "metric": "farfan.precision.target_achievement_rate",
                    "points": [
                        [
                            int(datetime.now(timezone.utc).timestamp()),
                            meets_target / total,
```

```
                        ]
                    ],
                    "type": "gauge",
                    "tags": ["component:context_filtering"],
                },
                {
                    "metric": "farfan.precision.avg_fp_reduction",
                    "points": [
                        [int(datetime.now(timezone.utc).timestamp()), avg_fp_reduction]
                    ],
                    "type": "gauge",
                    "tags": ["component:context_filtering"],
                },
                {
                    "metric": "farfan.precision.measurement_count",
                    "points": [[int(datetime.now(timezone.utc).timestamp()), total]],
                    "type": "count",
                    "tags": ["component:context_filtering"],
                },
            ],
            indent=2,
        )

    return ""


__all__ = [
    "get_patterns_with_validation",
    "validate_filter_integration",
    "create_precision_tracking_session",
    "add_measurement_to_session",
    "finalize_precision_tracking_session",
    "compare_precision_across_policy_areas",
    "export_precision_metrics_for_monitoring",
    "PRECISION_TARGET_THRESHOLD",
]



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/resource_alerts.py
================================================================================

"""Resource Pressure Alerting and Observability.

Provides comprehensive alerting and monitoring for resource management:
- Structured logging for resource events
- Alert thresholds and notifications
- Integration with external monitoring systems
- Historical trend analysis
"""

from __future__ import annotations

import json
import logging
```

```
from collections import defaultdict
from datetime import datetime, timedelta
from enum import Enum
from typing import Any, Callable

from farfan_pipeline.core.orchestrator.resource_manager import (
    ResourcePressureEvent,
    ResourcePressureLevel,
)

logger = logging.getLogger(__name__)


class AlertSeverity(Enum):
    """Alert severity levels."""

    INFO = "info"
    WARNING = "warning"
    ERROR = "error"
    CRITICAL = "critical"


class AlertChannel(Enum):
    """Alert delivery channels."""

    LOG = "log"
    WEBHOOK = "webhook"
    SIGNAL = "signal"
    STDOUT = "stdout"


class ResourceAlert:
    """Individual resource alert."""

    def __init__(
        self,
        severity: AlertSeverity,
        title: str,
        message: str,
        event: ResourcePressureEvent,
        metadata: dict[str, Any] | None = None,
    ) -> None:
        self.severity = severity
        self.title = title
        self.message = message
        self.event = event
        self.metadata = metadata or {}
        self.timestamp = datetime.utcnow()
        self.alert_id = f"alert_{self.timestamp.isoformat()}_{id(self)}"

    def to_dict(self) -> dict[str, Any]:
        """Convert alert to dictionary."""
        return {
            "alert_id": self.alert_id,
            "timestamp": self.timestamp.isoformat(),
            "severity": self.severity.value,
```

```
            "title": self.title,
            "message": self.message,
            "event": {
                "timestamp": self.event.timestamp.isoformat(),
                "pressure_level": self.event.pressure_level.value,
                "cpu_percent": self.event.cpu_percent,
                "memory_mb": self.event.memory_mb,
                "memory_percent": self.event.memory_percent,
                "worker_count": self.event.worker_count,
                "active_executors": self.event.active_executors,
                "degradation_applied": self.event.degradation_applied,
                "circuit_breakers_open": self.event.circuit_breakers_open,
            },
            "metadata": self.metadata,
        }

    def to_json(self) -> str:
        """Convert alert to JSON string."""
        return json.dumps(self.to_dict(), indent=2)


class AlertThresholds:
    """Configurable alert thresholds."""

    def __init__(
        self,
        memory_warning_percent: float = 75.0,
        memory_critical_percent: float = 85.0,
        cpu_warning_percent: float = 75.0,
        cpu_critical_percent: float = 85.0,
        circuit_breaker_warning_count: int = 3,
        degradation_critical_count: int = 3,
    ) -> None:
        self.memory_warning_percent = memory_warning_percent
        self.memory_critical_percent = memory_critical_percent
        self.cpu_warning_percent = cpu_warning_percent
        self.cpu_critical_percent = cpu_critical_percent
        self.circuit_breaker_warning_count = circuit_breaker_warning_count
        self.degradation_critical_count = degradation_critical_count


class ResourceAlertManager:
    """Manages resource pressure alerts and notifications."""

    def __init__(
        self,
        thresholds: AlertThresholds | None = None,
        channels: list[AlertChannel] | None = None,
        webhook_url: str | None = None,
        signal_callback: Callable[[ResourceAlert], None] | None = None,
    ) -> None:
        self.thresholds = thresholds or AlertThresholds()
        self.channels = channels or [AlertChannel.LOG]
        self.webhook_url = webhook_url
        self.signal_callback = signal_callback

```

```
        self.alert_history: list[ResourceAlert] = []
        self.alert_counts: dict[str, int] = defaultdict(int)
        self.suppressed_alerts: set[str] = set()
        self.last_alert_times: dict[str, datetime] = {}

    def process_event(self, event: ResourcePressureEvent) -> list[ResourceAlert]:
        """Process resource pressure event and generate alerts."""
        alerts: list[ResourceAlert] = []

        memory_alert = self._check_memory_threshold(event)
        if memory_alert:
            alerts.append(memory_alert)

        cpu_alert = self._check_cpu_threshold(event)
        if cpu_alert:
            alerts.append(cpu_alert)

        pressure_alert = self._check_pressure_level(event)
        if pressure_alert:
            alerts.append(pressure_alert)

        circuit_breaker_alert = self._check_circuit_breakers(event)
        if circuit_breaker_alert:
            alerts.append(circuit_breaker_alert)

        degradation_alert = self._check_degradation(event)
        if degradation_alert:
            alerts.append(degradation_alert)

        for alert in alerts:
            self._dispatch_alert(alert)
            self.alert_history.append(alert)
            self.alert_counts[alert.severity.value] += 1

        return alerts

    def _check_memory_threshold(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if memory usage exceeds thresholds."""
        if event.memory_percent >= self.thresholds.memory_critical_percent:
            return ResourceAlert(
                severity=AlertSeverity.CRITICAL,
                title="Critical Memory Usage",
                message=f"Memory usage at {event.memory_percent:.1f}% "
                f"({event.memory_mb:.1f} MB)",
                event=event,
                metadata={"threshold": self.thresholds.memory_critical_percent},
            )

        if event.memory_percent >= self.thresholds.memory_warning_percent:
            if self._should_alert("memory_warning", minutes=5):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="High Memory Usage",
                    message=f"Memory usage at {event.memory_percent:.1f}% "
```

```
                    f"({event.memory_mb:.1f} MB)",
                    event=event,
                    metadata={"threshold": self.thresholds.memory_warning_percent},
                )

        return None

    def _check_cpu_threshold(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if CPU usage exceeds thresholds."""
        if event.cpu_percent >= self.thresholds.cpu_critical_percent:
            return ResourceAlert(
                severity=AlertSeverity.CRITICAL,
                title="Critical CPU Usage",
                message=f"CPU usage at {event.cpu_percent:.1f}%",
                event=event,
                metadata={"threshold": self.thresholds.cpu_critical_percent},
            )

        if event.cpu_percent >= self.thresholds.cpu_warning_percent:
            if self._should_alert("cpu_warning", minutes=5):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="High CPU Usage",
                    message=f"CPU usage at {event.cpu_percent:.1f}%",
                    event=event,
                    metadata={"threshold": self.thresholds.cpu_warning_percent},
                )

        return None

    def _check_pressure_level(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if pressure level warrants alert."""
        if event.pressure_level == ResourcePressureLevel.EMERGENCY:
            return ResourceAlert(
                severity=AlertSeverity.CRITICAL,
                title="Emergency Resource Pressure",
                message="System under emergency resource pressure",
                event=event,
            )

        if event.pressure_level == ResourcePressureLevel.CRITICAL:
            if self._should_alert("pressure_critical", minutes=2):
                return ResourceAlert(
                    severity=AlertSeverity.ERROR,
                    title="Critical Resource Pressure",
                    message="System under critical resource pressure",
                    event=event,
                )

        if event.pressure_level == ResourcePressureLevel.HIGH:
            if self._should_alert("pressure_high", minutes=10):
                return ResourceAlert(
```

```
                    severity=AlertSeverity.WARNING,
                    title="High Resource Pressure",
                    message="System experiencing high resource pressure",
                    event=event,
                )

        return None

    def _check_circuit_breakers(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if circuit breakers warrant alert."""
        open_count = len(event.circuit_breakers_open)

        if open_count >= self.thresholds.circuit_breaker_warning_count:
            return ResourceAlert(
                severity=AlertSeverity.ERROR,
                title="Multiple Circuit Breakers Open",
                message=f"{open_count} circuit breakers are open: "
                f"{’, ’.join(event.circuit_breakers_open)}",
                event=event,
                metadata={
                    "open_count": open_count,
                    "executors": event.circuit_breakers_open,
                },
            )

        if open_count > 0:
            if self._should_alert("circuit_breaker", minutes=5):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="Circuit Breaker Opened",
                    message=f"Circuit breakers open for: "
                    f"{’, ’.join(event.circuit_breakers_open)}",
                    event=event,
                    metadata={"executors": event.circuit_breakers_open},
                )

        return None

    def _check_degradation(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if degradation strategies warrant alert."""
        degradation_count = len(event.degradation_applied)

        if degradation_count >= self.thresholds.degradation_critical_count:
            return ResourceAlert(
                severity=AlertSeverity.ERROR,
                title="Multiple Degradation Strategies Active",
                message=f"{degradation_count} degradation strategies applied: "
                f"{’, ’.join(event.degradation_applied)}",
                event=event,
                metadata={
                    "count": degradation_count,
                    "strategies": event.degradation_applied,
```

```
                },
            )

        if degradation_count > 0:
            if self._should_alert("degradation", minutes=10):
                return ResourceAlert(
                    severity=AlertSeverity.INFO,
                    title="Degradation Strategies Active",
                    message=f"Active degradation: "
                    f"{', '.join(event.degradation_applied)}",
                    event=event,
                    metadata={"strategies": event.degradation_applied},
                )

        return None

    def _should_alert(self, alert_type: str, minutes: int = 5) -> bool:
        """Check if alert should be sent (with rate limiting)."""
        now = datetime.utcnow()
        last_time = self.last_alert_times.get(alert_type)

        if not last_time:
            self.last_alert_times[alert_type] = now
            return True

        elapsed = (now - last_time).total_seconds() / 60
        if elapsed >= minutes:
            self.last_alert_times[alert_type] = now
            return True

        return False

    def _dispatch_alert(self, alert: ResourceAlert) -> None:
        """Dispatch alert to configured channels."""
        for channel in self.channels:
            try:
                if channel == AlertChannel.LOG:
                    self._log_alert(alert)
                elif channel == AlertChannel.WEBHOOK:
                    self._send_webhook(alert)
                elif channel == AlertChannel.SIGNAL:
                    self._send_signal(alert)
                elif channel == AlertChannel.STDOUT:
                    self._print_alert(alert)
            except Exception as exc:
                logger.error(
                    f"Failed to dispatch alert to {channel.value}: {exc}"
                )

    def _log_alert(self, alert: ResourceAlert) -> None:
        """Log alert with appropriate severity."""
        extra = {
            "alert_id": alert.alert_id,
            "alert_severity": alert.severity.value,
            "pressure_level": alert.event.pressure_level.value,
            "cpu_percent": alert.event.cpu_percent,
```

```
            "memory_mb": alert.event.memory_mb,
        }

        if alert.severity == AlertSeverity.CRITICAL:
            logger.critical(f"{alert.title}: {alert.message}", extra=extra)
        elif alert.severity == AlertSeverity.ERROR:
            logger.error(f"{alert.title}: {alert.message}", extra=extra)
        elif alert.severity == AlertSeverity.WARNING:
            logger.warning(f"{alert.title}: {alert.message}", extra=extra)
        else:
            logger.info(f"{alert.title}: {alert.message}", extra=extra)

    def _send_webhook(self, alert: ResourceAlert) -> None:
        """Send alert via webhook."""
        if not self.webhook_url:
            return

        try:
            import requests

            requests.post(
                self.webhook_url,
                json=alert.to_dict(),
                timeout=5,
            )
        except Exception as exc:
            logger.error(f"Webhook alert failed: {exc}")

    def _send_signal(self, alert: ResourceAlert) -> None:
        """Send alert via signal callback."""
        if not self.signal_callback:
            return

        try:
            self.signal_callback(alert)
        except Exception as exc:
            logger.error(f"Signal callback failed: {exc}")

    def _print_alert(self, alert: ResourceAlert) -> None:
        """Print alert to stdout."""
        severity_colors = {
            AlertSeverity.INFO: "\033[94m",
            AlertSeverity.WARNING: "\033[93m",
            AlertSeverity.ERROR: "\033[91m",
            AlertSeverity.CRITICAL: "\033[95m",
        }
        reset = "\033[0m"

        color = severity_colors.get(alert.severity, reset)
        print(
            f"{color}[{alert.severity.value.upper()}] {alert.title}: "
            f"{alert.message}{reset}"
        )

    def get_alert_summary(self) -> dict[str, Any]:
        """Get summary of alert history."""
```

```
        now = datetime.utcnow()
        hour_ago = now - timedelta(hours=1)
        day_ago = now - timedelta(days=1)

        recent_alerts = [
            alert for alert in self.alert_history if alert.timestamp >= hour_ago
        ]

        daily_alerts = [
            alert for alert in self.alert_history if alert.timestamp >= day_ago
        ]

        return {
            "total_alerts": len(self.alert_history),
            "last_hour": len(recent_alerts),
            "last_24_hours": len(daily_alerts),
            "by_severity": dict(self.alert_counts),
            "recent_alerts": [alert.to_dict() for alert in recent_alerts[-10:]],
        }

    def clear_history(self) -> None:
        """Clear alert history."""
        self.alert_history.clear()
        self.alert_counts.clear()
        self.last_alert_times.clear()



================================================================================
FILE: src/farfan_pipeline/core/orchestrator/resource_aware_executor.py
================================================================================

"""Resource-Aware Executor Wrapper.

Integrates AdaptiveResourceManager with MethodExecutor to provide:
- Automatic resource allocation before execution
- Circuit breaker checks before execution
- Degradation configuration injection
- Execution metrics tracking
- Memory and timing instrumentation
"""

from __future__ import annotations

import asyncio
import logging
import time
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from farfan_pipeline.core.orchestrator.core import MethodExecutor
    from farfan_pipeline.core.orchestrator.resource_manager import AdaptiveResourceManager

logger = logging.getLogger(__name__)


```

```
class ResourceAwareExecutor:
    """Wraps MethodExecutor with adaptive resource management."""

    def __init__(
        self,
        method_executor: MethodExecutor,
        resource_manager: AdaptiveResourceManager,
    ) -> None:
        self.method_executor = method_executor
        self.resource_manager = resource_manager

    async def execute_with_resource_management(
        self,
        executor_id: str,
        context: dict[str, Any],
        **kwargs: Any,
    ) -> dict[str, Any]:
        """Execute with full resource management integration.

        Args:
            executor_id: Executor identifier (e.g., "D3-Q3")
            context: Execution context
            **kwargs: Additional arguments for execution

        Returns:
            Execution result with resource metadata

        Raises:
            RuntimeError: If circuit breaker is open or execution fails
        """
        can_execute, reason = self.resource_manager.can_execute(executor_id)
        if not can_execute:
            logger.warning(
                f"Executor {executor_id} blocked by circuit breaker: {reason}"
            )
            raise RuntimeError(
                f"Executor {executor_id} unavailable: {reason}"
            )

        allocation = await self.resource_manager.start_executor_execution(
            executor_id
        )

        degradation_config = allocation["degradation"]
        enriched_context = self._apply_degradation(context, degradation_config)

        logger.info(
            f"Executing {executor_id} with resource allocation",
            extra={
                "max_memory_mb": allocation["max_memory_mb"],
                "max_workers": allocation["max_workers"],
                "priority": allocation["priority"],
                "degradation_applied": degradation_config["applied_strategies"],
            },
        )

```

```
        start_time = time.perf_counter()
        success = False
        result = None
        error = None

        try:
            result = await self._execute_with_timeout(
                executor_id, enriched_context, allocation, **kwargs
            )
            success = True
            return result
        except Exception as exc:
            error = str(exc)
            logger.error(
                f"Executor {executor_id} failed: {exc}",
                exc_info=True,
            )
            raise
        finally:
            duration_ms = (time.perf_counter() - start_time) * 1000

            memory_mb = self._estimate_memory_usage()

            await self.resource_manager.end_executor_execution(
                executor_id=executor_id,
                success=success,
                duration_ms=duration_ms,
                memory_mb=memory_mb,
            )

            logger.info(
                f"Executor {executor_id} completed",
                extra={
                    "success": success,
                    "duration_ms": duration_ms,
                    "memory_mb": memory_mb,
                    "error": error,
                },
            )

    async def _execute_with_timeout(
        self,
        executor_id: str,
        context: dict[str, Any],
        allocation: dict[str, Any],
        **kwargs: Any,
    ) -> dict[str, Any]:
        """Execute with timeout based on resource allocation."""
        timeout_seconds = self._calculate_timeout(allocation)

        try:
            result = await asyncio.wait_for(
                self._execute_async(executor_id, context, **kwargs),
                timeout=timeout_seconds,
            )
            return result
```

```
        except asyncio.TimeoutError as exc:
            logger.error(
                f"Executor {executor_id} timed out after {timeout_seconds}s"
            )
            raise RuntimeError(
                f"Executor {executor_id} timed out"
            ) from exc

    async def _execute_async(
        self,
        executor_id: str,
        context: dict[str, Any],
        **kwargs: Any,
    ) -> dict[str, Any]:
        """Async wrapper for executor execution."""
        loop = asyncio.get_event_loop()
        return await loop.run_in_executor(
            None, self._execute_sync, executor_id, context, kwargs
        )

    def _execute_sync(
        self,
        executor_id: str,
        context: dict[str, Any],
        kwargs: dict[str, Any],
    ) -> dict[str, Any]:
        """Synchronous execution wrapper."""
        try:
            from farfan_pipeline.core.orchestrator.executors import (
                D3_Q3_TraceabilityValidator,
                D4_Q2_CausalChainValidator,
            )

            executor_map = {
                "D3-Q3": D3_Q3_TraceabilityValidator,
                "D4-Q2": D4_Q2_CausalChainValidator,
            }

            executor_class = executor_map.get(executor_id)
            if not executor_class:
                raise ValueError(f"Unknown executor: {executor_id}")

            executor_instance = executor_class(
                executor_id=executor_id,
                config={},
                method_executor=self.method_executor,
            )

            return executor_instance.execute(context)

        except Exception as exc:
            logger.error(f"Sync execution failed: {exc}")
            raise

    def _apply_degradation(
        self,
```

```
        context: dict[str, Any],
        degradation_config: dict[str, Any],
    ) -> dict[str, Any]:
        """Apply degradation strategies to context."""
        enriched = context.copy()

        enriched["_resource_constraints"] = {
            "entity_limit_factor": degradation_config["entity_limit_factor"],
            "disable_expensive_computations": degradation_config[
                "disable_expensive_computations"
            ],
            "use_simplified_methods": degradation_config["use_simplified_methods"],
            "skip_optional_analysis": degradation_config["skip_optional_analysis"],
            "reduce_embedding_dims": degradation_config["reduce_embedding_dims"],
        }

        if degradation_config["entity_limit_factor"] < 1.0:
            for key in ["max_entities", "max_chunks", "max_results"]:
                if key in enriched:
                    enriched[key] = int(
                        enriched[key] * degradation_config["entity_limit_factor"]
                    )

        return enriched

    def _calculate_timeout(self, allocation: dict[str, Any]) -> float:
        """Calculate execution timeout based on allocation."""
        base_timeout = 300.0

        priority = allocation["priority"]
        if priority == 1:
            return base_timeout * 1.5
        elif priority == 2:
            return base_timeout * 1.2
        else:
            return base_timeout

    def _estimate_memory_usage(self) -> float:
        """Estimate current memory usage."""
        try:
            import psutil
            process = psutil.Process()
            return process.memory_info().rss / (1024 * 1024)
        except Exception:
            usage = self.resource_manager.resource_limits.get_resource_usage()
            return usage.get("rss_mb", 0.0)


class ResourceConstraints:
    """Helper to extract and apply resource constraints in executors."""

    @staticmethod
    def get_constraints(context: dict[str, Any]) -> dict[str, Any]:
        """Extract resource constraints from context."""
        return context.get(
            "_resource_constraints",
```

```
            {
                "entity_limit_factor": 1.0,
                "disable_expensive_computations": False,
                "use_simplified_methods": False,
                "skip_optional_analysis": False,
                "reduce_embedding_dims": False,
            },
        )

    @staticmethod
    def should_skip_expensive_computation(context: dict[str, Any]) -> bool:
        """Check if expensive computations should be skipped."""
        constraints = ResourceConstraints.get_constraints(context)
        return constraints.get("disable_expensive_computations", False)

    @staticmethod
    def should_use_simplified_methods(context: dict[str, Any]) -> bool:
        """Check if simplified methods should be used."""
        constraints = ResourceConstraints.get_constraints(context)
        return constraints.get("use_simplified_methods", False)

    @staticmethod
    def should_skip_optional_analysis(context: dict[str, Any]) -> bool:
        """Check if optional analysis should be skipped."""
        constraints = ResourceConstraints.get_constraints(context)
        return constraints.get("skip_optional_analysis", False)

    @staticmethod
    def get_entity_limit(context: dict[str, Any], default: int) -> int:
        """Get entity limit with degradation applied."""
        constraints = ResourceConstraints.get_constraints(context)
        factor = constraints.get("entity_limit_factor", 1.0)
        return int(default * factor)

    @staticmethod
    def get_embedding_dimensions(context: dict[str, Any], default: int) -> int:
        """Get embedding dimensions with degradation applied."""
        constraints = ResourceConstraints.get_constraints(context)
        if constraints.get("reduce_embedding_dims", False):
            return int(default * 0.5)
        return default
```